Security Analysis of HTML & JS Contents of Alexa top 1M Websites

Brenden Raulerson, Nagendra Posani, Prahathess Rengasamy, Ravi Prakash Giri,
Swarnim Vyas, Vishal Seshadri, Mehrdad Tahmasbi, and Kee Wook Lee

***Abstract*—We developed a system to analyze many websites in Alexa 1M websites from a security perspective. We looked at the HTML and JS separately. There are three classifier trained with known samples and the output of them are used as the security scores. Then, we found the websites with most changes in their scores and detect some security issues.**

1. Introduction

The content changes on a website can be harnessed to analyze the website and whether it has got some malicious content injected in it. As the front end languages grow more powerful each day, they open possibilities for malicious code injection through the front end. Although handling data appropriately at the front end itself is the primary target of the front end languages, this can be harnessed by malicious actors for their benefit leading to content changes that may be malicious. However, any content change can't be considered as malicious and for the same, we need to analyze what exactly has changed and then build a system which can do that for us with least manual intervention. We have built a classifier which takes in a crawled website, and analyzes and outputs a score between 0 and 1 for three sections of the website which are – Obfuscated JS De-obfuscated JS HTML 0 represents benign website and 1 represents the malicious website. We have trained our classifier with X JS features and Y HTML features and with 4000 data set which includes 1000 benign HTML codes, 1000 benign JavaScript codes, 1000 malicious HTML codes and 1000 malicious JavaScript codes. We have tested our classifier with known malicious and benign data sets and it gave XThere are a plethora of research done for the malicious JavaScript detection but the majority of the works in our survey focused on JavaScript without considering the other contents of the websites. This report makes the following contributions: We have related the JavaScript content with the "HTML" contents of the websites to identify the security problems. Static and Dynamic analysis of the JavaScripts

2. Related Work

With the growing rate of browser dependent applications, security analysis of web contents has recently gained much attention. In Zoozle [1] Curtsinger et al. presented a fast efficient static malicious JavaScript detection technique. They hooked into the JavaScript engine of a browser to get the expanded plain JavaScript version of the original code. They used `Detours` [2] binary library to intercept the calls to `compile` function of the JavaScript engine located in the `Jscript.dll` library. This novelty helped them to address the problem of obfuscation which most of the malicious JavaScript codes use. Rick et al. built a tool called CUJO [3] that performs both static and dynamic analysis of JavaScript to prevent drive-by-download attacks. They converted the source code into tokens and passed groups of them to extract Q-grams for the prediction of malicious contents. CUJO is potentially fast in it's static mode and detects 94% of the drive-by-download attacks.

Prophiler [4], a lightweight static JavaScript filter proposed by Canali et al. combines HTML, JavaScript and URL-based features to do a large-scale detection of malicious Web pages. They showed experimentally that their filter was able to reduce the load of traditional dynamic analysis tools by 85% with negligible miss detection rate. They used hand-picked features for their static analysis and on detection of a malicious script, they use `Wepawet` [5] dynamic analyzer to verify their result. Their feature extraction method is similar to ours but in addition to static analysis, we are also performing the dynamic analysis of JavaScript snippets to detect the runtime malicious behaviors of the codes.

Similar to the work of CUJO, Cova et al. presented a system `JSAND` – a static and dynamic features based classification system [5]. `JSAND` combined anomaly detection and emulation to identify malicious JavaScript and perform its analysis automatically. The emulation performed by them exercised the possibly hidden behaviors of malicious contents which they compared with the model they built with normal JavaScript code to detect anomalies. Since Zozzle leverages the existing JavaScript deobfuscation process, its performance is significantly faster than the `JSAND` and `CUJO`.

`EvilSeed` [6] leverages the known malicious characteristics to search the web more efficiently for pages that area likely malicious. Kapravelos et al. presented *Revolver* [7], an automated approach to detection of evasive web-based malware. They built this tool on the basis of the observation that two similar scripts should be classified in the same way by the web-malware detectors irrespective of malicious or benign nature. Their tool was able to detect the JavaScript code similarities on large scale. Similarity computation was done by creating the abstract syntax trees of different versions of the scripts followed by node-sequence matching. The similarity between malicious contents has been further discussed in [8], [9] to detect the variants of

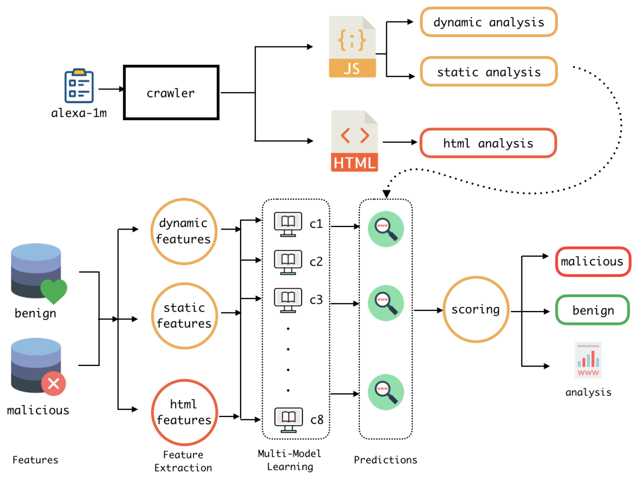

Figure 1: Schematic of Our Solution

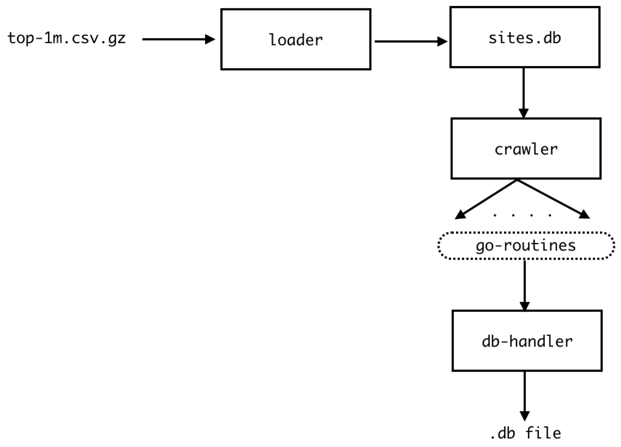

Figure 2: Schematic of the crawler

the same malware family.

Recently Kim et al. presented a crash-free forced execution engine – JForce [10] to identify execution paths of malicious JavaScript code equipped with obfuscated contents. JForce is able to handle or avoid the exceptions encountered during the dynamic analysis of the JavaScript codes and explores all possible paths to expose the malicious JavaScript contents.

MineSpider [11] proposed a technique for extracting URLs from environment-dependent drive-by download attacks. They implemented this as a browser emulator and detected an attack by extracting a large number of URLs while controlling the overhead during the analysis.

The downside that only one concrete execution path in one run is not enough for dynamic analysis techniques to detect the malicious behaviors, multiple symbolic and concolic execution based techniques has been proposed in [12], [13], [14]. The security analysis of JavaScript and web contents have further discussed in [15], [16], [17]

3. Our Solution

3.1. Design

The schematic of our design is depicted in Figure 1. We have performed extensive content analysis of the websites. We crawled various websites across 1 month. We crawled Alexa 1 million websites and developed modules to detect changes. We analyzed the HTML and JavaScript of the website. We extracted 11 HTML features, 11 features for static analysis of JavaScript and 8 features using Dynamic Analysis of JavaScript. We used malware jail to perform dynamic analysis of websites. We perform data analysis on sites whose behaviors are known i.e. whether they are malicious or benign. We train our classifier on these data sets and their corresponding feature values. We fetched these data from several websites like minotaur analysis, Moz and from several popular known libraries like jQuery, Angular and Node. We chose the classifier performing best for each section of feature and their values. At the end, we performed analysis of the data that we had crawled and get content changes with the website over different days. Figure X gives the overall design of our solution. Rest of the subsection explain individual modules in detail.

3.2. Crawler

The language options that we have to write the crawler are C, Python, and Go. We chose to Go over Python and C because the object-oriented compiled language gives better performance over-interpreted language like Python. Go is the new language developed by Google and the true strengths of Go is that succinct, minimalistic and fast. It's not a scripting language like Python or Ruby, it is much low level compared to these languages. Go has been very well designed and optimized for scaling while python has a rich module system making it easy to use different files in one project. Python is very easy to prototype an idea and work on it but when it comes to scalability it is not a good option. Our objective is to write an efficient crawler to crawl Alexa 1 Million websites, to solve this problem scalability and efficiency are our primary goals. Go language supports to write a multi-threaded program making use of maximum resources we have to crawl the websites, which made us decide to write crawler in Go.

The architecture of the crawler is as given in the Figure 2. We have loader, crawler, and database handler modules. First, we load the seed list of URLs into the database and start the crawler. The crawler module gets the URLs from the database and crawls them using goroutines, which is a lightweight thread managed by the Go runtime. Goroutines run in the same address space, so access to shared memory must be synchronized. The database handler stores the crawled data in the database after parsing it into our data structure.

We used the cronjobs on Linux VM to run the crawler daily allocating the maximum CPU usage we were able to crawl the data efficiently. We faced few problems for crawling some of them are listed below. In the initial few runs the crawler took more than 2 days to crawl due to DNS queries but after that, because of the DNS cache, it is able to crawl in 1.5 days. The crawler was not able to crawl few websites some of the reasons we found are given below. Few websites doesn't give permission for the bots to crawl Some of the fake websites or randomly generated websites which are not currently hosted or short-lived websites. This list has about 4220 websites. During the peak hours, our crawler couldn't fetch the number of websites that we intend to crawl using the goroutines. Also, during winter break we were able to crawl more efficiently due to less traffic.

3.3. HTML Features

nagendra [2:49 PM] The program created to perform the html feature ex-traction was written in go. Because of this, several librarieswere able to be used to assist with the work. The body ofthe webpage and its URL were provided to the extractorfor each address crawled. The content was then parsedusing a HTML5-complient tokenizer and parser providedby go. This created a document object model for the pagewhich was then traversed. Nodes were divided betweenelement, text, comment, and document. Much of the focus togenerate features was on the element nodes, although somedid include information from text and comment nodes aswell. For each node in the tree, its type and attributes wereexamined and the relevant features updated, and then anychildren present would be examined. The features of eachpage are output in a file as a list of 11 values, with allpage results concatenated into a single output file. However,the use of the parsing library does have limitations. Theparser does not properly recognize out of place tags, whichprevents the computation of certain features. To do otherwisewould require the construction of our own parser functionand likely lead to significant performance loss due to a lackof optimization. We have used in total 11 features to for the purpose of training. We found these set of features through `Prophiler` [4] and can have a significant correlation to malicious activity.

- *No. of iframe tags.* We count the number of iframe tags detected within a page.
- *No. of hidden tags.* We count the number of elements present in the page that have a hidden attribute. This would not be seen by a visitor to the page but would still be capable of performing actions.
- *No. of elements with a small area.* We count the number of that are considered small, with either an area less than 30 pixels or a width or height less than 2. Similar to hidden elements these would likely not perform a visual purpose and are designed to be unnoticed.
- *No. of script elements.* We count the number of script elements present in a page. This includes scripts both from an external source file as well as those inline within the page.
- *Percent of scripting content in the page.* We compute the amount of scripting content relative to the overall size of the page. More scripting in a page implies less actual visible content while more actions executed.
- *Percent of whitespace in the page.* We compute a number of whitespace characters present in the page. This gives an idea of the textual content of the page.
- *Presence of meta redirect tags.* We determine if the page contains meta tags designed to redirect visitors to a different page.
- *No. of embed or object tags.* These tags can be used to include external objects in a page and allow for the execution of code not actually present in the page itself.
- *No. of elements with an external source.* We count the number of sources hosted on an external domain. Most pages are contained within a single domain, and external sources may not be authenticated.
- *No. of included URLS.* We count the number of elements that are included in a page through a source URL rather than being inline in the page itself.
- *No. of characters.* We count the number of characters present in a page.

3.4. Static JS Features

We provide here brief description of our static features for JS analysis and heuristic behind the selection of each.

- The ratio of the number of keywords to the other words [4]: most of the malicious scripts try to obfuscate their commands and therefore they conain less number of keyword in comparision to the whole number of words
- Number of strings with length greater than 40 [4]: many obfuscation techniques use long strings
- Number of suspicious tags [4]: these tags contain *script*, *object*, *embed*, and *frame*. These tags can be used to write other scripts or object into the website.
- Number of *iframe* tag [4]: many malicious scripts inject many *iframe* into the website
- Entropy of the script: the malicious obfuscated scripts look more random and their entropy is higher
- Number of suspicious strings [4]: some malicious scripts use names such as *evil*, *shell*, *spray*, and *crypt*.
- Number of functions *clearAttributes*, *insertAdjacentElement*, and *replaceNode*
- Number of *decodeURIcomponent*
- Number of *setTimeOut*
- Number of *exec*
- Number of *applet* and *script*

3.5. Dynamic JS Features

Similar to static analysis, we selected some set of features for dynamic analysis and trained our classifiers withknown malicious and benign samples Feature Extraction:.We have used in total 8 featuresto for the purpose of training. We found these set of featureson the fly and found some of these strongly correspondingto malicious JavaScript characteristics.

- No. of dynamic Function Calls.We havecalculated the number of dynamically executed functioncalls that have been used to to interpret the JavaScript codeand DOM changes (e.g.document.write). Also used inJSAND[1].
- No. of Wscript saved files.We are quantifyingthe no. of files that the malware attempted to drop. These areprimarily wscript files for Windows OS dependent systems.
- No. of URLs.All URLs that the mal-ware intended to GET or POST (for e.g. usingdocument.location). Also used inJSANDhoweverthey only considered target URLs.
- :No.of Wscript objects.We are also keepingtrack of WScript or ActiveX objects created.
- No.ofsetTimeout()calls.No. of timesthis function is called for potential JavaScript time-bomb
- No.of eval() calls.Array of alleval()callsarguments. Useful ifeval()is used for deobfucation.
- No.of unescape() calls.No.of dynamic callsto this function. To decode the hexadecimal values back toplaintext. Generally used in obfuscated codes.
- No.of browser documents.Count of thebrowser documents - HTML document, XML document,WML document etc. Mostly in benign

Dynamic analysis of JavaScript malware focuses on two aspects in our literature survey. First, by generating inputs and triggering the registered event handlers in various orders for the purpose of path exploration. There are two different approaches for this purpose - Fuzzing and Symbolic execution [12], [18]. Second, by providing a special environment for the malware to execute by providing a highly simulated browser context as discussed in [19]. Such an environment is also called *JavaScript Sandbox* or *JavaScript Jail*. For the purpose of the brevity, we will focus on the later case. JavaScript Sandbox provides JavaScript interpretation and DOM emulation for the simulated execution of the malicious JavaScript contents. We are using Malware-Jail [20] - a Sandbox for the analysis of malicious JavaScript. The targeted scripts are executed in the Sandbox and all runtime characteristics are observed. The Sandbox is built on the `VM` module supported by Node.js.

This tool introduces the HTML and DOM emulation into the Sandbox which in turn introduces the implementation of methods and crucial members according to the specification referred to by the modern browsers. Similar to the static analysis of the JavaScript we feed the results of dynamic analysis into the classifier.

Supervised	Unsupervised
Linear Regression Logistic Regression Decision Tree SVM Naive Baysian K Nearist Points Random Forest	K means

Figure 3: Classification Algorithms

	Bening	Malicious
HTML analysis	998	467
Static JS analysis	363	353
Dynamic JS analy	377	648

Figure 4: Number of Samples for Training

3.6. Classification

In this section, we explain the high level overview of our classification approach. We have three classification problems, HTML, static JS, and dynmaic JS, but we elaborate the HTML classification, and the other two are excactly same. If $\mathbf{s}$ is a HTML script, let $f_H(\mathbf{s})$ denote HTML feature vector which can be represented by a point $\mathbf{x} = (x_1, \cdots, x_n)$ in the Eculidan space $\mathbb{R}^{11}$. In general, a classifier is function $h_\theta : \mathbb{R}^{11} \to [0, 1]$ where θ denote the classifier parameters needed to be trained. It takes the feature vector as input, and outputs the probability that the script corresponding to the feature vector is malicious.

For training, we start with two sets of scripts $\mathcal{B}$ and $\mathcal{M}$ containing benign and malicious scripts respectively. We use 80 percents of all scripts randomly selected from $\mathcal{B} \cup \mathcal{M}$ to learn the parameters θ. Afterward, we use the remaining 20 percents to test the classifier. In particular, we are choosing one threshold $\gamma \in [0, 1]$ and script $\mathbf{s}$ is specified as malicious if and only if

$$h_\theta(f_H(\mathbf{s})) \geq \gamma. \quad (1)$$

Then, we can find the percentage of malicious scripts detected as being (false negative or missed detection probability) and the percentage of benign scripts detected as malicious (false positive or false alarm probability). Finally, by changing the threshold from 0 to 1 we obtain true positive vs. false positive rate graph is known as ROC curve.

We tried seven supervised classifiers and one unsupervised classifier given in Table 3 and choose the one whose performance is the best at the desired false positive rate. In this regard, we used the library *scikit:learn* for Python [21].

4. Results from the Experiment

4.1. Training Results

The number of samples used for training of each one is mentioned in Table 4. We used high-rank websites as

	Precision
HTML analysis	0.013
Static JS analysis	0.019
Dynamic JS analy	0.016

Figure 5: Precision of the sum of missed detection and false alarm probability

HTML	Static JS	Dynamic JS
5.57188101e-01	9.55054758e-01	9.59783101e-01
2.04914912e-01	4.40523605e-02	3.97451591e-02
1.23142810e-01	7.10620998e-04	3.96656027e-04
6.32385679e-02	8.11914934e-05	5.46455120e-05
4.98418303e-02	6.04686929e-05	8.13571882e-06
1.67371881e-03	2.99395453e-05	4.61051455e-06
2.13409879e-08	1.06607646e-05	4.09821016e-06
1.66531674e-08	1.04882930e-34	3.59420853e-06
9.30085591e-09	2.86142599e-37	
8.02479635e-09	5.90031651e-38	
4.74976890e-09	3.03516707e-41	

Figure 6: PCA data

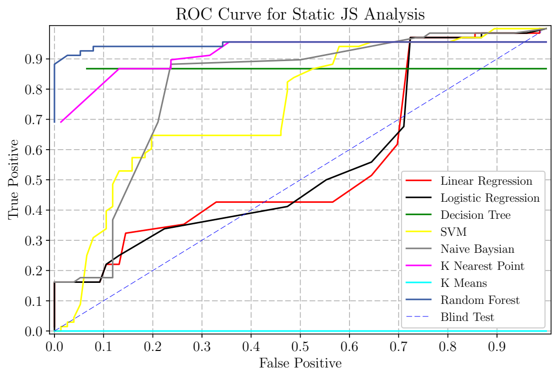

Figure 7: ROC curve for static JS analysis

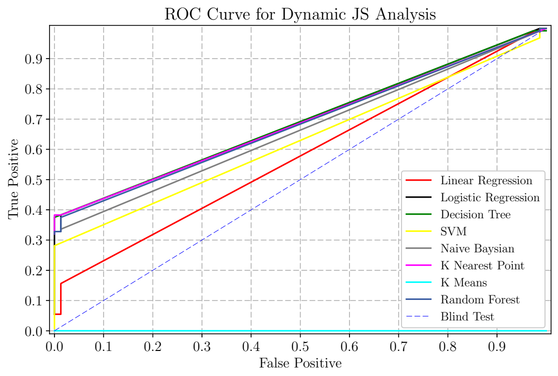

Figure 8: ROC curve for dynamic JS analysis

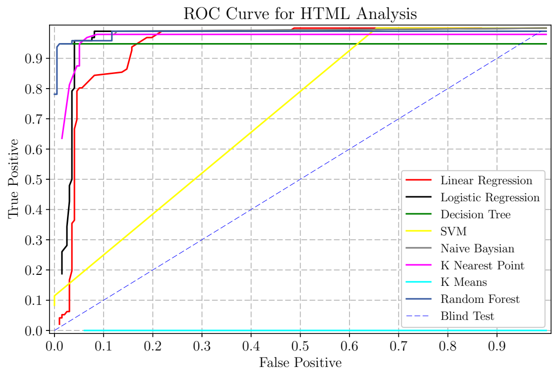

Figure 9: ROC curve for HTML analysis

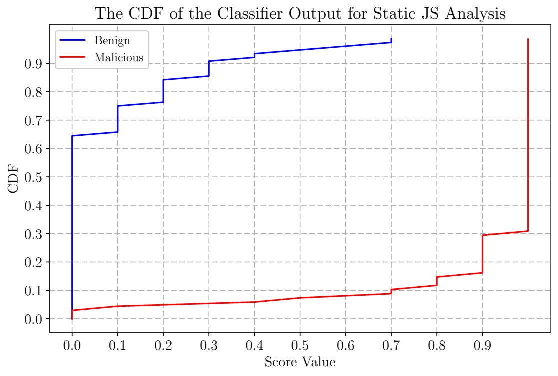

Figure 10: CDF of the classifier output for static JS analysis

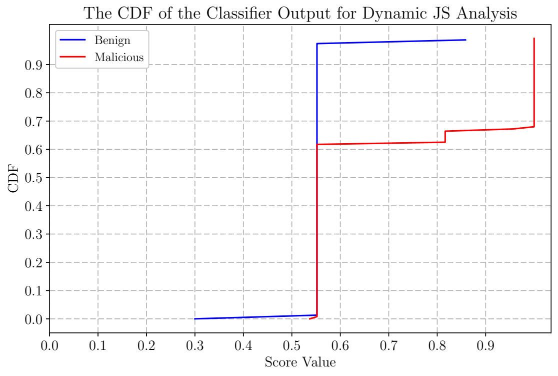

Figure 11: CDF of the classifier output for dynamic JS analysis

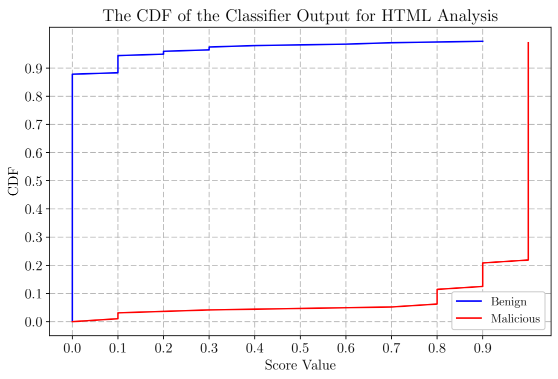

Figure 12: CDF of the classifier output for HTML analysis

benign samples, and malicious samples are obtained from [22]. The ROC curves corresponding to static and dynamic JS analysis, and HTML analysis are shown in Figure 7, Figure 11, and Figure 9 respectively. From the ROC curve, we can conclude that for low false positive rates, *Random Forest* classifier is the best one for HTML analysis and static JS analysis, and *Decision Tree* classifier is the best one for dynamic JS analysis. Moreover, the CDF for the output of the aforementioned classifiers are given in Figure 10, Figure 11, and Figure 12, Figure 10, and Figure 11 respectively. Note that the reason that ROC curve for dynamic JS analysis, is different is that for some malicious samples and all benign samples all features are zero. Hence, there is no *possible* way to distinguish those malicious samples based on our features. However, our features are useful for detecting some malicious JS.

The precision of the data in ROC curves can be approximated by Central Limit Theorem (CLT). More precisely, if $X_1, \cdots, X_n$ are n i.i.d. random variables with mean μ and variance σ^2, CLT yields that for n large, the average $\frac{1}{n}\sum_{i=1}^{n} X_i$ has distribution very close to the distribution of Gaussian random varialbe with mean μ and variance $\frac{\sigma^2}{n}$. Therefore, with probability around 0.95, we have

$$\frac{1}{n}\sum_{i=1}^{n} X_i \in \left[\mu - 2\frac{\sigma}{\sqrt{n}}, \mu + \frac{2\sigma}{\sqrt{n}}\right]. \tag{2}$$

Using this result for the sum of probabiliy false alarm and missed detection, and the fact that for a random variable $X \in [0,1]$, $\text{Var}(X) \leq \frac{1}{4}$, we obtain that if we have n benign and malicious samples in total, then the precision of our data is about $\frac{1}{2\sqrt{n}}$. The exact values are reflected in Table 5

Finally, to know how much information we have in our features, we applied PCA to our training data. The normalized variance of the principal components is given in Table 6. The results suggest that HTML features are approximately in 6-dimensional linear subspace; static JS features are approximately in 7-dimensional space; dynamic JS features are approximately in 4 dimensional space.

day1	08-03-2017	Wednesday
day2	20-03-2017	Monday
day3	23-03-2017	Thursday
day4	26-03-2017	Sunday
day5	30-03-2017	Thursday
day6	02-04-2017	Sunday

Figure 13: Information about days we crawled

4.2. Testing Results

We used the crawled data for six days. The information about these days are given in Table 13. For each URL and each day, we find three scores which are the output of the three classifiers for HTML analysis, static JS analysis, and dynamic JS analysis. Then, to observe changes in the security context of the websites, we are looking at two quantities: 1) the variance of each score for each website during the 6 days; 2) the absolute value of the differences between the scores for consecutive days. Histogram and CDF of these quantities are given in the next page. Then, 20 URLs with most changes are derived and given in Table **??**. These websites are manually analyzed to find security issues.

At the end, we set some threshold for the classifiers to get the false positive rate less than 0.05 and classified all websites. These results are given in Table 14.

4.2.1. Security Issues.

- *Day 1 to Day 2 in webdade.com*: Lot of scripts in day 1 vanished in day 2 and there were several calls in day 1 that point to www.smartsuppchat.com which is a blacklisted website across multiple blacklists
- *Day 3 to Day 4 in webdade.com*: The references are gone and the website is back to normal.
- *Day 1 to Day 2 in Torrent-ultra.com*: Day two contained some hidden inputs that are script generated trying to change some privacy settings of the user in privacy.html this may be to get access to restricted information about the user.
- *Day 3 to Day 4 in Torrent-ultra.com* Malicious scripts were gone and there is no hidden input or reference to any sort of privacy settings
- *Day for in show play.tk* This website takes user cookies and references findbetterresults.com a blacklisted website several. Times based on user interaction with the website to send cookie information divulging sensitive user data.
- *In bigjapeneseporn.com:* the scripts tags reference many external sources, spawning many popups which tries to click bait the user into navigating into external domains. From day 3, the structure of the website has changed, with inline JS being modified, dropping many excessive popup spawning functions. The image element injected using document.write function is modified post day 3. Also, day 3 onwards the inline JS is modified to include document.write

Days	day1	day2	day3	day4	day5	day6
Malicious HTML	461	458	453	405	415	396
Malicious Static JS	71	66	68	69	71	67
Malicious Dynamic JS	345	346	346	353	351	353
Malicio Overall	833	824	822	782	790	772

Figure 14: Number of malicious cases found

Variance	Difference
radio24syv.dk	creation-compte.com
xxxaloha.com	creation-compte.com
yingyujiaoxue.com	creation-compte.com
adpornmedia.com	creation-compte.com
hqindiantube.com	adpornmedia.com
webdade.com	hqindiantube.com
bigjapaneseporn.com	webdade.com
ichvideo.com	ichvideo.com
starone.org	starone.org
torrent-ultra.com	torrent-ultra.com
qylbbs9.com	qylbbs9.com
xxxyounghub.com	shekulli.com.al
shekulli.com.al	shoowplay.tk
filmeleporno.xxx	shoowplay.tk
filmeleporno.xxx	shoowplay.tk
shoowplay.tk	shoowplay.tk
hardhentaisex.com	khafan.net
brmethodprofit.com	brmethodprofit.com
khafan.net	brmethodprofit.com
sexyoungsex.com	brmethodprofit.com

Figure 15: URLs with most changes in terms of our security scores

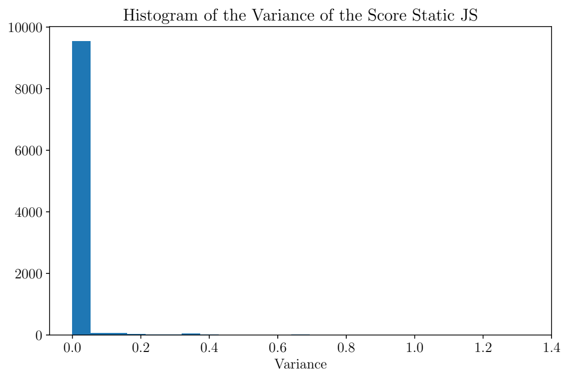

Figure 16: Histogram of the variance in the score of static JS analysis

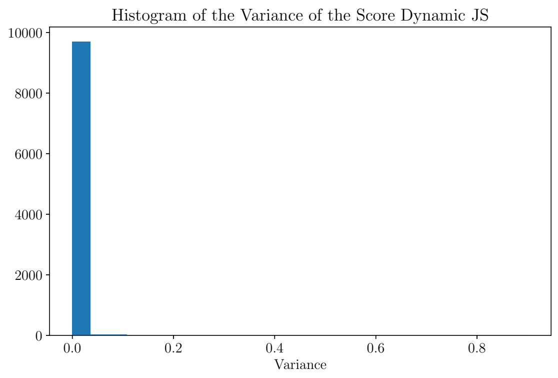

Figure 17: Histogram of the variance in the score of dynamic JS analysis

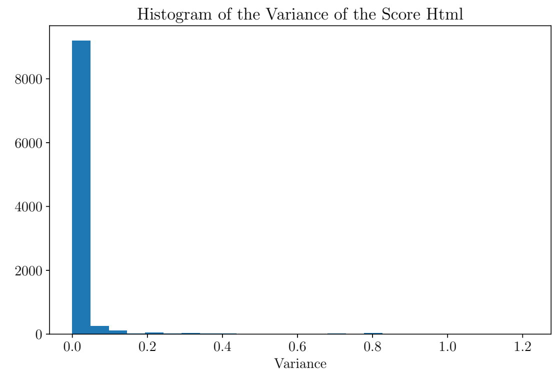

Figure 18: Histogram of the variance in the score of HTML analysis

for the onclick function of the popups. The site has extensive javascripts which donot follow same source origin. This could be due to ads, but one link of the domain source was flagged against one of the known blacklist.

- *In creation-compte.com:* service was unavailable for the first 2 days and then again on the 4th day. On the third day, the script includes a transparent iframe, which is sourced from an external domain hosted on godaddy servers. As mentioned in class, such hosting services, are easy to obtain with simple credit card information. The legitimacy of the site is not verified based on its apparent purpose. The site undergoes another offline session on day 4,

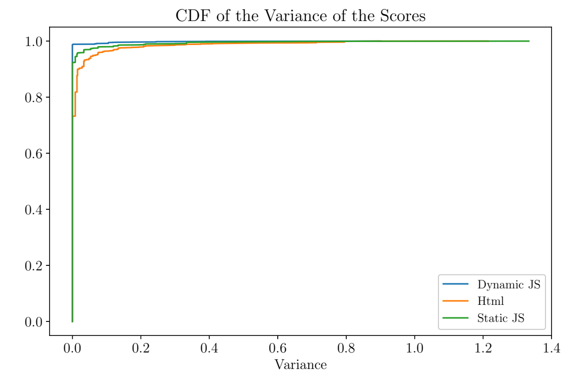

Figure 19: CDF of the variances in the scores

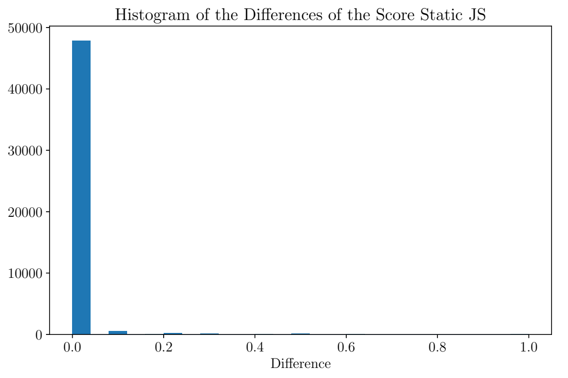

Figure 20: Histogram of the difference in the score of static JS analysis

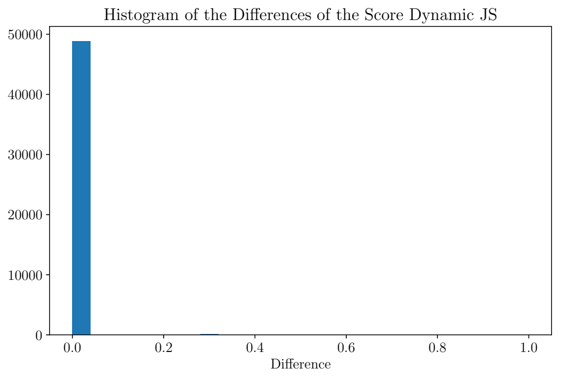

Figure 21: Histogram of the difference in the score of dynamic JS analysis

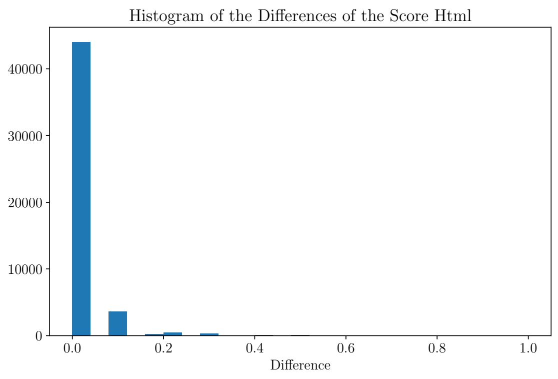

Figure 22: Histogram of the difference in the score of HTML analysis

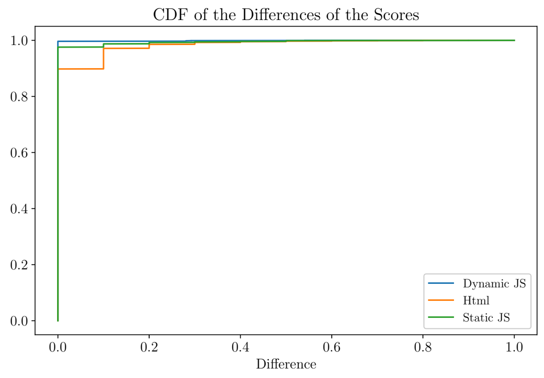

Figure 23: CDF of the score changes

followed by modifications to its inline javascripts, which now contains an image element of unobservable size(which is one of the features in malicious HTML), again referencing an external domain. This peculiar inclusion of the img element is questionable and empirical data has shown, that this could be a source of vulnerability.

- *In shoowplay.tk:* this websites take user's cookies and references findbetterresults.com a black listed website in several places. Times based on user interaction with the website to send cookie information divulging sensitive user data.

5. Conclusion

Content changes on the web page can severely affect the security of the website. Analysis on HTML and JavaScript provides deep insight into the content modifications that take place on a website and how they are related with the security of the site. We present in this report, our detailed analysis that spanned over feature extraction and their analysis, both static and dynamic. Our classifier has produced reasonable results against known benign and malicious samples.

References

[1] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, “Zozzle: Fast and precise in-browser javascript malware detection,” in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC’11. Berkeley, CA, USA: USENIX Association, 2011, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028070

[2] G. Hunt and D. Brubacher, “Detours: Binary interception of win32 functions,” in *Proceedings of the 3rd Conference on USENIX Windows NT Symposium - Volume 3*, ser. WINSYM’99. Berkeley, CA, USA: USENIX Association, 1999, pp. 14–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268427.1268441

[3] K. Rieck, T. Krueger, and A. Dewald, “Cujo: Efficient detection and prevention of drive-by-download attacks,” in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC ’10. New York, NY, USA: ACM, 2010, pp. 31–39. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920267

[4] D. Canali, M. Cova, G. Vigna, and C. Kruegel, “Prophiler: A fast filter for the large-scale detection of malicious web pages,” in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW ’11. New York, NY, USA: ACM, 2011, pp. 197–206. [Online]. Available: http://doi.acm.org/10.1145/1963405.1963436

[5] 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*, ser. WWW ’10. New York, NY, USA: ACM, 2010, pp. 281–290. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772720

[6] L. Invernizzi, S. Benvenuti, M. Cova, P. M. Comparetti, C. Kruegel, and G. Vigna, “Evilseed: A guided approach to finding malicious web pages,” in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 428–442. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.33

[7] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna, “Revolver: An automated approach to the detection of evasiveweb-based malware,” in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC’13. Berkeley, CA, USA: USENIX Association, 2013, pp. 637–652. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534821

[8] J. Jang, D. Brumley, and S. Venkataraman, “Bitshred: Feature hashing malware for scalable triage and semantic analysis,” in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS ’11. New York, NY, USA: ACM, 2011, pp. 309–320. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046742

[9] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, “Efficient detection of split personalities in malware.” in *NDSS*, 2010.

[10] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, “J-force: Forced execution on javascript,” in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW ’17. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017, pp. 897–906. [Online]. Available: https://doi.org/10.1145/3038912.3052674

[11] Y. Takata, M. Akiyama, T. Yagi, T. Hariu, and S. Goto, “Minespider: Extracting urls from environment-dependent drive-by download attacks,” in *Proceedings of the 2015 IEEE 39th Annual Computer Software and Applications Conference - Volume 02*, ser. COMPSAC ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 444–449. [Online]. Available: http://dx.doi.org/10.1109/COMPSAC.2015.76

[12] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A symbolic execution framework for javascript,” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 513–528. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.38

[13] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, “Jalangi: A selective record-replay and dynamic analysis framework for javascript,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 488–498. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491447

[14] K. Sen, G. Necula, L. Gong, and W. Choi, “Multise: Multi-path symbolic execution using value summaries,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 842–853. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786830

[15] S. Lekies, B. Stock, M. Wentzel, and M. Johns, “The unexpected dangers of dynamic javascript.” in *USENIX Security*, 2015, pp. 723–735.

[16] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee, “Understanding malvertising through ad-injecting browser extensions,” in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW ’15. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2015, pp. 1286–1295. [Online]. Available: https://doi.org/10.1145/2736277.2741630

[17] C. Neasbitt, B. Li, R. Perdisci, L. Lu, K. Singh, and K. Li, “Webcapsule: Towards a lightweight forensic engine for web browsers,” in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15. New York, NY, USA: ACM, 2015, pp. 133–145. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813656

[18] G. Li, E. Andreasen, and I. Ghosh, “Symjs: Automatic symbolic testing of javascript web applications,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 449–459. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635913

[19] A. Dewald, T. Holz, and F. C. Freiling, “Adsandbox: Sandboxing javascript to fight malicious websites,” in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC ’10. New York, NY, USA: ACM, 2010, pp. 1859–1864. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774482

[20] Malware-Jail, “https://github.com/hynekpetrak/malware-jail/.” in *GitHub*, 2017.

[21] scikit learn, “http://scikit-learn.org/stable/.” in *GitHub*, 2017.

[22] js-malicious dataset, “https://github.com/geeksonsecurity/js-malicious-dataset.” in *GitHub*, 2017.