Hunting for malicious infrastructure using big data

Shadi Al-Hakimi and Freek Bax
University of Amsterdam

In this paper we present two new machine learning features extraction methods which can be used to hunt for anomalies in HTTP responses to find malicious infrastructure. We also discuss the advantages and disadvantages of different machine learning methods like supervised and unsupervised learning. We proposes a uniqueness feature which indicates how unique the combination of HTTP headers of a specific HTTP response is in comparison to the whole data set. And a feature which captures the order of the most common headers. The evaluation shows that these feature extraction methods could be included when using machine learning to find command and control (C&C) servers inside of a data set of HTTP responses.

Introduction

The HTTP protocol is one of the most used protocols on the internet today. This is one of the reasons a large amount of malicious actors are using HTTP to hide their communications [35]. As HTTP traffic is usually not blocked by firewalls and finding the malicious communication in the large amount of HTTP traffic is a hard task. C&C servers of both Botnets and Post-exploitation tools are known to use it for their communication. An infected Host will send a HTTP GET request to the C&C server and receive a response from the server with the actions it should carry out [29]. Identifying these malicious C&C servers can be valuable for blue teams to better protect their networks against attackers.

One example of to identify Cobalt Strike C&C servers is the fingerprinting technique published by Fox-IT [14]. They show an anomaly, an extra white space at the end of the status line, used to fingerprint C&C server of this post exploitation tool. With this, they were able to identify all the webservers that used NanoHTTPD, the open source webserver that caused the anomaly, including Cobalt Strike C&C servers. This anomaly was fixed in Cobalt Strike version 3.13, so for actors using the latest version of the software, this anomaly can no longer be used to fingerprint their communication. But it does indicate that anomalies found in HTTP responses can be used to find red team or malicious actors C&C servers.

Most research on anomaly detection focuses on outgoing HTTP requests [37], [16], [10] or analyses all the incoming and outgoing traffic of a specific host or application [5], [35], [23]. The advantage of these approaches is that more data is gathered for each connection and therefore more information is accessible to the anomaly detection algorithm. The disadvantage is that it requires communication between a beacon and C&C server to be active, which means the network is already breached.

In our research we follow a passive approach by only looking at HTTP responses for HTTP GET root (/) in public data sets, which from this point on we will refer to as *HTTP responses*. The advantage of this approach is that it can be done before a connection is made between a beacon and C&C server. Meaning before the malicious actor has breached the network. Using only one HTTP request of the root to scan a webserver is also less intrusive than techniques which include other requests to gather more data. Another advantage of using a public data set of HTTP responses is that one party could scan the internet, identify the C&C servers and publish a blacklist containing the IP's or domains of these servers.

In this paper we will investigate feature extraction methods that can be used in machine learning algorithms to analyse large data sets of HTTP responses to identify anomalies and hunt down these malicious C&C servers. This will consist of an overview of related work in the field of anomaly detection on HTTP traffic. A theoretical analysis of possibilities and limitations of different machine learning techniques. An empirical analysis of two new different feature extraction methods for HTTP response headers.

Problem statement

Finding traces of communication generated by malware or other malicious activity inside large amounts of network traffic is referred to as anomaly detection. The definition of an anomaly has been given in several different papers. Barnett and Lewis [2] define an anomaly as an "observation which appears to be inconsistent with the remainder of the set of data". Lakhina et al. [17] define anomalies as "unusual and signification changes in a network's traffic levels, which can often span multiple links." Hoque et al.[12] define anomalies as "non-conforming interesting

patterns compared to the well-defined notion of normal behavior". From these definition we can see that finding a definition of "normal" data within the data set is the first step of anomaly detection. From this definition of "normal" we can identify data points which appear to be inconsistent with "normal" and can be labeled as an anomaly.

There are a few different techniques for doing anomaly detection. They all share the concept of creating a numerical representation of the data which can be used in an algorithm or other mathematical function. Statistical methods and methods based on information theory try to look at the mathematical properties of the data, like entropy or correlation [1]. While machine learning methods like classification or clustering use algorithms which receive the data as an input. From this input, it learns their parameters to group or classify the data set. [9].

We are going to give an overview of the possibilities of machine learning on a data set containing HTTP responses. Our research question is;

- How can machine learning be used on a data set of HTTP responses to identify malicious webservers?

For this we will look at two problems which are needed to be solved when using machine learning.

1. Which machine learning algorithms are best suited for anomaly detection on a data set of HTTP responses?
2. What features can we extract from the HTTP responses?

The first sub-question will not be answered with experiments, but we will give an overview of which machine learning algorithms can be used in the background section. The second sub-question will be the main question of our experiments. Because we have not found any relevant research looking into feature extraction for HTTP responses and feature extraction is an important problem to be tackled before machine learning can be used on a data set of HTTP responses. We will propose two new feature extraction methods and analyse if they are suited for anomaly detection with machine learning algorithms. We think that these two sub-questions together will answer how machine learning could be used to identify malicious webservers inside a data set of HTTP responses and we will provide recommendation for future research.

Earlier Work

Since we are going to look for anomalies in a large data set of HTTP communication, our research relates to the field of HTTP anomaly detection. Our research also relates to HTTP fingerprinting, because we are trying to identify webservers on the basis of their HTTP response. In this section we will give an overview of earlier work done in both of these fields.

HTTP Anomaly detection

Research in HTTP anomaly detection focused on finding communication of botnets has been done with different methodologies. By looking for specific properties of the network traffic generated [10], [11], [31], using Clustering techniques [20], [7] and Supervised learning techniques [6], [32]. Other approaches like hidden Markov modeling [30], semi-supervised learning [28] and genetic algorithms have also been used [22].

Research on finding a more general definition of malware, instead of only botnets, has also been done. Zarras et al.[35] and Bortolameotti et al.[5] both proposed an anomaly detection method with two phases: a learning phase to learn what normal HTTP traffic on the network looks like and A detection phase where anomalies not previously seen on the network are identified as malicious. Another way of anomaly detection is to first look at the communication from a known C&C server and build up knowledge from this communication, this method was used by Nelms et al. [24], Perdisci et al. [26] and Rafique & Caballero [27].

K-means clustering was used in different studies trying to detect malware. N. Kheir [16] used K-means clustering on the headers of HTTP requests from user agents. Münz et al.[23] used K-means clustering on the time-intervals of network traffic. A more recent study uses recurrent neural networks to model HTTP traffic as natural language [34]. Another study used Gaussian Mixture Modelling to cluster anomalous data [4].

As we can see a lot of different approaches and algorithms have been used as anomaly detection for both botnet C&C server and other HTTP-based malware. The difference between all of these studies and our problem is the data used in the detection tools. Most research only looks at outgoing traffic or at all the communication inside the network it is trying to protect. However, we are going to look only at incoming traffic, the HTTP responses from webservers. As explained in our introduction, this has the disadvantage of having less data to work with, but the advantage of it being less intrusive and requiring no active scanning.

HTTP fingerprinting

The goal of fingerprinting in network traffic is to identify a user, agent or device. For HTTP fingerprinting, it can be used to identify web clients, such as browsers or webservers. Laperdrix et al. [18] give an overview of research into client fingerprinting. Our data consists of HTTP responses from webservers, which is a version of webserver fingerprinting.

webserver fingerprinting is used mostly by hackers or other offensive actors to identify the version of a targeted webserver. Knowing what webserver you are attacking is valuable information. HTTPrecon [13] and HTTPrint [15] are tools which send different types of requests to a targeted server, they look at the responses they receives from the server to identify them. In the academic world, webserver fingerprinting has not been a popular topic for research. However, there have been studies to help defend against webserver fingerprinting [19], [33].

These fingerprinting tools and studies do show some examples of what information can be used to identify webservers based on the HTTP responses. The main difference between the previous work and what our research will focus on is what to identify. We are not trying to identify a webserver version, but trying to find anomalies introduced by C&C servers that try to hide themselves by mimicking legitimate webservers [14]. Another difference between our research and these fingerprinting tools is how we identify webservers. They use a more active approach by sending a few different requests, a normal GET request, but also requests which will cause errors. Another difference is that our approach will focus on machine learning, while other fingerprinting tools use a logic based identification.

Background theory

Machine learning

Machine learning is a field in Artificial Intelligence which studies algorithms that learn through data or experience. There are two main approaches for algorithms learning from data, supervised learning and unsupervised learning. In supervised learning the data is labeled with the expected output of the data. The supervised learning algorithm will then try to learn the function that translates the input into the output, which it can then use to calculate the output of new data. If the output is a set of categories, like in our case where the data points should be divided into groups, this is called classification. Unsupervised learning, in contrast to supervised learning, does not need the output labels from the data it receives as input. It will take the input data without these labels and look for a structure in the data, like groups or clusters. If unsupervised learning is used to identify different groups inside the data this is usually referred to as clustering algorithms.

Classification. Examples of classification algorithms are logistic regression, support vector machines and neural networks. What they all have in common is that they work in phases. First, a training phase where the algorithms are fed the labeled data on which they train. After this the trained algorithms can be used to classify new data. Usually, to evaluate the performance of the classifier part of the data set is preserved as a test set. The algorithms will classify this test data set and the error rate can be calculated from this.

If we want to implement a classifier for HTTP responses, we will need a data set containing those responses and their classification. Such a data set does not exist and creating a data set with labeled data is not trivial. Labelling by a human is not possible, since we do not precisely know what an anomaly looks like. Some anomalies like the extra white space in the status-line found on old versions of a cobalt strike server are known. Creating a labeled data set only containing already known anomalies is possible. The drawback of this approach is that a classification algorithm trained on this data set will then only learn these already known anomalies. New anomalies will not be found, unless they have some other similarity with the HTTP responses of C&C servers found with the already known anomalies.

Clustering. Clustering algorithms are used to find groups of data points in a data set which are more similar or closer together than the data points in the other groups or clusters. They are used on unlabeled data and therefore do not have the problem of first creating a data set with labeled data like we saw with supervised learning. There are two popular algorithms that are based on a different clustering approach. K-means clustering is one of the most used clustering algorithms, it was first proposed by Macqueen [21]. It is an example of centroid-based clustering. It divides the data into k amount of clusters where each data point is assigned to the closest cluster centre. These cluster centres are then updated by computing the mean of all the data points in the cluster. The second algorithm that is used a lot is based on Gaussian Mixture modelling and is called the Expectation-maximization (EM) algorithm and was first proposed by Dempster et al.[8]. It is an example of a distribution based clustering algorithm. It models clusters as a distribution and tries to find the maximum likelihood for a specific amount of distributions.

The shape of groups in k-means clustering are always going to be spheres, Since they are defined by the distance from a centre point. While the shape of the clusters from the EM algorithm can be any distribution. The EM algorithm has other drawbacks, it is more prone to both overfitting and getting stuck in a local optimum. This could mean that noise in the data will influence the performance more than with k-means clustering or that the global optimal distributions will not be found. This means k-means clustering is the more robust option, while the EM algorithm could have better results if overfitting and local optimums can be avoided. For a decisive answer on which clustering algorithm is best suited, an empirical experiment needs to be carried out. This is outside

the scope of this research.

Feature extraction

Before it is possible to use any machine learning algorithm, the HTTP responses need to be transformed into input vectors consisting of numerical data. This process of transforming data into numerical vectors is called feature extraction. It has applications in different fields of artificial intelligence like image processing and natural language processing (NLP). As HTTP/1.1 responses are text based, the feature extraction methods that are possible on this are closely related to feature extraction methods in NLP. Feature extraction is a crucial part of the process, as poor feature extraction will lose information that is inside the data set. So choosing which features to extract and how to represent them will have a big impact on the effectiveness of any machine learning algorithm used.

One of the most popular feature extraction methods in NLP is a bag of words representation [36]. This is a method where every word and the frequency of the word in the specific document is saved. This can be represented as a vector where every number of the vector is the frequency of a specific word. The bag of words representation is used in popular information retrieval methods like TF-IDF [3]. There is however a difference between our data and NLP corpus. In an NLP corpus we expect important words in the text to appear more often. While on the HTTP headers this is not the case, header names and header values normally only appear once in a header. So a bag of words representation will be a vector which is both very sparse, meaning most values will be 0 and will mostly have values of 1 as the frequency of a specific word. Another drawback of the bag of words representation is that the context of the words is lost. For HTTP headers it also poses a problem, since header values are relevant in context, namely which header field it corresponds to. Since the usual text feature extraction techniques used in NLP are not suited for HTTP headers we propose two new features which we will test on the data set and evaluate.

Approach/Methodology

To help solve the problem of feature extraction on HTTP responses, we will propose two feature extraction methods that can be used on HTTP headers. We will implement these features and use them on a data set of HTTP responses. We will then evaluate the features by looking at what the distribution and other properties of the feature extraction methods output.

Data Set

The data set we are using is an internet wide scan, provided by rapid7. This data set is constructed by sending a HTTP GET-request to the root(/) of the webserver. The advantage of using this data set is that for our research no active scanning is done, therefore the analysis is not intrusive and repeating this research will also not need active scanning. One limitation of using the rapid7 data set is that for each IP address only the root request is performed. This means that webservers sharing an IP will be lost, as only the response on the request of the root will be requested and saved. Our analysis will be done on a scan of port 80, the default port of HTTP. We chose this port to have a lot of different responses to test our feature extraction on. We also have indication that C&C servers use this port to blend in with normal HTTP communication [14].

Header Field Uniqueness

The first feature we propose is a feature that calculates a uniqueness value based on the header fields that are present in a HTTP response. This uniqueness value will represent how unique the combination of headers that are present in the HTTP response are. The feature extraction process consists of two steps. First, a frequency analysis of the header field names inside the data set. For this, we count every header field name of every HTTP response inside the data set. From this frequency we calculate the uniqueness value of every header field name with the following formula:

$$Uniqueness(x) = 1 - \frac{frequency(x)}{totalResponses}$$

This will give a number between 0 and 1 for every header field name corresponding with the uniqueness of that specific header field. If a header is very common and therefore not unique, it will have a value close to 0, if a header is very unique the value will be close to 1. We now have a uniqueness value for every header field name, from this we can create a lookup table with all the header field names and their corresponding value. This table can now be used to calculate a Header Field Uniqueness for every HTTP response in the data set, this is the second step of the process. For this we add the uniqueness value of all headers that are present in the HTTP response together. We also add the inverse uniqueness value for the headers that are not in the HTTP response. We defined the inverse uniqueness with the following formula:

$$inverseUniqueness(x) = 1 - Uniqueness(X)$$

This means that a missing header that is not very unique will have a high inverseUniquness and therefore a response which misses these common headers will be more unique. For example, if a header has a uniqueness value of 0.1, it has

an inverse uniqueness value of 0.9.

The combination of all these uniqueness values of the present headers and the inverse uniqueness values of the missing headers will be the final Header Field Uniqueness of a HTTP response and will be calculated for every HTTP response in the data set. There is one problem with this approach; when the total amount of header field names is too large. Since we add a value to the uniqueness for every possible header field name, either the normal or the inverse, this becomes very computationally heavy as we need to add together a large amount of values for each HTTP response. We therefore decided to use a cutoff point at the n most frequent header field names, where the n is present in more than 1% of the data set. We do however still add uniqueness for headers that are present in the response, but outside the n most frequent header field names. For all these less common header field names we use the value of 0.99. To summarise, every HTTP response gets assigned uniqueness value based on the presence of the n most common headers, for which either a uniqueness value or inverse uniqueness value is assigned and the presence of less common headers for which the static value of 0.99 is added.

Ordering

The second feature we propose is the ordering of the headers. The information about the ordering of different header fields is a property commonly used in webserver fingerprinting tools such as HTTPrint. HTTPrint looks at the order of the Date field and Server field specifically. They show that Apache servers normally have the 'Date' field before the 'Server' field, while Netscape and Microsoft servers have the 'Server' field before the 'Date' field [15]. The feature we propose will extract the ordering of the most frequent headers. To simplify the representations of our results, we will use the n most common headers in our research, as visualization a large amount of different partial orderings is not possible. It is however possible to use the ordering of more headers in the actual feature. The exact cutoff point will be determined by the frequency of the headers in the data set. We will look to find a clear fall of in the frequency of headers and want to only include headers which are present in a significant part of the data set, preferably more than 50%.

Since not all of the n most common headers fields are present in every HTTP response the sequence will give a partial ordering indicating the order of the header fields that were present in the HTTP response. In our feature these orderings will be represented as a sequence to make the ordering more readable for humans, but to feed them into a machine learning algorithm another way of representing the ordering should be used, as just a sequence of number does not represent an ordering. One way this could be done is by saving the ordering of all the combinations of two individual headers with one bit and saving the presence of all the headers with another bit. To better illustrate this we will show an example:

If we would only look at the ordering of the three most common headers the set of headers would be: $\{1, 2, 3\}$, we would need a bit for the ordering of all the combinations of headers. So one bit for the ordering of 1 and 2, one bit for the ordering of 1 and 3, and one bit for the ordering of 2 and 3. We also need 3 bits, each saving the presence of a specific header. If we would then encode the ordering $\{2, 1, 3\}$ the first bit representing the ordering of 1 and 2 would have the value of 0, because 2 is before 1. the second bit, representing the ordering of 1 and 3 would be 1, because 1 is before 3. The third bit, representing the ordering of 2 and 3 would be 1, because 2 is before 3. The last three bits would all be 1 because all three headers were present. So the total value for the ordering $\{2, 1, 3\}$ would be 011111.

Evaluation

To evaluate our features, we need to reflect on how the feature would help a machine learning algorithm cluster or classify the data. For this, the feature needs to distribute properties over the data set in such a way that differences between HTTP responses can be identified. So looking at the distribution of the values of the feature over the data set is one of the ways we can evaluate our features. For the uniqueness feature this can be done by showing the probability density function (PDF) [25] over all the uniqueness values inside the data set. The PDF is a function that shows how a variable is distributed. This is done by showing the probability that a value of the variable would be within a range of values. If this range is small, this can be plotted as a graph showing the distribution of the variable. For the ordering feature using the PDF to show the distribution is not possible, because the ordering values are not continuous. To show the distribution of the orderings we will show what parts of the data set can be constructed with how many orderings. This will if the orderings are distributed evenly or unevenly.

Another way to evaluate if the features are usefull is to look if anomalies can already be found when only looking at the values of the feature. If there are values that are very uncommon, this can already indicate that anomalies can be found with these features. Lastly, we will try to analyse other properties that can be extracted from the features. Especially for the ordering feature, as this is not a feature which has a continuous output.

Implementation and data gathering

The data set that we used is part of the rapid7 project sonar data set to be more specific the HTTP GET root '/' scan for port 80 dated 11-01-2020 under the filename `'2021-01-11-1610327300-http_get_80.json.gz'`. The data set has 52,073,472 rows with more than 150GiB. The data set consist of the ip and the base64 encoding of the HTTP response for the whole internet IPv4 address space. Therefore in order to analyse the data, preprocessing of the data set is required. The preprocessing is done using python script that can be found in appendix A. The preprocessing should adhere to the following criteria:

- The preprocessing should not manipulate the data in anyway except decoding base64 into text and separating the three main parts of the response (status-line, headers, body). Otherwise anomalies could be lost.
- The separator of the three main parts of the response should be preserved.
- The ip of the response is used as an index for each response taking advantage of the fact that each ip by itself unique.
- Our features only look at the headers of the HTTP response. Therefore, the status-line and body have been discarded.

The result of this preprocessing was 'json' file in the following format:

```
{
  'ip': '<iPv4-address>',
  'headers':'<HTTPheaders>'
}
```

Technologies and Infrastructure

We decided to use Dask a python library for parallel processing which turned to be a good solution for our case. What make Dask powerful is that we are able to use all the advantages that Pandas has but on a distributed (cluster) of machines. Dask accepts multiple data formats without tokenizing the data which in turn preserve data originality to avoid losing anomalies even trailing white spaces and end of lines. The first step in processing the data was to convert the json file to parquet data format to reduce both the size on disk and increase the processing speed.

When Dask processes the data set it tries to read the data one partition at a time and distributes this work between the machine (cluster) members. Therefore, having a good partitioning strategy is recommended, an optimal partition size is around 1GiB an optimal partitions number is around $4 * CPUcores$. In order for Dask to read the data from disk the data should be accessible for all the cluster members by the mean of using Hadoop file system(HDFS), Google or Amazone file storage. The bottleneck for query performance when using local storage was the speed of the hard disk and in case of online storage the network speed. For faster query time Dask require that data to be loaded in memory. An important thing to consider when loading data in memory is to avoid an out of memory error by considering how Dask expands the data in memory. You may expect duple or sometimes triple the size of data on disk, but it depends on the type of data and whether the data compressed or not.

Implementation Uniqueness Feature

The implementation started by doing basic statistical analysis for the data set specifically the headers field. By doing a simple regex operations we were able to identify the frequency of the header names used in the data set. We did also a frequency count on these header names and a couple of particular header values for both the 'Content-type' header-name and the 'Server' header-name all of this files is existed in the projects GitLab repository .

To compute the uniqueness we had first to extract all the header names in the data set and then compute how many times the header-name was repeated. We then use this information to compute a uniqueness value for each header-name found in the data-set. The computed uniqueness for each header name is also in the GitLab repo. Next, we have chosen the first n most common headers out of the previous list to be used in the next step. We expect the header occurrences may decline significantly after those chosen n headers. The next step is to calculate the uniqueness value for the header names of each data entry. We can summarize the steps as follow: Step 1: Computing the "uniqueness value" for every header name which consist of :

- Reading every header with the ip
- Taking the count value from the header_count list
- Computing the uniqueness for header name x with the fomula: $Uniqueness(x) = 1 - \frac{frequency(x)}{totalResponses}$
- Storing the header name, and the uniqueness value in a new data set.

Step 2: For each HTTP response we compute the following:

- A lookup of all the headers present in the HTTP response

- A computation of the uniqueness values of all the header fields present
- A computation of the uniqueness values of all the headers absent(the inverse value)
- Adding all the values together

We expected four possible cases when computing the uniqueness for each HTTP response in the data set.

1. The header-name is in both the response and in the uniqueness values data set. In this case we add the values inside the uniqueness values. $unique = uniq\_df["header-name"]$
2. The header-name is in the response but not in the uniqueness values here we add 0.99. $unique = 0.99$
3. The header name is not in the response but is in the uniqueness values. In this case we add the inverse value of the uniqueness values. $unique = 1 - uniq\_df["headername"]$
4. The header-name not in the response and not in the uniqueness values. In this case we add nothing to the uniqueness value.

The following Pseudo code should give a better idea about the calculations while the python implementation of the code is in appendix B.

```
header_name_counts={}
uniqueness_values=[]
For each header_name in dataset:
     header_name_counts.append({'header_name': header_name.count()})

For each header_name, value in header_name_counts:
     uniqueness_values.append({'header_name': value/len(header_name_counts))

For each response in dataset:
  For each header_name in uniqueness_values:
    If header_name inside response:
      Add uniqueness_value[header_name]
    Else:
      Add (1.0 - uniqueness_value[header_name])
    For each header_name in response:
        If header_name not in uniqueness_values:
            Add 0.99
```

Implementation Ordering Feature

To extract the ordering we had to query the data to see which header field are most frequent (which also done for the uniqueness). Next, we should select the n most common headers to do the frequency count on. We then queried the data set for the index location of these n most common headers and stored the values in a sequential orders. We then calculated the frequency of this orders to do the analysis on. The actual python implementation is in appendix C.

Results

Before actually extracting the feature from the data, we needed to investigate the cutoff point of the most common headers of both the uniqueness feature and ordering feature. We do this by looking at the frequency of the most common headers. In the appendix there is a table of the 50 most common headers inside of the data set. For the uniqueness value we specified a cut off point of headers that are present in more than 1 % of the data set. This is at the thirty-fifth most common header, which is 'X-UA-Compatible'. For the Ordering we did not have as strict cutoff, instead we would look at a point where the frequency starts to fall off and headers are not present in a significant part of the data set anymore. We decided that this was at the fifth most common header. The 5 most common headers are all present in more than 80% of the data set, while the 6th most common header is only present in 27% of the data set. So choosing to use only the 5 most common headers will mean that we don't have a lot of partial orderings that are very similar but hard to represent in the results as such.

We will analyse the distribution of the two features we proposed earlier on our data set. For the uniqueness feature this can be done by using the probability density function (PDF) over the uniqueness values inside the data set. For the ordering this is not possible, because orders are not on a linear scale, meaning they can not be presented in a linear order with a set scale. For example, the difference between the sequence (header1, header2), (header1, header3) and (header1, header4) are all the same, but this can be represented on a scale, like with natural or real numbers. Therefore, we will explain other properties of the distribution of orderings we found in the data set.

Uniqueness Feature

Figure 1 shows the probability density for the uniqueness value of each data entry for the whole data set. The mean of the uniqueness of all the HTTP responses in the data set is 5.457 and the median is 4.861, both are also represented in figure 1. We choose to show the distribution only for data points with a uniqueness value between 2.5(the minimal value) and 20. There are HTTP responses with a uniqueness value higher than 20, 12065 in total this is around 0.02% of the data set. Since they are spread out between 20 and 913, which is the maximum uniqueness value and are only a small

amount of the data set. They do not show in the distribution graph. The standard deviation is 2.01.

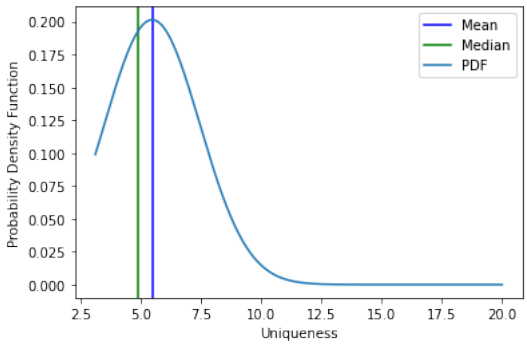

Figure 1. Uniqueness density distribution

Ordering

Since topological orderings can not be ordered in a linear way, we can not show a density distribution. To show the distribution between all the orderings we constructed table 1. This table shows how many unique orderings are needed to form the % amount of the data set that is stated as the row. For example 75% of the data set will only contain the 10 most common orderings, so the other 25% of the data set will then contain the other 291 most common orderings.

% the data set	Number of unique orderings
50%	4
75%	10
90%	32
95%	55
99%	87
99.9%	168
99.99%	230
100%	301

Table 1
The number of unique orderings needed to construct a % of the data set

Table 2 shows the 10 most common orderings and their frequency inside the data set. From this we can see that the orderings are not evenly distributed, as the most common orderings make up a very large portion of the data set, while there is a long tail with a lot of orderings that are very uncommon.

In table 3 we show the ordering of the pair of individual headers. It gives the percentage that the header specified in the row was found before the header specified in the column. For example, 4,62% of the time the Connection header is before the Date header. The final column gives the average for the values, this indicate the overall percentage that the headers was before any of the other headers.

Discussion

To explain the importance of the result above we will reflect on how the extracted features can help machine learning algorithm in clustering the data set. As we can see in Figure(1) the skewed graph show that uniqueness can be used by clustering algorithms for two main reasons: First, it is a representation that give the data a property which is numerical and continuous. A feature that is continuous has a lot more information for the machine learning algorithm to use then for example a categorical feature. Since there are a lot more unique values. The distance or difference between data point is also more different than for categorical features. For a categorical feature two data points are either in the same category or not in the same category. While for a feature with a continuous value the distance is also continuous, meaning that the difference can be any real number.

The second reason that figure(1) indicate the uniqueness feature could be useful is that the distribution is very similar to a normal skewed distribution. The distribution is cut off at the minimum, but the rest follows a normal distribution with one long tail. Since the uniqueness is following a skewed normal distribution quite closely it suggests that benign webservers will be packed in the center of the distribution, close to the median and mean, while the anomalies are far away from the mean. More investigation on the HTTP responses that are farther away from the mean could give new insights into these anomalies.

When looking at the distribution of the ordering feature we should look at Table 1. It shows that the distribution is very skewed towards common orderings. As only 4 of the 301 orderings make up over 50% of the data set and 168 of the orderings make up 99.9% of the data set. Meaning that the other 133 orderings together only cover 0.1% of the data set. The fact that some orderings are so rare even when only looking at the 5 most common headers, shows that anomalies based on the ordering can already be found, even without combining them with other properties of the HTTP response. For example the ordering [Content-type, Connection, Content-length, Server, Date] is only found in 7 out of 52 million responses. When configuring the HTTP response of a C&C servers, malicious actors could easily miss the implication of the headers ordering. Therefore, unknowingly constructing a HTTP response that looks normal, in terms of the headers present and their value, but is anomalous because of the ordering of these headers.

To fully understand the ordering feature we need to reflect on Table 3 which shows the relations between individual orderings. We can see that the connection is Usually behind the other orderings. The fact that in only 4.64% of the cases

Ordering	Frequency
Server, Content-type, Content-length, Date, Connection	9824687
Server, Date, Content-length, Content-type, Connection	8099211
Date, Server, Content-length, Connection, Content-type	7875328
Server, Date, Content-type, Connection	4528994
Content-type, Server, Date, Connection, Content-length	3612256
Date, Content-type, Content-length, Connection, Server	1469608
Date, Server, Content-length, Content-type, Connection	1239995
Date, Server, Connection, Content-type	1041208
Date, Server, Connection, Content-length Content-type	980881
Date, Content-type, Content-length, Connection	571911

Table 2
The 10 most common orderings and their frequency

	Connection	Date	Content-type	Server	Content-length	Average
Connection		4.62%	25.49%	7.68%	17.97%	13.93739275%
Date	95.38%		61.66%	35.74%	67.55%	65.08130996%
Content-type	74.51%	38.34%		18.13%	70.49%	50.36916664%
Server	92.32%	64.26%	81.87%		91.09%	82.38651886%
Content-length	82.03%	32.45%	29.51%	8.90%		38.22561181%

Table 3
The % that a header occurs before another header. the row being the header which is before, the column being the header that is after.

the Connection is before the Date, could be an indication that the webservers that have the Connection before the Date are anomalies or at least worth investigating further. Table 3 also shows that the relations between the specific headers are all different, this is another indication that there is more information to be found inside of the orderings. A machine learning algorithm might find other similarities that HTTP responses with the Connection header before the Date header have without requiring HTTP responses to have the exact same ordering of all headers. These similarities could then be used to find anomalies, for example when these similarities are not present in an HTTP response which has the Connection header before the Date.

Table 2 gives indication that the ordering feature could be even more useful in combination with other features. Since we expect that the ordering of the headers are related to other properties of the webserver, such as which webserver version is used. For example, the most common ordering is the ordering that is usually found in HTTP responses which have server header value 'AkamaiGhost'. Another indication of the correlation between this ordering and the server value 'AkamaiGhost' is that the frequency of the ordering; 9824687 is very close to the amount of HTTP responses with 'AkamaiGhost' as their server value; 9808983. So finding AkamaiGhost server with a different ordering could also be a way of identifying anomalies that could indicate a C&C server configured without keeping the ordering of headers in mind.

Conclusion

We have given an overview of which machine learning algorithms could be used on a data set of HTTP responses. Classification algorithms are not suited for this problem, because there is no clear way to label the data. This leaves unsupervised learning as the best methods of using machine learning. Different clustering algorithms can be used and have their own benefits and drawbacks.

One of the problems on using the HTTP responses with machine learning is feature extraction. Since HTTP responses do not have the same syntax and semantics as natural language, the most used NLP feature extraction methods are not well suited for HTTP responses. We have proposed two new feature extraction methods and have empirically tested them on a data set of HTTP responses.

The results of the implemented feature show that they could be used with machine learning algorithms. The uniqueness feature gives every response a uniqueness value which is numerical and continuous. Which means that it is very suitable for machine learning algorithms. The ordering feature also shows that it stores a lot of information about a response. It distributes a lot of different orderings, although the majority of the data set has an ordering from the top 10 most common orderings. But these common orderings

do seem correlated with specific webservers version, so combining the ordering with other information could be a way to find anomalies. Both the uniqueness and the ordering feature also show that anomalies can be found with these features. As there are outliers found in both features. For the uniqueness there is a long tail of HTTP responses with a very high uniqueness values. For the ordering feature there are a lot of orderings that are very uncommon, which indicates the HTTP responses with these values could be anomalies.

Our research shows how machine learning could be used to hunt down C&C servers, but how effective these methods are can not be concluded from our research. For this, experiments using machine learning on HTTP responses should be conducted. In the next section we will go into future research that could be done building upon our research.

Future work

There are a few research topics that emerge from our research. Using our proposed features with machine learning to hunt down C&C servers in a large data set of HTTP responses is a clear new topic. As our features were constructed with this specific goal in mind. This could also answer the question if using machine learning to tackle this problem is an effective strategy to find C&C servers. Investigating the performance of different machine learning algorithms is another topic for future research. This could give more insight into which algorithms are best suited for a data set of HTTP responses. Another point for future research is expanding the ordering feature beyond the 5 most common headers. For the simplicity of analysis and representation that was not done in our research, but this could lead to finding more anomalies. The combination of our proposed feature with other features is also a topic for future research. One example of this is the combination of webserver version and ordering to lead to better results. As the ordering of headers is correlated with the webserver version. Further investigation into what other features can be extracted is also an interesting topic for future research.

Other features

Since we think other ways of feature extraction would be an interesting topic for future research we will explain a few different ways this could be done with the knowledge we gained from working with a data set of HTTP responses. One way is by constructing a binary feature. This can be done by finding properties that could indicate a difference between servers. For example, a known HTTP header is the content-length, which should indicate the length of the HTTP body. It is possible to construct a query which finds all the HTTP responses where this content-length does not match with the actual HTTP body. This could indicate that the content-length is not actually based on length of the body, but hard coded. It is possible that this is caused by a misconfiguration of a webserver, but could also indicate a malicious C&C server which is not configured with enough knowledge of the HTTP protocol.

Similar to this, categorical features can also be constructed. An example of this could be the values found in the Encoding. There are two values that are very common, gzip and chunked. There are other values, but those are very rare and could be combined into a third category, we will also need a fourth category, for the HTTP responses which do not have an encoding field. To encode a categorical feature, one-hot encoding can be used. One-hot encoding will assign a bit for every category of the feature. Then each data point will have zeroes on all the bits except the bit of the category it is in, this will be a 1. So for our example each HTTP response will have four numbers corresponding to the four different categories, of which one will be a 1 depending on the Encoding header inside the response.

Trying to use onehot encoding to build a feature of every header value might sound like another way of using categorical features. This is unfortunately problematic for headers which have a lot of different values. If we would for example use one-hot encoding for the server value, we will have over 85 thousand different values for the different server versions. This means that every HTTP response would need 85 thousand bits to save this feature. Most of these bits will be zero, so the total feature matrix would become a enormous matrix of primarily zeroes. Note that this way of encoding these values will look very similar to the bag of words feature extraction we explained earlier. It also has the same problem where the matrix will be very sparse and only containing the number 1 on entries that do have a value. One way of trying to prevent this is by using less categories, for the server field this can be done by combining all the versions of a specific server into one category and all the server values that are very unique into an 'other' category. However, this may lead to a loss of information.

References

1 Ahmed, M., Mahmood, A. N., & Hu, J. (2016). A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, *60*, 19–31.

2 Barnett, V., & Lewis, T. (1984). Outliers in statistical data. *Wiley Series in Probability and Mathematical Statistics. Applied Probability and Statistics*.

3 Beel, J., Gipp, B., Langer, S., & Breitinger, C. (2016). Paper recommender systems: A literature survey. *International Journal on Digital Libraries*, *17*(4), 305–338.

4 Bigdeli, E., Mohammadi, M., Raahemi, B., & Matwin, S. (2018). Incremental anomaly detection using two-

layer cluster-based structure. *Information Sciences*, *429*, 315–331.

5 Bortolameotti, R., van Ede, T., Caselli, M., Everts, M. H., Hartel, P., Hofstede, R., Jonker, W., & Peter, A. (2017). Decanter: Detection of anomalous outbound http traffic by passive application fingerprinting. *Proceedings of the 33rd Annual computer security applications Conference*, 373–386.

6 Brezo, F., de la Puerta, J. G., Ugarte-Pedrero, X., Santos, I., & Bringas, P. G. (2013). A supervised classification approach for detecting packets originated in a http-based botnet. *CLEI Electronic Journal*, *16*(3), 2–2.

7 Cai, T., & Zou, F. (2012). Detecting http botnet with clustering network traffic. *2012 8th international conference on wireless communications, networking and mobile computing*, 1–7.

8 Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, *39*(1), 1–22.

9 Fernandes, G., Rodrigues, J. J., Carvalho, L. F., Al-Muhtadi, J. F., & Proença, M. L. (2019). A comprehensive survey on network anomaly detection. *Telecommunication Systems*, *70*(3), 447–489.

10 Gu, G., Perdisci, R., Zhang, J., & Lee, W. (2008). Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection.

11 Gu, G., Zhang, J., & Lee, W. (2008). Botsniffer: Detecting botnet command and control channels in network traffic.

12 Hoque, N., Bhuyan, M. H., Baishya, R. C., Bhattacharyya, D. K., & Kalita, J. K. (2014). Network attacks: Taxonomy, tools and systems. *Journal of Network and Computer Applications*, *40*, 307–324.

13 *Httprecon project - advanced http fingerprinting*. (n.d.). Retrieved January 24, 2021, from https://www.computec.ch/projekte/httprecon/?

14 *Identifying cobalt strike team servers in the wild*. (n.d.). Retrieved January 7, 2021, from https://blog.fox-it.com/2019/02/26/identifying-cobalt-strike-team-servers-in-the-wild/

15 *An introduction to http fingerprinting*. (n.d.). Retrieved January 8, 2021, from https://net-square.com/httprint_paper.html

16 Kheir, N. (2013). Behavioral classification and detection of malware through http user agent anomalies. *Journal of Information Security and Applications*, *18*(1), 2–13.

17 Lakhina, A., Crovella, M., & Diot, C. (2004). Diagnosing network-wide traffic anomalies. *ACM SIGCOMM computer communication review*, *34*(4), 219–230.

18 Laperdrix, P., Bielova, N., Baudry, B., & Avoine, G. (2020). Browser fingerprinting: A survey. *ACM Transactions on the Web (TWEB)*, *14*(2), 1–33.

19 Lee, D., Rowe, J., Ko, C., & Levitt, K. (2002). Detecting and defending against web-server fingerprinting. *18th Annual Computer Security Applications Conference, 2002. Proceedings.*, 321–330.

20 Lu, W., Rammidi, G., & Ghorbani, A. A. (2011). Clustering botnet communication traffic based on n-gram feature selection. *Computer Communications*, *34*(3), 502–514.

21 MacQueen, J. et al. (1967). Some methods for classification and analysis of multivariate observations. *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, *1*(14), 281–297.

22 Mathew, S. E., Ali, A., & Stephen, J. (2014). Genetic algorithm based layered detection and defense of http botnet. *International Journal on Network Security*, *5*(1), 50.

23 Münz, G., Li, S., & Carle, G. (2007). Traffic anomaly detection using k-means clustering. *GI/ITG Workshop MMBnet*, 13–14.

24 Nelms, T., Perdisci, R., & Ahamad, M. (2013). Execscent: Mining for new c&c domains in live networks with adaptive control protocol templates. *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 589–604.

25 Parzen, E. (1962). On estimation of a probability density function and mode. *The annals of mathematical statistics*, *33*(3), 1065–1076.

26 Perdisci, R., Ariu, D., & Giacinto, G. (2013). Scalable fine-grained behavioral clustering of http-based malware. *Computer Networks*, *57*(2), 487–500.

27 Rafique, M. Z., & Caballero, J. (2013). Firma: Malware clustering and network signature generation with mixed network behaviors. *International Workshop on Recent Advances in Intrusion Detection*, 144–163.

28 Sakib, M. N., & Huang, C.-T. (2016). Using anomaly detection based techniques to detect http-based botnet c&c traffic. *2016 ieee international conference on communications (icc)*, 1–6.

29 Sood, A. K., Zeadally, S., & Bansal, R. (2017). Cybercrime at a scale: A practical study of deployments of http-based botnet command and control panels. *IEEE Communications Magazine*, *55*(7), 22–28.

30 Venkatesh, G. K., Srihari, V., Veeramani, R., Karthikeyan, R., & Anitha, R. (2013). Http botnet detection using hidden semi-markov model with snmp mib variables. *International Journal of Electronic Security and Digital Forensics*, *5*(3-4), 188–200.

31 Wurzinger, P., Bilge, L., Holz, T., Goebel, J., Kruegel, C., & Kirda, E. (2009). Automatically generating models for botnet detection. *European symposium on research in computer security*, 232–249.

32 Yamauchi, K., Hori, Y., & Sakurai, K. (2013). Detecting http-based botnet based on characteristic of the c & c session using by svm. *2013 Eighth Asia Joint Conference on Information Security*, 63–68.

33 Yang, K.-x., Hu, L., Zhang, N., Huo, Y.-m., & Zhao, K. (2010). Improving the defence against web server fingerprinting by eliminating compliance variation. *2010 Fifth International Conference on Frontier of Computer Science and Technology*, 227–232.

34 Yu, Y., Yan, H., Guan, H., & Zhou, H. (2018). Deephttp: Semantics-structure model with attention for anomalous http traffic detection and pattern mining. *arXiv preprint arXiv:1810.12751*.

35 Zarras, A., Papadogiannakis, A., Gawlik, R., & Holz, T. (2014). Automated generation of models for fast and precise detection of http-based malware. *2014 Twelfth Annual International Conference on Privacy, Security and Trust*, 249–256.

36 Zhang, Y., Jin, R., & Zhou, Z.-H. (2010). Understanding bag-of-words model: A statistical framework. *International Journal of Machine Learning and Cybernetics*, *1*(1-4), 43–52.

37 Zolotukhin, M., Hämäläinen, T., Kokkonen, T., & Siltanen, J. (2014). Analysis of http requests for anomaly detection of web attacks. *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, 406–411.

Appendix A
Python Script for Preprocessing the data set

```
from io import BytesIO
import base64
import pandas as pd
import json
import hashlib
import re
import numpy as np
import multiprocessing as mp
from fastparquet import ParquetFile, write
from os import listdir


def preprocessing(row):
    try:
        #print(row[2])
        #decoding to bytes opbject
        base64_message = row[2]
        base64_bytes = base64_message.encode('utf8')
        message_bytes = base64.b64decode(base64_bytes)
        #trying to split the headers and body in different ways
        splitbytes = re.search(b'(?:\r?\n){2}',message_bytes).group()
        splitbody =re.split(b'(?:\r?\n){2}', message_bytes,1)
        headers = splitbody[0]
        body = splitbody[1]
        #print(message_bytes)
        #print(headers)
        #print(body)
        #trying to split the statusline from the headers in 2 different ways
        try:
            status, header = headers.split(b"\r\n", 1)
            statussep = "\r\n"
        except:
            try:
                status, header = headers.split(b"\n", 1)
                statussep = "\n"
            except:
                #print('no headers found')
                #print(i)
                #print(headers)
                #print(message_bytes)
                header = b''
                statussep = splitbytes.decode("utf-8")[:]
                splitbytes = b''
        #trying to make a string out of the status and headers
        try:
            finalstatus = status.decode("utf-8")
            finalstatus += statussep
            finalheader = header.decode("utf-8")
            finalheader += splitbytes.decode("utf-8")
        except:
            #print('decoding error')
            return {'ip': row[2]}
        #exporting headers to new dataframe
        return {'ip': row[1],'statusline': finalstatus, 'headers': finalheader, 'bodylength': len(body), 'bodyhash': hashlib.sha256(body).hexdigest()}
    except:
        #print('decoding error')
        return {'ip': row[1]}
    #handling decoding error to different dataframe
    #print("decoding error")
    #print(i)
    #errorData100 = errorData100.append({'ip': http100_df['ip'][i],'data': message_bytes}, ignore_index=True)

def main():
    pool = mp.Pool(mp.cpu_count())
```

```
    fileNames = listdir('pathToFile/file.parquet/')
    counter = 0
    for file in fileNames:
        if file.endswith('.parquet'):
            http100_df = ParquetFile('pathToFile/file.parquet/'+file)
            http100_df = http100_df.to_pandas()
            result = pool.imap(preprocessing, http100_df.itertuples(name=None), chunksize=10)
            output = [x for x in result]
            counter += 1
            print(counter)
            outputjson = json.dumps(output)
            with open('dataSet.json', 'a') as outfile:
                outfile.write(outputjson)
            if counter > 5:
                break

if __name__ == "__main__":
  main()
```

Appendix B
Python Implementation for the Header Uniqueness Feature

```
import pandas as pdd
import dask.dataframe as dd
from dask.distributed import Client


#uniqueness.json contain frequency count of every header divided by the total count of all the headers
udf = pdd.read_json('gs://pathToFile/uniqueness.json')
def test_fun(headernames):
    uniqueness = 0
    headernames['bool'] = headernames['header_name'].isin(udf.index)
    #headerNamesList.columns=['ip','match','bool']
    for index, row in headernames.iterrows():
        if not row['bool']:
            #print(0.99)
            uniqueness += 0.99


    udf['bool2'] = udf.index.isin(headernames['header_name'])
    for index2, row2 in udf.iterrows():
        if row2['bool2']:
            uniqueness +=udf['uniqueness'][index2]
        else:
            uniqueness +=1.0 - udf['uniqueness'][index2]
    return uniqueness


for i in range(12,27):
    print(i)
    client = Client('145.100.104.126:8786')
    df = dd.read_parquet("gs://pathToFile/part."+str(i)+".parquet",engine='fastparquet')
    df = df.repartition(npartitions=4*60)
    df = df.set_index('ip')
    df1 = df['headers'].str.extractall('(\n[\w\-]*\:|^[\w\-]*:)')
    df1.columns=['header_name']
    df1['header_name'] = df1['header_name'].str.lstrip()
    df1= df1.reset_index()
    df1=client.persist(df1)
    df2 = df1.groupby('ip').apply(lambda group: test_fun(group) , meta=('int')).compute(schduler='processes')
    df2.to_json('part.0'+str(i)+'-uniquenessAll.json')
    client.restart()
```

Appendix C
Python Implementation for the Header Ordering Feature

```
import dask.dataframe as dd
from dask.distributed import Client



def ordering(row):
    test = row.sort_values()
    result = []
    for i in test.index:
        if test[i] > -1:
            result.append(i)
    #test['order'] = result
    return result

for i in range(1,27): # this is the number of dataset partition files
    print(i)
    client = Client('145.100.104.126:8786')
    df = dd.read_parquet("gs://pathToFile/part."+str(i)+".parquet",engine='fastparquet')
    df = df.repartition(npartitions=4*60)
    df = df.drop_duplicates(subset=['ip'])
    df = df.set_index('ip')
    df = client.persist(df)
    df2 = df['headers'].str.find('Connection:')
    temp= df['headers'].str.find('Date:')
```

```
    df2= df2.to_frame().merge(temp.to_frame
    (),left_index=True, right_index=True)
    listNames=['Content-Type:','Server:','
    Content-Length:']
    for name in listNames:
        temp=df['headers'].str.find(name)
        df2= df2.merge(temp.to_frame(),
    left_index=True, right_index=True)
    df2.columns=[1,2,3,4,5]
    newResult = df2.apply(lambda raw:
    ordering(raw),axis=1,meta=(None, 'object'
    )).compute(schduler='processes')
    newResult.to_json('orderAll/part.'+str(i
    )+'-orderAll.json')
    client.restart()
```

Appendix D
Table of the 50 most common HTTP headers

Header Name	Frequency	percentage
Connection:	50119601	96.36%
Date:	47935778	92.16%
Content-Type:	47612247	91.54%
Server:	46010194	88.46%
Content-Length:	42092999	80.93%
Expires:	14288570	27.47%
Last-Modified:	12713100	24.44%
ETag:	10555627	20.29%
Mime-Version:	10279962	19.76%
Location:	9353744	17.98%
Accept-Ranges:	9183534	17.66%
Content-Encoding:	8772700	16.87%
Vary:	8296925	15.95%
Cache-Control:	8176182	15.72%
Transfer-Encoding:	6547003	12.59%
X-Powered-By:	5460656	10.50%
Set-Cookie:	5027493	9.67%
X-Frame-Options:	4639789	8.92%
Pragma:	3381981	6.50%
X-Content-Type-Options:	2086189	4.01%
X-XSS-Protection:	1822288	3.50%
Via:	1558692	3.00%
Content-type:	1269677	2.44%
X-Cache:	1167272	2.24%
WWW-Authenticate:	1036940	1.99%
Upgrade:	992345	1.91%
X-Amz-Cf-Pop:	946771	1.82%
X-Amz-Cf-Id:	946768	1.82%
cf-request-id:	858603	1.65%
CF-RAY:	855256	1.64%
Strict-Transport-Security:	804800	1.55%
Content-Security-Policy:	789614	1.52%
Access-Control-Allow-Origin:	666621	1.28%
Referrer-Policy:	602841	1.16%
X-UA-Compatible:	593810	1.14%
P3P:	444120	0.85%
Cache-control:	435249	0.84%
X-Squid-Error:	423976	0.82%
Content-Language:	420092	0.81%
Content-length:	413783	0.80%
Etag:	407530	0.78%
Link:	388293	0.75%
CONTENT-LENGTH:	362065	0.70%
content-type:	347457	0.67%
CONNECTION:	343699	0.66%
CONTENT-TYPE:	343652	0.66%
X-Redirect-By:	340918	0.66%
Access-Control-Allow-Headers:	290766	0.56%
X-AspNet-Version:	281662	0.54%
Access-Control-Allow-Methods:	273465	0.53%
Total response:	52013837	100%