THE UNIVERSITY OF CHICAGO

JAVASCRIPT CLASSIFICATION: AN EMPIRICAL ANALYSIS OF UNTRUSTED CLIENT-SIDE JAVASCRIPT

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE THESIS DIVISION
IN CANDIDACY FOR THE DEGREE OF
MS

DEPARTMENT OF COMPUTER SCIENCE

BY
MINHAJ US SALAM KHAN

CHICAGO, ILLINOIS
GRADUATION DATE
Dedication Text
Epigraph Text
# TABLE OF CONTENTS

LIST OF FIGURES ......................................................... vi
LIST OF TABLES ......................................................... vii
ABSTRACT ................................................................. viii
1 INTRODUCTION ......................................................... 1

2 BACKGROUND ......................................................... 5
   2.1 Transparency Logs .............................................. 5
   2.2 Code similarity comparison .................................. 5

3 CRAWLING ............................................................. 7
   3.1 Approach ......................................................... 7
   3.2 Implementation ................................................ 8

4 SITE SIGNATURES .................................................... 10
   4.1 Choosing a Signature Scheme ................................. 10
   4.2 Using AST to Generate Signature ............................ 11
   4.3 Constructing a signature ...................................... 12
   4.4 Implementation ................................................ 14
   4.5 Example .......................................................... 15

5 PAGE TEMPLATES ..................................................... 19
   5.1 Clustering ........................................................ 19
      5.1.1 Agglomerative Clustering ............................... 20
      5.1.2 DBSCAN Clustering ..................................... 20
      5.1.3 URL Based Clustering .................................. 21
   5.2 Evaluating the Clusters ...................................... 21
   5.3 Least Common Structure ..................................... 22

6 EVALUATION .......................................................... 24
   6.1 Ground Truth ................................................... 24
   6.2 Results .......................................................... 25
      6.2.1 Crawling .................................................... 25
      6.2.2 Clustering .................................................. 27

7 ANALYSIS ............................................................. 32
   7.1 Crawling ........................................................ 32
   7.2 Clustering ....................................................... 32

8 CONCLUSION AND FUTURE WORK ................................. 37
# LIST OF FIGURES

1.1 logging phase ................................................. 3  
1.2 verification phase ............................................. 4  

4.1 Sample Tree .................................................. 12  
4.2 Node types .................................................... 16  
4.3 Feature Vector: Node count .................................. 16  
4.4 Feature Vector: Depth of node ............................... 17  
4.5 Feature Vector: Size of subtree .............................. 17  
4.6 Node list ...................................................... 17  
4.7 Feature vector for hybrid script ............................. 18  

5.1 Example of what an AST in the LCS might look like .............. 23  
5.2 High level structure of LCS of Page Template .................. 23  

6.1 Results of Clustering using Node count ........................ 28  
6.2 Results of Clustering using Node-Depth ........................ 29  
6.3 Results of Clustering using Sub-Tree size ...................... 30  
6.4 Results of Tracking changes in clusters over time ............. 31
LIST OF TABLES

6.1 Unique links discovered .............................................. 26
ABSTRACT

As web applications are increasingly used for security and privacy-sensitive activities, there is growing concern that the providers of those products could act maliciously due to compromise or coercion. Most webpages are actually JavaScript programs. Currently, when a user visits a website, the JavaScript served by that website is inherently trusted by all of the commodity web browsers. This creates a problem where a compromised server can serve malicious JavaScript just to a small subset of clients without the possibility of detection. In an ideal world, there would exist a system that counters this by building a log that allows the clients to verify that the code they are being served is the same as the code served to other clients. In this paper we present the results of an empirical study of the nature of JavaScript on the web with the objective of determining the possibility of building the aforementioned log. To this end, we use an instrumented version of Google’s Chromium browser to fetch a large corpus of JavaScript samples served by popular websites. We then test various methods for classifying differences between variants of JavaScript code. Each of these methods differ in the methods of feature vector construction from JavaScript code and the choice of clustering algorithms. Early results of the classification experiments suggest that constructing a system that allows a client to validate the JavaScript that it receives is a hard problem due to the various inherent properties of JavaScript.
CHAPTER 1
INTRODUCTION

Websites serve JavaScript code that runs in users’ browsers. When a client connects to a server, it can verify the identity of the server using a protocol called Transport Layer Security (TLS), which ensures that the client can reasonably trust that the server with which it is communicating is not equivocating about its identity. However, no such protocol exists for the client to verify the authenticity of the actual JavaScript being served that will run inside the client’s browser. In fact, the only existing security that browsers offer for JavaScript code is sand-boxing which means that the code is excluded from accessing resources outside of the origin it was loaded in. An origin for a webpage is the list of domains from which it can safely load resources. However there is no protection against a piece of JavaScript code behaving maliciously within the context it was loaded in despite a very real threat that the code received might be malicious. The web server itself might be malicious and intentionally sending harmful code to the clients. The server might also have been compromised using a plethora of attacks available to the modern adversary. There is also a chance that a website is coerced by a powerful agency like a nation state to send malicious code to all or a subset of clients. Furthermore, in recent times the types of services being offered over the web have diversified and therefore so have the functions carried out by the JavaScript code served by websites that runs in a user’s browser. Whereas in the past client side code was largely used for things like generating the page’s HTML code or dynamically updating the style of elements within a page, the client side code now frequently handles private data and carries out security sensitive functions like encryption and decryption. This is especially true of cloud based services like Google Apps and Dropbox, making the damage a malicious JavaScript code can cause more severe.

Therefore, clients should be able to detect whether they are being served suspicious code. One way to do so is to compare the code that is being received against some expected version
of the code. This expected version of the code would represent the JavaScript that an honest version of the website serves to the clients. This approach is illustrated in figure 1.1 and figure 1.2. Figure 1.1 shows the first step where a client after receiving the code logs it with a third party service. The idea is that different clients will each provide a version of the site they have received. The log will then accept the version that has the most consensus. Figure 1.2 shows the second phase in which a client can verify the code it receives using the logged version. Here we assume that the server is acting maliciously whether of its own volition or due to coercion or compromise.

The challenge in this approach is that the JavaScript code that a client receives is rarely the same as the ones received by the same client at some different instance of time let alone to the one being received by other clients. More often than not, the code contains dynamic values such as user IDs and timestamps that may differ between versions of the same code whilst preserving the core functionality. Therefore, it is difficult to come up with a method to effectively classify differences between different variants of the same JavaScript code. A solution could be to create signatures for different web pages such that all the different versions of JavaScript belonging to that page have the same signature. More formally, a signature for a webpage will be a unique representation of a particular instance of that webpage that will allow the client to determine whether the code that it received is the same as the code served by the server for that webpage at a different time.

Furthermore, whereas a website may have a very larger number of pages, those pages are generated from a much smaller set of templates. These templates represent skeleton JavaScript code that is used by a particular website to generate a set of functionally similar pages. So, once we have the signatures as described above, we build a system to group together these signatures into distinct groups, where each group contains signature from web pages that have essentially the same code. This will allow the client to match the new pages that it receives against a expected templates to ascertain which template generated that
particular page. The client can then perform a more fine-grained analysis to determine the nature of any differences between the template and the new page that it has received.

In particular we try to answer the following three questions in the rest of this paper:

- Is it possible to build a classification system for JavaScript code samples that accurately identifies the page templates from which the webpages of a particular website are generated?
- Given a group of web pages is it possible to construct a unique representation over that whole group?
- Is it possible to track how the JavaScript structure of a webpage or a group of webpages evolves over time?
Figure 1.2: verification phase
CHAPTER 2

BACKGROUND

2.1 Transparency Logs

The use of transparency to provide security against equivocating parties in distributed systems and cloud computing is not new and has been applied in a variety of contexts. Perhaps the most well-known of these is Google’s Certificate Transparency (google) project which uses a publicly visible tamper evident log to track the SSL certificates being issued by Certificate Authorities so that wrongly issued or maliciously acquired certificates may be identified. Later it was shown how certificate transparency can be extended and used to build an end to end messaging or email service using PKI (D, 2014). Another major work in this area implements the idea of transparency logs for software binary codes to allow monitoring and auditing of software provider activity (D u. a.). Similarly, CONIKS (S u. a., 2013) uses the idea of tamper evident logs for key verification in an end-to-end encrypted communication system. However, there is no such transparency system for the JavaScript served by websites to ordinary clients on the web.

2.2 Code similarity comparison

There has also been some interesting work done regarding the use of code signatures to help detect changes between different versions of the code. One of them, a W3C recommendation called Subresource Integrity (SRI) (w3c), is widely deployed in modern browsers. SRI ensures that a website does not load any script unless its hash digest matches some expected value. However, since SRI uses digests of raw source code, it is not suitable for the highly malleable JavaScript that is served to clients on the web. Another set of approaches uses the Abstract Syntax Tree of a piece of JavaScript code (Jones). An example would be SICILIAN (Soni u. a., 2015) which uses JavaScript signatures to detect code injections by comparing the
code signature of the web page against the expected signature maintained by the system. It extracts the JavaScript AST and computes the signature over it instead of doing so over the textual representation of the code. Another example is that of Revolver (A u. a., 2013), which employs JavaScript signatures to detect evasive web based malware. Their system is different from SICILIAN because instead of using the AST as it is, it uses pre-order traversal of the AST to construct a sequence of nodes. They use the number of times each node type appears in the corresponding AST to map each AST on to an n-dimensional Euclidean space, which they combine with k-nearest neighbour algorithm to identify similar scripts. As described in further detail in a later section, this was the basis of the first method we tried out in our study. The results, whilst being good enough for Revolver, were not precise enough for what we were trying to accomplish.

Finally, there exists a considerable amount of literature on identifying changes between samples of non-JavaScript code (K. u. a., 2009). These techniques have been developed over time to tackle a number of challenges, especially in the Software engineering community. Some of these techniques use ASTs to track software evolution over time. Most of these are unsuitable for our purposes because they rely on principles such as static function names which do not apply to client-side JavaScript. Next, there are techniques such as MOSS (Saul u. a., 2003), which aim to detect plagiarism by comparing two code samples. Unfortunately, these are just comparison tools and do not compute any metric over a single script. Therefore they do not provide any intuitive way of creating a signature which helps to uniquely identify a particular script.
CHAPTER 3
CRAWLING

The first step in the study was to collect a significant amount of JavaScript code samples so that we can perform an analysis of both the nature of JavaScript being served by the popular websites, and test the classification methods described later in the paper. In particular we collected samples over a six month period to see how the JavaScript changes over that period.

3.1 Approach

Crawling was performed to collect the JavaScript samples. To this end, 36 different websites were visited in an automated manner, and at each visit the JavaScript served by the page being visited was stored in a database. To ensure that we capture the different versions of JavaScript served to different users, we created user accounts on the aforementioned websites.

For this experiment, crawling is done in two phases. The first phase is the one in which new URLs are extracted. This was capped to a depth of three. A depth of three here means that the homepage of a website is crawled, the links extracted from that page are crawled and the links extracted from all of those pages are crawled. In the second phase, which is performed immediately following the first phase, the URLs that have already been discovered are crawled once again to capture any temporal changes in the JavaScript. Furthermore, a lot of JavaScript on the web is loaded in response to how the user interacts with various elements of the webpage. Therefore, the second phase also makes use of monkey testing to discover new JavaScript. Monkey testing is the method of testing a system by supplying random inputs. For our purposes, we used monkey testing to mimic user interaction with various elements of a webpage to trigger more JavaScript to be loaded, which we then also stored in our database.
3.2 Implementation

To collect the samples, an instrumented version of Google’s Chromium web browser was used. The source code of this browser is open source and can be downloaded, modified and compiled using a custom build configuration. For this project, we modified Chromium’s `executeScript` function, which loads and executes each individual script received from a web server, so that before a script is executed, the raw script and its metadata is logged. The metadata of each script together with its SHA-256 digest is stored in a JSON file whereas the raw JavaScript is stored in an instance of LevelDB (leveldb), which is a database that stores key value pairs. The LevelDB stores each script keyed with its digest. To crawl the webpages and collect samples from them, Selenium (selenium) web driver is used. The selenium web driver makes calls to the Chromium (chromium) browser using its native support for automation. To use Selenium, we wrote a script in Python which uses Selenium to drive the custom-built Chromium browser and navigate to each URL, cache the scripts and extract all the links from the page. Each of these extracted links is then in turned crawled in a similar manner.

To implement monkey testing we used a library called gremlins.js (gremlins) that provides efficient interface for applying monkey testing to a web browser. This allows the discovery of URLs that were missed in the first phase and new JavaScript loaded in response to scrolling the page and clicking on random elements. However, there still remains some JavaScript that could not be archived because the HTML element containing that script could not be focused by selenium.

To get a good measure of how accurate the classification methods discussed later in this paper are, we decided to crawl hundreds web pages. Not unlike most of the academic studies that crawl the web, we turned to Alexa (Alexa) to obtain the list of websites. Each website in our study needed to fulfill certain criteria: they needed to have different ‘types’ of web pages, there should be an option to create a user account, and the web site should serve some level of personalized content to each users. We found 36 of the top ranked web sites that fit
this criteria. We also added some popular sub-domains of google, namely docs.google.com and play.google.com to our sample list for reasons explained later in the paper. Table 1 lists all the domains that were crawled. To scale the crawl to multiple sites, the code is hosted and executed on an instance of Amazon’s AWS server. The server is able to run multiple threads with each thread running multiple instances of the instrumented Chromium browser. This ensures that at any given point in time, a fixed number of sites are being crawled. A queue is implemented that contains all the sites to be crawled. The queue insures that whenever one of the sites completes the crawl, the next site in the queue is selected to be crawled, and the site that just finished crawling is added to the end of the queue. Finally, we followed certain best practice guidelines when conducting a crawl of such scale. These guidelines are listed on the website https://security-research.cs.uchicago.edu/projects/web-transparency/. The link to this website is listed in the crawler’s user-agent. Every account we create on the websites being crawled was clearly labeled as being created by the UChicago Distributed Systems Security Research Group. It was also associated with either the web-transparency@cs.uchicago.edu email address or another email address that forwards to it. The website operators also have an option to contact us on this email if they want to opt-out of the crawling. We also parse the robots.txt file of a web server if present and obey the rules laid out in the file as much as we can. Finally, we try our best to not place undue load on any server by rate limiting our crawls and not reloading pages more than twice in case of load failures.
CHAPTER 4
SITE SIGNATURES

Once the samples were collected, the next step in the process was to construct a signature over each sample. A signature here refers to a representation of a web page that is unique to that web page. The signature must also be similar for web pages that have functionally identical JavaScript. Functionally identical here means that given two web pages, the only differences between their JavaScript code are the data values and names which do not affect the overall flow of the program. More precisely given two functionally identical JavaScripts their ASTs have the same flow. Which means that for a given input, if in Tree T1, the flow from source A ends in sink B, then in Tree T2 not equal to T1, the same input produces a flow from source A that also ends at sink B. For this to be true, T1 and T2 can only differ in data nodes. This section describes why an Abstract Syntax Tree (AST) (Jones) based signature scheme was chosen for this project and how it was used to construct signatures over the collected JavaScript samples.

4.1 Choosing a Signature Scheme

Any signature scheme that would be used for the purpose of this project would need to have certain desirable properties. Firstly, the signature must not merely be an identifier for an individual script but should also be an intrinsic measure of that script. This is because given two scripts with distinct signatures we do not need to make a binary decision about whether they are equal or not but instead need to know how similar or different the two are. Secondly, there must be a method for representing a group of scripts succinctly using a version of the signature over that entire group so that any new script could be compared to that whole group in an efficient manner. Furthermore, the signature should not significantly change for non-functional changes in the script. This means that if two versions of JavaScript
only differ in values for certain variables but have identical overall functionality, they should produce similar signatures. Finally, the signature should change for functional changes in the script which is the easiest property to satisfy since if two versions of the JavaScript have different functionality, they are essentially different scripts and hence will produce different signature under any scheme of choice.

Three intuitive choices of signature schemes were considered for this project. Firstly, one possibility is to just compare the raw JavaScript text from two samples. This could be used as a measure and can also detect major changes in the script but is not stable under small changes in the script. One way to compare the two texts is to compute the digest of the script using some secure hash function and use it as a signature. This would again satisfy the property that major changes within the script are effectively detected. However, the digest cannot be used as a measure since the difference between two digests is not a function of the difference between the structures of the underlying codes. This is important because we ultimately want to cluster together the scripts functionally closest to each other and not just figure out which scripts are identical and which are not. The digest is also not stable under small changes within a script due to the second pre-image resistance property of hash functions. Therefore, a good signature would be the one that leverages the AST of the JavaScript code. An AST based signature can be used as a measure of the script as it records the detailed structure of the code. The signature would be stable under small changes in the code since the AST not only records the differences between two different code samples but also the exact type of differences. For the same reason, any AST based signature will also allow detection of major changes between two different scripts.

4.2 Using AST to Generate Signature

The AST is a representation of the abstract syntactic structure of a source code [10]. Each node of an AST represents some construct occurring in the source code. The AST stores
things like the name of variables within a node. This does not affect the overall structure of the tree if naming is altered between two different versions of the code. AST also does not contain any non-essential punctuation and delimiters which again makes it suitable for analyzing different versions of the same JavaScript code. However, it is still a challenge to come up with a way to construct a signature using the AST representation of JavaScript. There are two prior works that do use ASTs to derive a signature for JavaScript. SCILIALN uses set fingerprinting and Merkle Hash Tree (Merkle, 1989) based techniques to represent each AST with a unique hash value. This approach is insufficient because just like calculating the digest of the code, the hash value of the AST cannot be used to detect how similar two ASTs are to each other. Revolver on the other hand makes use of a novel technique in which the number of each type of node within each AST is calculated and used to a construct a n-dimensional vector where n represents the total number of different types of nodes within the AST. In this project, we use variations of Revolver’s approach to construct signatures over the JavaScript samples.

### 4.3 Constructing a signature

Once the AST is constructed, it can be used to calculate a signature over a web page. We used three different approaches to construct this signature. Figure 3 shows the sample tree used to illustrate the signatures produced by each of these different signature schemes.

![Sample Tree](image)

Figure 4.1: Sample Tree

The first approach is identical to Revolver where, the AST is used to construct a feature vector S:
\[ S = [c_1, c_2, \ldots, c_n] \]

where each \( c_i \) represents the count of the \( i \)-th type of node in the AST. This vectors then acts as the signature of the JavaScript sample. For the sample tree, the feature vector will be:

\[ S = [1,1,2,1] \]

The second approach is different in that instead of simply counting the number of each type of node, we also incorporate the depth of the node in the AST. Therefore now the feature vector \( S \) is:

\[ S = [w_1, w_2, \ldots, w_n] \quad , \quad w_i = \frac{1}{\sum_j d_j} \]

where each \( d_j \) represents the depth of the \( j \)-th occurrence of the node type \( i \) in the AST. This is based on the idea that deeper the node in the AST less ‘significant’ is the node. This is because the removal or addition of a node higher up in the tree has more of an effect on the overall structure on the tree and therefore on the functionality of the code. For the sample tree, the feature vector will be:

\[ S = [1,0.5,0.83,0.33] \]

Compared to the previous method in this one, D node has lesser impact on the feature vector than the B node. Also both the C nodes have less of a combined impact than they did previously, where each node counts equally, because one of the nodes is deeper in the tree than the other.

Finally, the last approach uses the size of the sub-tree rooted at each node as a weight. In this case the feature vector \( S \) looks like:

\[ S = [h_1, h_2, \ldots, h_n] \quad , \quad h_i = \sum_j t_j \]
where each \( t_j \) = number of nodes in the sub-tree rooted at \( j \)-th instance of node type \( i \) in the AST. This is based on the intuition that the bigger the sub-tree rooted at the node, the more influence that node has on the functionality of the code. Just looking at the depth is not a true measure of significance of the node because, for example, a node representing a literal higher up in the tree is less significant than a node representing a complex control flow construct lower down in the tree. For the sample tree, the feature vector will be:

\[
S = [5, 3, 2, 1]
\]

Compared to the previous two methods in this one, the C node has less of an impact. This is because there are no subtrees rooted at either of the C nodes. Whereas the B node has a subtree with two nodes. This is better representative of the structure of the tree because removing B node has more impact on the structure than the removal of either of the C nodes. Similarly as compared to the first approach, here the A node has the highest value because its removal has a bigger on impact on the tree than the removal of D, B or C.

4.4 Implementation

There are many ways to construct and represent the AST of a piece of JavaScript code. In this project, GumTreeDiff (GTD) (J. u. a., 2014) was used to generate the AST. GTD is an open source code differencing tool. It uses a parser to convert a JavaScript code into its AST representation which can then be used to calculate the code’s signature. The program to do so was written in Python. For each JavaScript sample whose signature is to be calculated, the program fetches the raw JavaScript from the LevelDB instance using the scripts digest. The raw JavaScript is passed through a beautifier function which formats the code. This formatted code is then used to initialize a Script object which is given as input to GTD. The AST which is outputted by GTD represents each different type of node using a unique value.
All of the signature schemes discussed above were implemented in Python using standard Python libraries for arithmetic and vector manipulation. This not implemented independently but rather as a second part of a pipeline, the first part of which is the AST construction discussed above. The AST constructed by GTD is fed into this part of the process as a JSON data structure. This is then parsed and the relevant information is extracted and distilled into a feature vector. An important point to note here is that each AST is a representation a single script and almost all web pages on the web are collection of many scripts. Therefore, the output of this pipeline is partial feature vector of a single script and the final feature vector is obtained by adding together all the partial feature vectors.

4.5 Example

Listed below are a few examples of simple JavaScript programs. Figure 4 shows the corresponding vectors of node types in their respective ASTs. The first script has a single variable which stores a number, the second variable has a single variable that stores a string and the last one has two variables, one storing a number and one storing a string.

```
1 var myNumber = 10;
```

Listing 4.1: Number

```
1 var myHeading = "Hello World";
```

Listing 4.2: String

```
1 var myHeading = "Hello World";
2 var myNumber = 10
```

Listing 4.3: Number and String

We then constructed ASTs for each of these scripts. Figure 3 shows the list of node types in each of those ASTs. As can be seen in the figure the first two scripts have largely the same types of nodes except for the last one: the first scrip has "number" and the second script has
"name". This is evident from the scripts themselves which are the same except for what the variable stores. This also shows that just changing the name of the variable doesn’t effect the structure of the AST because the first and second script have different variable names. Furthermore the AST of the third script has a larger number of nodes owing to there being two distinct variables in the script.

Looking at the above example it is clear that ASTs do in fact show the properties posited above which make them a better fit for JavaScript comparison then script digests and raw texts. The AST is not effected by changing the variable names and is effected by changing the functionality of the script.

Next we calculated the feature vectors for our example scripts. As stated above, three different methods were used to calculate the feature vectors.

The first method simply counts the number of each type of node. The second method weights the nodes according to how deep they are in the AST. Figure 5 and figure 6 show the results of the first and second approaches respectively. As it clear under these methods all three scripts produce different feature vectors. There isn’t really much of a difference between two approaches at least from the simple examples that we have used.

The third approach looks at the size of the sub tree rooted at each node. The results of this are displayed in figure 7. Here there is a clear difference from the first two approaches. In this case, the value corresponding to the SCRIPT node in the feature vector is not same
for all three pieces of code. This is because the AST of the last script would be bigger than the first two since it has more lines of code.

```
1 var myHeading = hello;
2 hello = 10
```

Listing 4.4: Nested assignment

The node list for this script is shown in figure 8. In this case there are a few extra node types such ASSIGN and EXPR_RESULT. The node ASSIGN is for the assignment of value 10 to the variable hello and the EXPR_RESULT node encapsulates this entire assignment expression.

```
['SCRIPT', 'VAR', 'EXPR_RESULT', 'VAR', 'ASSIGN', 'NAME', 'NAME', 'NAME', 'NUMBER']
```

Figure 4.6: Node list

Next we constructed the feature vector for this new hybrid script using the method that looks at the size of the subtrees. As can be seen from this example, shown in figure 9, the SCRIPT node has the same value because the number of lines of code are the same. However, now there are more nodes also present in the feature vector. These nodes have values that are higher than the NAME and NUMBER, showing that they have more functional importance.
than those nodes. This is true because the node NUMBER simply encodes the type of the
value being assigned to the variable whereas something like EXPR_RESULT would store the
actual expression which is more indicative of the code’s functionality.

\[
\begin{array}{cccccc}
NAME & NUMBER & ASSIGN & VAR & EXPR\_RESULT & SCRIPT \\
[3 & 1 & 7 & 7 & 8 & 9]
\end{array}
\]

Figure 4.7: Feature vector for hybrid script
Once the signatures over JavaScript samples have been calculated, the next problem is to train a classifier over these signatures which is able to group together pages with similar signatures into what we call Page Templates and is able to correctly assign a newly discovered web page to one of these templates.

Conceptually, a page template is a the skeleton JavaScript code that is used by a web server to generate the set of functionally similar web pages. This makes use of the knowledge that most websites pages are often generated dynamically from a set of templates. Such a set of templates can than be used to build a log such that when a client encounters a new page it is able to verify that it is generated from one of the templates. The client can then proceed to perform more fine grained analysis of the ways that the new page differs from this basic template in order to determine if the said changes are malicious or benign. This section describes how such set of templates can be created from the signatures constructed using the method described in the previous section.

5.1 Clustering

The problem of learning page templates from a large collection of unclassified signatures is a case of unsupervised learning since the goal is to design a system which needs no manual intervention and without such manual input it is impossible to obtain a labelled set of signatures to act as training data. Clustering, therefore, is the most natural solution to the problem. It is the most well defined and understood unsupervised learning method and has a wide variety of flavors all suited for different types of data. For our purposes we used two types of clustering algorithms: Agglomerative clustering (Lior und Maimon, 2005) and DBSCAN clustering (Ester u.a., 1996). Both of these approaches are well-suited for
problems where the number of clusters cannot be known a priori.

5.1.1 Agglomerative Clustering

The agglomerative clustering algorithm starts by treating each sample as an individual cluster and iteratively merges the closest clusters to form new clusters. At each step, a quantitative measure of the quality of clusters is calculated. This is done by measuring the maximum inter-cluster distance \(d\) between clusters. The step which has the maximum \(d\) is considered to be the best set of clusters as it represents the set of clusters in which the individual clusters are more further apart than in any other set of clusters.

This project uses agglomerative clustering to group together web page signatures into clusters where each cluster represents a page template. As stated above each page is represented by a feature vector. The distance between these feature vectors is the euclidean distance between them in the n-dimensional Cartesian space, where \(n\) is the number of different types of nodes. We implemented a custom agglomerative clustering algorithm in Python using the best principles and practices common in the machine learning community. We did not use a library implementation because we wanted some extra information from each step of the algorithm to help in testing as will be explained in a later section.

5.1.2 DBSCAN Clustering

DBSCAN is short for Density-based spatial clustering of applications with noise. As the name suggests, the algorithm groups together points according how spatial density. Points that are close together are grouped in the same cluster whereas points which are further part are marked as outliers. The algorithm takes as input the \(\epsilon\) radius of the neighbourhood of a point to be scanned and the minimum number of points that must be in that region to form a dense region. The algorithm works by scanning the \(\epsilon\) region around a point and if contains more than the minimum number of points in that region a new cluster is formed, otherwise
the point is marked as an outlier. The points marked as outlier may become part of other clusters if they fall in the $\epsilon$ region of another point.

Once again the DBSCAN is used to form page templates by grouping together page signatures. The inputs are once again the feature vectors of the web pages and the distance, the euclidean distance between those vectors. In this case we used an implementation of DBSCAN clustering from the Python library pyclustering (pyclust).

5.1.3 URL Based Clustering

To compliment the AST based clustering, we also decided to cluster the web pages based on the URLs of the web pages. Here we tokenize the URL by partitioning it along characters like `/`, `.`, `:` and `?`. We then cluster the URL using agglomerative clustering where each URL is represented by a vector of its tokens. The distance between two such vectors is the percentage of tokens that are identical across the two URLs. A cluster that contains more than one URL is represented by the vector of largest sequence of common tokens. The places where the tokens differ are instead represented by placeholders. Similar to the agglomerative clustering for AST based signatures, all the URLs start off as separate clusters and then are combined together until there is only one cluster. On each step, the algorithm records the distance between clusters and the step where this is maximized is deemed to be the optimal clustering. Clustering using URL matching was performed to verify whether it is possible to build a system where the page templates can be defined using some regular expression representing the URLs. This would have been simple and effective and would have required less processing than analyzing the entire code file.

5.2 Evaluating the Clusters

To validate the quality of clustering, a set of clusters is chosen at random from the final clustering. The set of clusters are then manually analyzed for correctness. This is done by
looking at the URLs in the cluster and determining whether they are suitably similar to each or in other words belong to the same page template. Unfortunately this is a very coarse and inexact method since there is no effective method of acquiring a labelled training set automatically. This is because, having a web server that labels the pages itself contradicts the fundamental assumption of the project that the servers are not trusted. The results of manual testing are described in more detail in the later section.

5.3 Least Common Structure

Given a cluster with a set of web pages, the next step is to construct a representation for that cluster. This representation is essentially what will be a page template. Hence the purpose of the representation is to be able to assign new samples to the appropriate page templates and also to compare different page templates with each other.

Since the goal of this representation is to act as the page template, the nature of the representation has to be such that it encodes all the similar features of the web pages that it contains while leaving holes where the differences can occur. To this end, one solution is to come up with a sort of a Least Common AST Structure (LCS) for the entire cluster corresponding to each template, where the common AST contains all the nodes that are common among the individual ASTs within the cluster. In addition to that the LCS also has holes or empty nodes where the nodes may vary in different pages belonging to that particular template. To find the LCS of two ASTs, pre-order traversal is performed for each tree giving two sequence of nodes which essentially encode the structures of the respective tree. The list is then compared node by node, keeping the common nodes in place and putting holes in places where the nodes are different in the two trees. An example of how such a tree would look like is presented in figure 10.

Furthermore, as stated above each web page contains multiple scripts and therefore is represented by a structure that contains multiple ASTs, each for a script in the web page.
Therefore to construct the LCS for a template, the scripts have to be matched up and then the least common AST of each of the matched scripts needs to be calculated. As a result, as shown in figure 11, the final LCS is a list of least common ASTs where each least common AST is constructed using scripts which are common across each web page.

![Diagram](image)

Figure 5.1: Example of what an AST in the LCS might look like

Figure 5.2: High level structure of LCS of Page Template
CHAPTER 6
EVALUATION

To determine the accuracy of our classifiers we carried out a series of experiments. We also instrumented our code to measure the performance of both the crawler and the classifier. We evaluated each of the component of our system described above, both independently and as a pipeline. We also collected significant amount of raw unprocessed data from the crawls. The following sections summarize the results of the crawl, the clustering and the LCS construction. The results are followed by an analysis which attempts to explain why the system is more challenging than first anticipated and how successful each of the different attempted solutions were in tackling both the larger problem and each of the smaller sub-problems.

6.1 Ground Truth

In order to verify the correctness of our signature and clustering algorithms, we tested them on a custom WordPress website. WordPress is a free and open-source content management system based on PHP MySQL. It allows users to create and host a website with web pages generated from one or more templates that the service provides.

For the purposes of our tests, we created a WordPress website that had 10 templates and 3 web pages per template. We then used our crawler to collect the JavaScript from the aforementioned website. These JavaScript samples were then run through the classifier which clustered them according to their respective ASTs.

The results show that the clusters accurately identify the similarity of the underlying JavaScript code.
6.2 Results

6.2.1 Crawling

First we evaluated our crawler. We are not particularly concerned with the performance of our crawler because it is not part of the system we are proposing. Instead we are more interested in analyzing the results of the crawl. We look how many websites we were able to crawl successfully and where we were unsuccessful. This allows us to gauge where and how to focus our classification efforts. If there were too many failures for a particular website we decided to either terminate the crawl for that website or decided to not run the classification algorithm for that website. Because we cannot be sure whether all the JavaScript has been correctly collected and incomplete scripts may negatively impact the output of the classification algorithms.

To test our system we deployed our crawler on an instance of the Amazon AWS server. The server was configured to run twice during each 24 hour period. During each run a controller script, written in Python, is triggered. This controller script reads the list of websites, calls the download method with each of the websites. As mentioned above, there is rate limiting in form of both a small pause in between loading of subsequent loads and not re-trying more than three times in case of failed loads. The results of the crawl are the extracted links stored in csv files, the script metadata stored in json files and the raw scripts collected and stored in the LevelDB instance.

In total 314350 total unique links were discovered during the crawl. Table 6.1 summarizes the number of unique link discovered during the crawl for a subset of all the sites crawled. Some of the sites, specifically those for which the number of links discovered were less than a 1000 have not been included in the reported results. For example, Expedia had several timeout issues which lead us to terminate it from our crawl. Airbnb and Eventbrite on the other hand required 2 factor authentication when a user logs in from a different
location, which was a common problem since the Amazon EC2 instance runs from different geographical locations. eBay and Spotify on the other hand shut down the accounts due to high volume of activity.

<table>
<thead>
<tr>
<th>Website</th>
<th>Unique links</th>
</tr>
</thead>
<tbody>
<tr>
<td>9gag</td>
<td>23192</td>
</tr>
<tr>
<td>buzzfeed</td>
<td>1051</td>
</tr>
<tr>
<td>etsy</td>
<td>26580</td>
</tr>
<tr>
<td>eventbrite</td>
<td>1450</td>
</tr>
<tr>
<td>facebook</td>
<td>8762</td>
</tr>
<tr>
<td>gyfcat</td>
<td>2447</td>
</tr>
<tr>
<td>imgur</td>
<td>3004</td>
</tr>
<tr>
<td>instagram</td>
<td>19524</td>
</tr>
<tr>
<td>kickstarter</td>
<td>18847</td>
</tr>
<tr>
<td>linkedin</td>
<td>2994</td>
</tr>
<tr>
<td>msn</td>
<td>3147</td>
</tr>
<tr>
<td>quora</td>
<td>71768</td>
</tr>
<tr>
<td>reddit</td>
<td>22391</td>
</tr>
<tr>
<td>stackoverflow</td>
<td>6276</td>
</tr>
<tr>
<td>tumblr</td>
<td>1890</td>
</tr>
<tr>
<td>vimeo</td>
<td>38293</td>
</tr>
<tr>
<td>wikihow</td>
<td>30734</td>
</tr>
<tr>
<td>wikipedia</td>
<td>7750</td>
</tr>
<tr>
<td>wordpess</td>
<td>1999</td>
</tr>
<tr>
<td>yahoo</td>
<td>2626</td>
</tr>
<tr>
<td>zillow</td>
<td>2674</td>
</tr>
</tbody>
</table>

Table 6.1: Unique links discovered

Also from the table, the websites with the highest number of discovered links are the ones with a large number of unique links to other pages on each page. This are links to other questions (quora), videos (vimeo), topics (wikihow), products (etsy), pictures (instagram) and other user accounts.

The other websites, which had fewer links, either have more timeouts or in some cases have a lot of repeat links.
6.2.2 Clustering

As stated in a previous section, clustering was performed using two different types of clustering algorithms and three different types of feature vector construction techniques. In this section a subset of results of all of these parameters are reported. We chose to present the results that are representative of the overall results. Finally, since we created user accounts, the results are reported for scripts extracted using both user accounts and without logging in to the accounts.

For the first set of experiments, the feature vectors were constructed using the method outlined in the Revolver paper i.e. by counting the number of each type of node in the AST. We did this for each of the site in the data set, followed by performing the clustering using one run of the Agglomerative algorithm and one run of the DBSCAN algorithm. The results are presented in the graph in Figure 6.1.

Then, as stated above, we altered the feature vector construction approach to also take into account the depth of each node within the AST. We then performed clustering using the feature vectors constructed using these new feature vectors and both the agglomerative and DBSCAN algorithms. The results are recorded in graph in Figure 6.2.

Next, clustering was performed using the feature vectors constructed using the size of the sub-tree rooted at each node. Again clustering was performed using both agglomerative clustering and DBSCAN clustering. The results are presented in graph in Figure 6.3.

As it is clear from the graph, in most cases, the clustering is not able to produce a suitably small number of page templates. The reasons for this are analyzed in the next section. Furthermore, as stated above, for a more rigorous test of the quality of clusters, a random sample of clusters are inspected manually. The results of this analysis showed that the while some clusters made intuitive sense, in that they had pages which belonged to the same basic template, there were other clusters which had pages which were quite different from each other. This method was also used to test the quality of clusters formed
Figure 6.1: Results of Clustering using Node count with different types of feature vectors. The appendices further elaborate on the quality of clustering by listing some example clusters, with a small description about each explaining what they say about the clustering algorithms and feature vector choices.

Also, since we collected the data over a long period of time, we were also able to perform temporal analysis. In particular we looked at how the set of page templates changed over time and whether we were getting what we called 'stable cluster', i.e. if two pages $p_1$ and $p_2$ are clustered together at time $t_1$ then are they also in the same cluster at time $t_2$. The results are summarized in the graph in Figure 6.4. As it can be seen from the graph, the number of clusters change as time progresses. This is not in and of itself not an unsurprising result because as crawling proceeds, more pages are discovered and added to the database.
However, more interestingly, the content of the clusters also changes with time. For Facebook, some URLs do not appear in the same clusters at time $t_0$ and $t_1$. On the other hand, for wikihow most URLs do appear in the same clusters at time $t_0$ and $t_1$. And then there are sites like Quora, where the pages being are clustered are too similar too each and very small differences result in pages ending up in different clusters at different points in time.

Finally, we report the results of our URL based clustering approach. Although the results were encouraging at first, unfortunately, as the algorithm was tested on larger samples and across different websites, several problems emerged. As a lot of websites have matching URLs for disparate web pages which makes this method unsuitable for a scalable log in the real world. For example Facebook has the same format of facebook.com/userid/itemid for
Figure 6.3: Results of Clustering using Sub-Tree size

both photos and status updates which are completely different types of web pages in terms of JavaScript. Another example is that of buzzfeed in which almost all article pages have the same format, buzzfeed.com/author/articletitle but some of the pages are a gallery of images, some are ‘listicles’, some are just text-based articles and some are interactive quizzes and unsurprisingly all of them have different JavaScript templates. This is illustrated in the appendix at the end of the paper.
Figure 6.4: Results of Tracking changes in clusters over time
CHAPTER 7
ANALYSIS

The overarching goal of the project was to measure the nature of JavaScript online to determine whether it is possible to build a system that allows clients to verify client side JavaScript code without the need of any implicitly trusting the server or requiring any manual action from the client themselves.

7.1 Crawling

The results of the crawl show that there are several challenges in conducting a large scale deep crawl of the web. Firstly, the scripts on an individual page do not load in a consistent manner each time, which results in gaps in the data. Secondly, when crawling while logged in as different user accounts, we have to deal with several security measures like two-factor authentication, rate limiting etc. Furthermore, a lot of JavaScript is loaded dynamically in response to how the user interacts with the page. The crawl results of Google also highlight the issue of sub-domains existing within a number of web sites. As is the case for Google, these sub-domains might all have a completely different set of basic templates from which they are generating their pages.

7.2 Clustering

Clustering was performed with the objective of grouping together functionally identical JavaScript to learn page templates which can then be used by a classifier to verify newly served JavaScript. However, the results stated above show that clustering did not prove encouraging as a solution to this problem. This was largely due to the fact that all of the data was unlabelled which ruled out any supervised learning methods. Furthermore it made it practically impossible to test the results of the unsupervised clustering that was used.
Testing almost entirely relied in manually sampling and inspecting subset of clusters which kind of creates a circular problem. The entire point of AST based clustering was to find non-obvious, non-intuitive differences in otherwise similar web pages. But if this clustering has to be analyzed manually then we are re-introducing the imprecision back into the training process.

Furthermore, even with some hypothetical perfectly accurate unsupervised learning algorithm, the JavaScript samples are not suitable for such an analysis. This is because the JavaScript code in the wild is very unstructured due to practices like obfuscation and minification which makes it unsuitable for unsupervised learning. Obfuscation is the technique of transforming the code into completely unreadable form while maintaining the original functionality. This breaks our pipeline because it is no longer possible to construct the AST from the obfuscated JavaScript (Obfuscation). Minification is the process of removing features from the code until what remains is the bare minimum needed to not alter the functionality of the code (minification). These characteristics of JavaScript reduce the features available for clustering algorithms and therefore the results are not as accurate as they can be. Furthermore, each web page is made up of a large number of scripts and the temperamental nature of the internet means that a lot of times, all the scripts do not load. And it is not possible to determine whether the page has loaded completely or not without having a list of what scripts should load each time the page is visited. A large number of scripts are also generated dynamically which means that even if the same page is loaded on separate occasions, the code has many differences, which throws off the clustering algorithm. This is why when the same list of pages are crawled and clustered at a different time the number of clusters are different and the same pages are not always clustered together. This also points to a larger problem because if page $p$ being loaded at a subsequent time $t_1$ has sufficiently different code as compared to its code at $t_0$ than a theoretical transparency log will fail to correctly classify $p$. 
Next, we perform a deeper analysis of the results in the context of the three strategies that we employed in calculating the feature vectors from the AST. As stated above, we used three different methods of constructing feature vectors in our study. We evolved our method as our study progressed and the results came in. In the following analysis, we repeatedly refer to the structural importance of a node in a tree. Here structural importance of a node refers to the effect that removing that node will have on the overall structure of the tree. More precisely, how many other nodes will be removed from the tree when the node in question is removed. Or alternatively, if we make a change to that node, to how many other nodes will the effects of that change propagate down to.

The first method of constructing feature vectors was based on counting the number of nodes in the AST. Just looking at the numbers from the tables above suggests that the feature vectors based on the depth of node produce the least amount of clusters. Counting the number of nodes is the least accurate because of two main reasons. Firstly, it is possible for two pages to be functionally the same but have different data values. This would result in the ASTs of the respective pages to have a different number of data nodes. Furthermore, just counting the number of nodes does not take into account the position of the nodes in the AST. Two completely different pages may have the same number of nodes but the way those nodes are arranged in the tree might be different and consequently, the structural importance of each node might be different.

Therefore, to further take into account the position of the node in the AST, we decided to augment the node count with the depth of the node in the tree. Specifically, the deeper the node in the tree, the less it counts towards the final feature vector. However, this approach does not work all that better than just counting the number of nodes. The main reason is that just because two nodes are at the same depth in the tree, doesn't mean that they have the same structural importance to the AST. The neighborhood of the nodes, i.e. the other nodes that the nodes in question are connected to, is also important. Firstly, it is possible
that one of the nodes has more sibling nodes. In this case, having more sibling nodes doesn't necessarily reflect whether the node in question has more structural importance in its AST.

Secondly, each node has a subtree rooted at that particular node. This sub-tree could be a tree with a single node, a tree with more than one node and a tree with zero nodes which would be the case if the original node was a leaf node in the AST. In this case, the larger the sub-tree the more structurally important it is to its respective AST.

Wherefore, to capture the complete picture we decided to also take into account the size of the sub-tree rooted at each node. The results show that this approach while being better than the previous two is still not perfect. The reason that it works better than the previous two is that, as explained above, the larger the subtree the more structurally important is the node because removing the node will cause all the nodes in that subtree to be removed. While this method does yield better results than the previous two methods, it still has some issues. Firstly the size of the subtree does not provide any information about the nature of the subtree. The nature of the nodes here means the types of nodes that make up the subtree. Just because two subtrees have the same number of nodes doesn't mean that those nodes have the same significance to the overall AST. For example, the subtree of one of the nodes may primarily have data nodes and the subtree of the other node, while being of the same size might have all control flow nodes.

Finally, comparing the two clustering algorithms, we observed certain performance and accuracy differences. Overall, the library implementation of DBSCAN was much quicker than our custom agglomerative algorithm. While DBSCAN would execute in the order of milliseconds, the agglomerative algorithm would take order of minutes.

In terms of accuracy, the DBSCAN algorithm is less accurate than the agglomerative algorithm. This is because DBSCAN needs an initial parameter, namely the $\epsilon$ neighbourhood for each point. This parameter determines to a certain extent how the points get clustered. In particular, the fact that it is hard to exactly pin down the structure and scale of the data,
makes choosing an appropriate $\epsilon$ for the algorithm. A small $\epsilon$ will result in a lot of points being marked as outliers and if the value is too large a significant part of the data will not be clustered. These issues do not apply to the agglomerative clustering algorithm because it simply functions in a greedy manner, merging the closest clusters at each step. Therefore there is no apparent glaring disadvantage of using the agglomerative method other than the high time complexity. The lack of accuracy in the result of the agglomerative algorithm therefore is likely due to the unpredictable nature of JavaScript being served by the web sites.
CHAPTER 8
CONCLUSION AND FUTURE WORK

The goal of the project was to study the feasibility of building an infrastructure necessary for implementing web transparency in a fashion similar to the Certificate Authority transparency log. However, the results show that such a system may not be entirely feasible to construct using the methods we tried out in this paper. Whereas CA Certificates remain fairly constant over time, the client-side JavaScript code on the web is very unwieldy and dynamic which makes it difficult to make a static log such that all future instances of a script can be compared to the entries in the log. Furthermore, our study also shows that building a system without somehow also including the server in the loop is not feasible. This is because, without the server annotating its code, testing the classification methods is extremely challenging.

Moving forward, due to the difficulty of classification and the unavailability of labeled training data, we propose non-machine learning approaches. One possible method of testing whether two JavaScript samples are functionally similar is to run some canonical tests for both the scripts and check if they behave identically under those tests. We also propose developing methods which include some human input within the process. Manual input can be used to label training data, rate the accuracy of intermediate results, and tweaking the parameters of algorithms like DBSCAN. Furthermore the systems and techniques developed in this paper can be extend to applications beyond just code verification. Page templates can be used to change how the Same Origin Policy is defined; a webpage can be restricted to accessing resources only loaded from the context of the larger template from which the page is generated. The Least Common Structure, as described above, can be used to conduct a study of how the code evolves in the wild as new versions are introduced and libraries are updated. Finally, the approach outlined in this paper can be extended to the entire page rather than just the JavaScript code which can then be used by the site operators to identify elements within the page that are necessary to defining the content security policy for a their
particular website.
REFERENCES


[google ] : Certificate Transparency. – URL
    https://www.certificate-transparency.org/


[minification ] : Code minification. – URL


[Obfuscation ] : Javascript Obfuscator. – URL
    https://javascriptobfuscator.com/faqs.aspx


[w3c ] : Subresource integrity. – URL https://www.w3.org/TR/SRI/


[D u. a.]  D, Zhang ; D, Gillmor ; D, HE ; B, Sarikaya: CT for binary code. – URL


Appendix A

DBSCAN v AGGLOMERATIVE

➢ The following two clusters are from an instance of the DBSCAN algorithm run for Facebook.

[ ]
- https://www.facebook.com/pankratov?lst=100010693126221%3A100010693126221%3A1501191523
- https://www.facebook.com/groups/pkplaceseditors
- https://www.facebook.com/brian.patterson.14289
- https://www.facebook.com/baltazar.cuatecontzi.35/place_reviews_written

As can be seen the pages belong to several different templates. First and third are trending topics pages, second is an activity log, which is followed by groups and profile pages. This cluster therefore is not suitable for the purposes of learning a page template.

[ ]
- https://www.facebook.com/UChicagoMaya
- https://www.facebook.com/uic.edu/?nr
- https://www.facebook.com/ChicagoPrintmakers/?nr
- https://www.facebook.com/99029450818
- https://www.facebook.com/csudosa/?nr
- https://www.facebook.com/alonearestaurant/?nr
- https://www.facebook.com/alonearestaurant/?ref=page_internal

The next cluster is an example from the same run of the algorithm as above. In this case, all the URLs point to Facebook ‘pages’ which are non-personal Facebook profiles for celebrities, bussinesses etc. Therefore we have two clusters one which has a homogenous group of web
pages and one which has a very diverse range of web pages. The above cluster also illustrates the shortcomings of using a URL based clustering scheme since different types of URL strings point to the same type of web page.

➢ The next cluster is for Stackoverflow from an instance of the Agglomerative algorithm and an instance of DBSCAN respectively.

[● https://stackoverflow.com/questions/tagged/jquery?sort=newest&pageSize=50
● https://stackoverflow.com/questions/tagged/jquery?sort=active&pageSize=50
● https://stackoverflow.com/questions/tagged/css-shapes
● https://stackoverflow.com/questions/3087975/how-can-i-make-the-cursor-a-hand-when-a-user-hovers-over-a-list-item
● https://stackoverflow.blog/2017/07/17/podcast-112-please-direct-hate-mail-jay-hanlon-%e2%84%85-stack-overflow
]

[● https://stackoverflow.com/help
● https://stackoverflow.com/unanswered
● https://stackoverflow.com/company/press
● https://stackoverflow.com/company/salary
]

The first cluster above is an example of an inaccurate cluster produced by the agglomerative algorithm and the second cluster is an example of an inaccurate cluster from the DBSCAN algorithm. As it is evident the output of Agglomerative is much more homogenous, whereas DBSCAN output is much more scattered with all pages being different from each other. This also further illustrates the futility of trying to use URLs for clustering and classification because company/press and company/salary are completely different page templates.
The following two examples (one in blue and one in green) are of three different partial clusters each from the same instance of the clustering algorithm where the feature vectors were constructed by counting the number of node types. Agglomerative clustering was used in both cases. These examples show how similar pages are clustered into different clusters. In the first case (blue) the URLs point to pages with comments on different posts. The second example (green) has three URLs linking to photos from the same album.

[ ... ]
  ● https://www.facebook.com/uchicago/posts/10158452383356650?comment_tracking=%7B%22tn%22%3A%22O%22%7D

[ ... ]
  ● https://www.facebook.com/uchicago/posts/10158418445240660?comment_id=10158421345356560&comment_tracking=%7B%22tn%22%3A%22R0%22%7D

[ ... ]
  ● https://www.facebook.com/uchicago/posts/10158425267155660?comment_id=10158437759040660&comment_tracking=%7B%22tn%22%3A%22R%22%7D

[ ... ]
  ● https://www.facebook.com/uchicagomag/photos/a.10154925511036839.1073741842.26073421838/1015492551246839/?type=3

[ ... ]
  ● https://www.facebook.com/uchicagomag/photos/a.10154925511036839.1073741842.26073421838/1015492551256839/?type=3

[ ... ]
  ● https://www.facebook.com/uchicagomag/photos/a.10154925511036839.1073741842.26073421838/10154925511491839/?type=3
The next example is from an instance where the feature vectors were made using sub-tree size. The algorithm is Agglomerative again. Here the pages with photos are in the same cluster. Note that this is a partial cluster. There are other pages in this cluster too so this is not perfectly accurate but more accurate than just counting nodes.

[...
...]

The next cluster is from an instance where the feature vector were made using node depth. The algorithm is Agglomerative. Here three very distinct types of pages are in the same cluster. As a result this produces a very small set of clusters.

[...
- https://www.kickstarter.com/discover/categories/art/mixed%20media?ref=discovery_overlay&sort=most_funded
- https://www.kickstarter.com/settings/account
...]}