INTERFERENCE ON WEBSITE FINGERPRINTING

By

Weiran Lin

Senior Thesis in Computer Engineering

University of Illinois at Urbana-Champaign

Advisor: Professor Nikita Borisov

December 2018

Abstract

Although nowadays Internet users are better and better protected by advanced encryption mechanisms, their privacy is still not yet well protected: eavesdroppers can use the patterns of network traffic to learn sensitive information. One such attack is website fingerprinting, which adopts machine learning techniques to detect which webpage the user is visiting. In this thesis, we try to answer the question to what extent new web technologies, particularly HTTP/2 and server push, could interfere with website download packet traces, and hence defend against website fingerprinting. In our experiment, we extracted our website models from real world-sites and evaluated HTTP/2 server push and size padding with website fingerprinting on these models. The result shows that HTTP/2 and server push could lower the accuracy of website fingerprinting, and random size padding could further decrease this accuracy.

Subject Keywords: website fingerprinting; server push; HTTP/2

Contents

1. Introduction 1

2. Literature Review 2

3. Research Approach 3

4. Description of Research Results 7

5. Conclusion 12

References 13

1. Introduction

Although protocols like TLS encrypt contents of packets in website browsing to prevent eavesdropping, the sizes, numbers and directions of packets are still visible to anyone who can see the network traffic such as ISPs and routers. Hence, the connections between clients and servers are still vulnerable to website fingerprinting attacks, which can reveal the specific webpage the user is visiting by observing network traffic.

In our experiment, we studied the influence of HTTP/2 server push on website fingerprinting. HTTP/2 streams the downloading, eliminates the time overhead due to the website structure and hence reduces the information that could be potentially learned. In addition, we also tested random size padding in our experiment which could further prevent a website from being recognized according to its total size. Meanwhile, server push eliminates time needed for requests after the initial one, and hence decreases the possibility of information leakage due to delay in time caused by the dependency structure.

To be able to observe the whole process of website downloading, we extracted models from real world sites, specifically the top 99 sites from the Alexa top sites list [1]. These models have the same sizes and dependency hierarchy as the original sites. We performed measurements of website fingerprinting on these models and compare the accuracy with interference techniques applied.

Our results show that HTTP/2 server push and size padding could reduce the vulnerability to website fingerprinting.

2. Background

Generally speaking, In HTTP/1.1 when a client sends a request with URL of a website to the respective server, the server will return a document list html file, which contains names of files to be downloaded next; the client will then send a request for each file mentioned in this html file and the server will send a response for each file requested, which may incur further requests and responses afterwards. Downloaded html files, JavaScript files and style sheets could trigger downloads of more files, and hence formulate a tree-like dependency structure of website downloading. This is a strict hierarchy as the browser must finish downloading a file before performing the downloading of any other files that it initiates. From traffic analysis, there is a clear time gap between the downloading timeline of this initiator, and downloading of the following files.

Website fingerprinting is the technique of extracting features of timing, packet direction and packet size information from network traffic, with machine learning algorithms, to tell the website that the client is visiting. When adversaries watch the traffic of website browsing from the router, although the client-server connection is encrypted, and the packet content is invisible to them, they may still gather the traffic information and conducts attacks. Previous studies show that eavesdroppers can even perform such attacks on the Tor network [2, 3].

Different from HTTP/1.1, HTTP/2 uses multiplexing. When the browser starts to visit a website, all the requests go through one singe TCP connection, while all the responses are also sent through this connection, in parallel, asynchronously. There is no forceful dependency structure in downloading, and hence there is less strict time order.

In our studies, we want to verify the idea that using HTTP/2 could be a defense against the website fingerprinting attack, as the attackers may lose timing information due to the lack of dependency hierarchy. Previous work on defenses against website fingerprinting [4, 5, 6] manipulated several features but are shown to be ineffective when the adversaries use more advanced machine learning skills [7, 8]. Recent approaches of the defenses [9, 10] adopt the constant-rate stream of communication between server and client, which has a provable upper bound of information website fingerprinting attackers could get. Our defense with HTTP/2 server push and padding also has constant rate, and we want to find out the efficiency of such countermeasures in our experiment.

3.Research Approach

To fully understand the process of browsing, we build models of websites based on browser instrumentation [11] so that we can observe the whole process of a browser visiting a website. These models are supposed to have the same dependency structure and same size for each file.

In Chromium browser, the downloads triggered by visiting a website can happen in parallel if they do not have dependency relationships. In our case, we only considered five types of responses: script, document, stylesheet, image and fonts. Among these five, fonts and images are not able to initiate downloads, and they are always leaf nodes in the dependency tree structures of websites. On the other hand, scripts, documents and stylesheets are able to initiate downloads: documents, files with ".html" suffix, could initiate downloads of all five types of files; scripts, java scripts with ".js" suffix, could also initiate downloads of all five types of files; stylesheets, files with ".css" files, could only initiate downloads of other stylesheets and fonts. Specifically, the Chromium browser instrumentation assumes that all fonts are downloaded from html files even though fonts may refer to stylesheets and within a single website, all fonts start downloading in parallel as soon as the all stylesheets finish downloading.

In our experiment, we used the embedded functions of the Chromium browser. Specifically, four functions are used: requestWillBeSent, loadingFinished, responseReceived, and dataReceived. These functions are triggered accordingly at different stages in the downloading process of corresponding to their names. We obtained information about the download of each file from the parameters of these functions. When we collected data for a website, we always waited for 5 seconds first, visited only one site at a time, created a tab, visited the website in this tab while recording the information, waited for loading for another 30 seconds, and finally closed the tab. Hence, the traces of two websites would not interfere with each other.

From these functions, we can know the URL, type, initiator, and size of each file downloaded. In the dependency tree structure, every file has an initiator besides the root, the top-level document. Based on the information, we could produce a dummy website that mimics the downloading behaviors of a real-world website and hence has similar downloading traces. An essential characteristic of the downloading traces is the size of each file downloaded, and hence we needed to produce files of given sizes, and we achieved that by writing comments of computed sizes. Meanwhile, loading commands and some of the formattings are still necessary; for instance, the Chromium browser aborts downloading an image when the first 65536 bytes of the image are not in the right format, as the browser categorizes the image as corrupted. Thus the reproduced images must have a correct format. Meanwhile, we believe that the

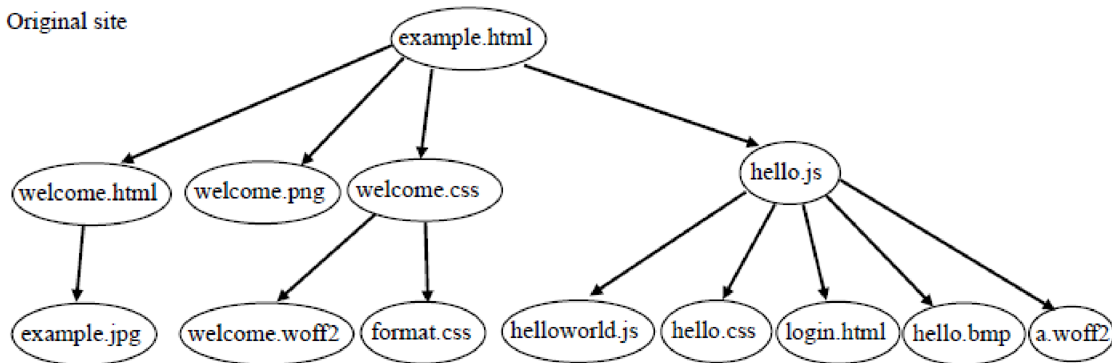
differences between formats within one of the five types have no effects on the overall downloading traces, and thus we picked one format for reproduced files of each type: all images are in gif format, with repeated comment blocks; all fonts are in woff format, with simply repeated “A” characters as correctness of font file is not checked while downloading. As mentioned before, these two types of files have no loading commands. In addition, the Chrome browser attributed the loading commands in stylesheets, either to load a font or to load another stylesheet to the nearest document (nearest parent document file in loading structure, recursively). Admittedly, we lost information about the structure of stylesheets loading here and in our models stylesheets could not load other files either, so they consist of only comments starting with “/*”, ending with “*/” and consisting of repeated characters “A” in between. In the scripts, we used “document.write(…);” commands loading stylesheets, other scripts, and images. All files have their corresponding tags. Documents are loaded by “document.createElement(‘iframe’);” and “document.body.appendChild(…);” while fonts are loaded by “document.createElement();” and “document.fonts.add();”. Comments starting with “/*”, ending with “*/” and consisting of repeated characters “A” in between follow the loading commands. In HTML documents, in the head part, stylesheets are loaded first, documents next and then the scripts. If there are fonts loaded from this document, there is also a style part at the end of the head part. Each font file is loaded independently in a font family by the font-face rule and a class of this font-family is claimed. Following the head part, there is a body part with images loaded first and then there is a character “A” with span tag for each font class to force loading the font. Ultimately in the body part there are comments starting with “<!--”, ending with “-->” and consisting of repeated characters “A”. Repeated characters “A” appear in the later four types of files. The number of “A” is such that the size of the file equals the size of the corresponding original file. This is always feasible since a character is always one byte. For each model, all files we generated are in a single directory.

To generate the hierarchy within each model, we started by labeling the files that need to be generated. Besides “index.html”, other documents are named “index0.html”, “index1.html” … while images were named “image0.gif”, “image1.gif” …and fonts were named “font0.woff”, “font1.woff” … Stylesheets and scripts have similar names. These names also appear in the loading commands. Then we went over all the information that was collected from the four embedded functions of the Chromium browser and stored in a json file. We firstly went over all the file information and found the document file with no initiator and with the earliest starting time, which is most likely to be our top-level document. As there are possible redirecting issues, we did not limit this document to files that come from servers with the IP address or URL that our initial request was sent to. Then we went over the information of all other files.

Now we only considered files with initiator, as we did not consider XHR, and data connections with no initiator were not and should not be taken into account. For eligible files, as we mentioned, since in our models assume that stylesheets, images, and fonts cannot trigger further downloading, if the file belongs to these three types, we generated the corresponding file directly. Otherwise, where the file is either a script or a document, for each of the five categories of other files, we found all files that are initiated from the current file. Then we wrote the loading commands of different categories in their corresponding places and wrote comments with calculated sizes so the total file size is the same as the original one. By doing this, we built the loading structure of our models.

In order to prove that these mimic websites have the correct size and loading hierarchy characteristics, we also loaded websites that we produced, by running Apache2 inside docker to serve the mimic website on our own server while using the headless chromium browser to visit this server and measure the sizes of this download as we did before. After comparing with size characteristics of original corresponding real-worlds site, we could make sure that these two sites are same with respect to sizes.

Figure 1 and 2 show the structure of a website with its corresponding model.



This is the dependency graph of an example site.
In the following two graphs:
example.html and index.html have the same sizes;
welcome.html and index0.html have the same sizes;
welcome.png and img0.gif have the same sizes;
welcome.css and sheet0.css have the same sizes;
hello.js and script0.js have the same sizes;
example.jpg and img1.gif have the same sizes;
welcome.woff2 and font0.woff have the same sizes;
format.css and sheet1.css have the same sizes;
helloworld.js and script1.js have the same sizes;
hello.css and sheet2.css have the same sizes;
login.html and index1.html have the same sizes;
hello.bmp and img2.gif have the same sizes.
a.woff2 and font1.woff have the same sizes
Assuming all files on the above graph are from the same server, example.com
In the following code example, … represents repeated character "A"s.

Figure 1: the dependency structure of an original site

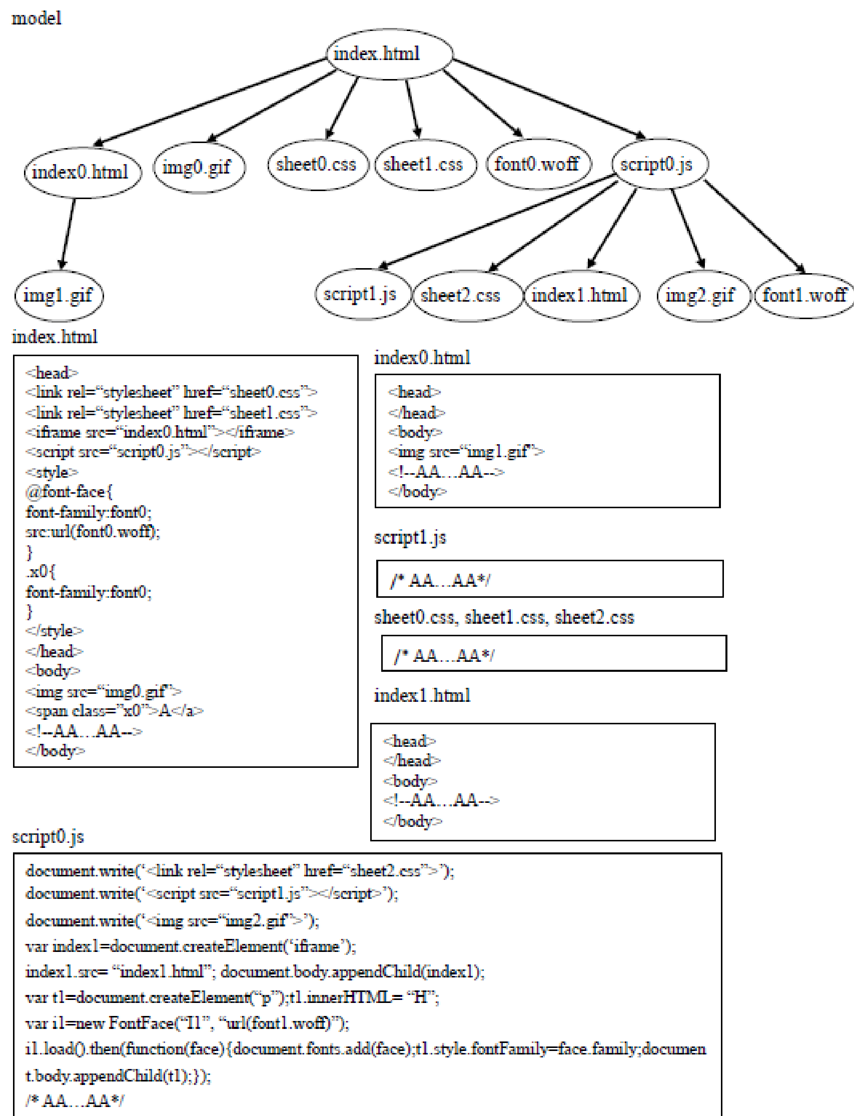


Figure 2: dependency structure of the model site

There are already established website fingerprinting techniques, such as Wang's k-nearest neighbors' method, which uses 3766 features. Each visit of a website is recorded as a vector of 3766 elements. In our experiment, we found that recursively reducing the 3766 features to 100 most relevant features could boost the accuracy, and we built a random forest classifier which uses these 100 features. In our tests, we use closed-world classification, which could have generally higher accuracy than open-world, and thus could better show the results of interference on website fingerprinting.

4. Description of Research Results

Table 1: Website fingerprinting accuracy with different server settings

Experiment	Accuracy
Original sites	91.3%±4.3%
One model per site, HTTP/1.1	99.9%±0.5%
One model per trace, HTTP/1.1	80.2%±3.8%
HTTP/2 with server push	74.2%±4.3%
HTTP/2 without server push	80.4%±2.5%
HTTP/2 with 25% padding	68.5%±4.6%
HTTP/2 with 25% padding, padded filenames	63.5%±8.0%

We used docker containers, independent computer environment in our experiment as both the client side and server side. We listened to traffic between them. The client side is a docker that contains a headless Chromium crawler, performs as a browser of the client. The server side could be either real-world websites, or docker containers with Caddy web servers. Caddies are web server models that allow us to set to HTTP/1, HTTP/2, or HTTP/2 server push mode. The process is shown in the picture below. Three kinds of information are taken from traffic: directions, relative time and sizes of packets.

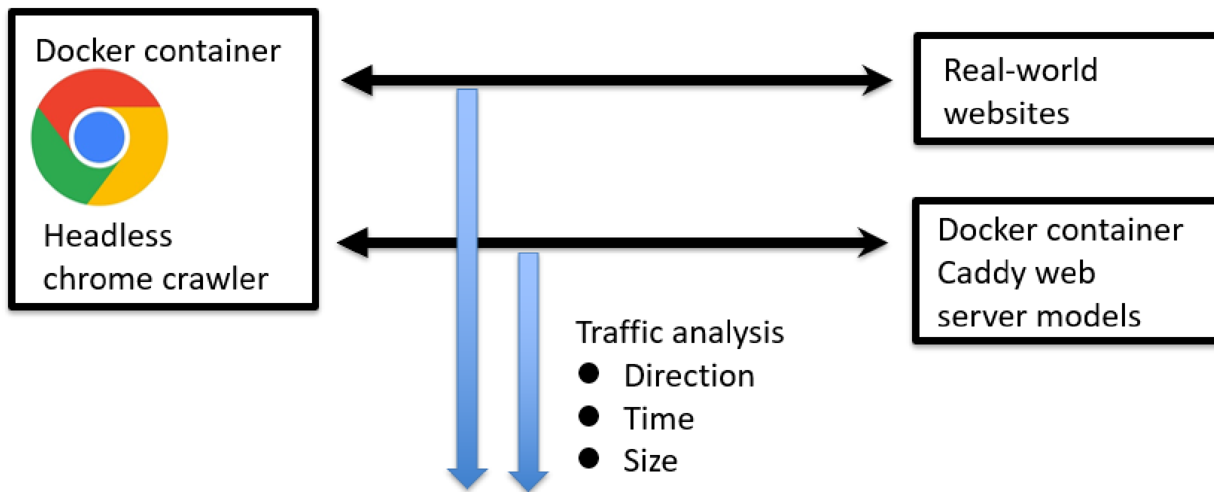


Figure 3: the data collecting process

We started with the Alexa top 100 sites. One of the websites was down during our experiment, so we ended up with 99 sites. Thirty instances are recorded for each site, and there are 99*30=2970 such instances. We collected the trace of original websites by running a tcpdump while visiting the sites in a headless chromium browser, one at a time. The pcap file generated then could be parsed into the formatted data that website fingerprinting needs, which includes time, package size and direction information. The result of our random forest website fingerprinting shows that these websites have high susceptibility to website fingerprinting.

Then we collected models from these 99 sites as mentioned above. We create one model for each website. For each model, all files we generated are in a single directory which has the same name as the URL of the website corresponding to the model. Then for each of them, we create a docker network, attached a docker image using the model we created onto it and collected 30 instances using a crawler() to generate pcap files. After parsing these pcap files from docker networks, we have an extremely high accuracy in this new dataset.

In order to further improve our modeling process, we visited each original website 30 times from the Chromium browser while collecting information with the embedded functions and generating models according each trace of the visit. There are 99*30 models. In this dataset, only one instance is collected from each model, so there are also 2970 instances in this dataset.

Admittedly, there are still differences between our models and the real websites and that would influence the classification accuracy measurement. The classification results on these datasets show that while the classification has higher accuracy when the dataset originates from one model per site, it has much lower accuracy when the dataset originates from one model per trace, even lower than the original website. This is expected, as even the same webpage does not have the same size for each visit, and Figure 4 shows the size distribution looks our 99 sites. The web servers could have dynamic site behavior, as the website may not be the same upon each visit. For example, YouTube may prompt different videos. In addition, we padded the URL, as we found that the URL alone generates 3.2 bits of entropy in our dataset, and website fingerprinting can always distinguish websites of different URL length by just watching the length of the first request.

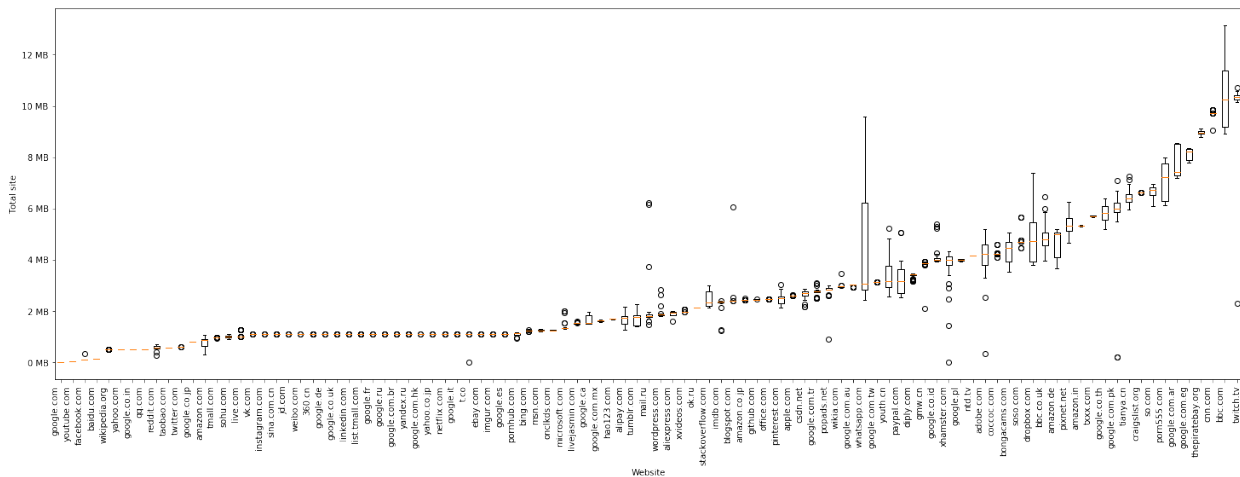


Figure 4: Size distribution of the 99 sites

In addition, to eliminate potential information leakage from different lengths of the ClientHello request due to website names, we label and each website from 0 to 98 and each visit of the same website (from 0 to 29). We named the models with, instead of original website names or anything related to that, a 14 character string, starting with character s, followed by the website label in 6 digits, followed by a '_' character and then the visit label in 6 digits For instance, the model built according to website 14, visit 6 would be called "s000014_000006". This is also one of the reasons that one model per trace has significantly lower accuracy, as the site name has 3.2 bits of entropy in our datasets.

To study the efficiency of website fingerprinting on HTTP2 Push traffic, each of the caddy files also contains push commands and pushes all resource in the model besides the index.html file. This push command is not activated in the HTTP/1.1 mode but utilized in the HTTP2 mode. Like before, we started a docker network, attached a caddy image to it, but with HTTP2 option this time, and used the crawler to generate 1 instance of downloading traces. The classification results show that HTTP2 Push could decrease the accuracy of website fingerprinting

However, the accuracy is still quite high. One main feature that contributes to this accuracy is the total size of websites. In fact, if we train a random forest classifier using only the total size feature, the accuracy is around 58.1%. A common method to minimize information leakage from sizes is padding, which adds random size "garbage" information to the website and is ultimately not displayed to users. To learn the efficiency of size padding as a defense against website fingerprinting, we also build new models with padding. The padding file is a javascript, named "padding.js" that contains only comments that consist of comment symbol "/**/" and "A" characters between the symbols. The size of the padding file is a uniformly distributed random number, between 4 bytes and 50% of the total size of the original files, i.e. with an average of 25% of original size. The padding file must be at least 4 bytes as that is the size of the comment symbols. The padding file is in the same directory as all other files of that model. Meanwhile, padding.js is added to the corresponding caddy file as the last push parameter, and it is also added to index.html as the last item loaded in the body, right in front of all the comments. Figure 5 shows how padding interferes with website fingerprinting, while we use HTTP/2 server push, when we use only the total size feature and when we recursively select the top 100 features. From our test results we can tell that with HTTP/2 server push, a small amount of padding could significantly lower the accuracy of website fingerprinting, and the more the padding is, the lower the accuracy is. Nonetheless, with average 10% padding and HTTP/1, the accuracy of classification is 79.1%±4.3%.

Compared to the accuracy of 80.2%±3.8% in the dataset of HTTP/1 with no padding, it shows that padding alone does not have much effect on website fingerprinting in HTTP/1.

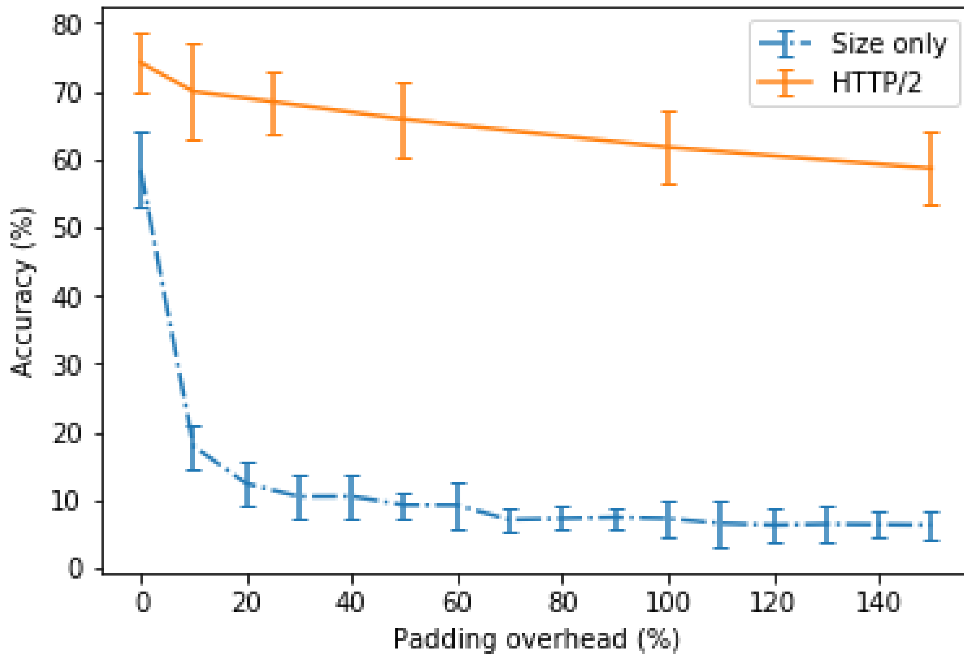


Figure 5: Differences in website fingerprinting when using only size features and using all features

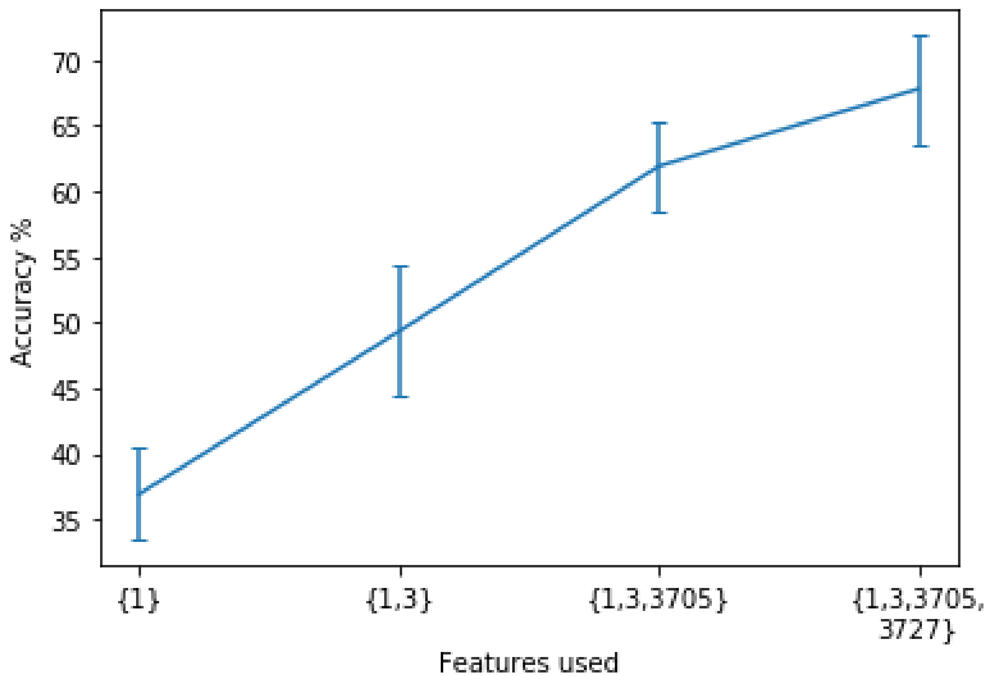


Figure 6: Website fingerprinting accuracy when using the top four features

Furthermore, we did feature analysis to find out which features contribute the most, i.e. have the highest weights in classification. We used recursive feature elimination and greedy search. The top

features used by classification are total number of packets received (1), total page loading time(3), the maximum burst size(3705) and size of the 12th packet in the trace(3727). The accuracy of using these features is shown in the graph below (with average 25% padding).

While features 1,3,3705 are related to total website sizes, the 12th packet contains the first push promise, which contains the name of the file pushed. The graph below shows the distribution of the length of this packet, all ranging from 154 to 156 bytes. Hence, we change the naming in our modeling process, padding the name of all files generated to the same length. After the file name is padded, there is a observable drop in the accuracy of website fingerprinting.

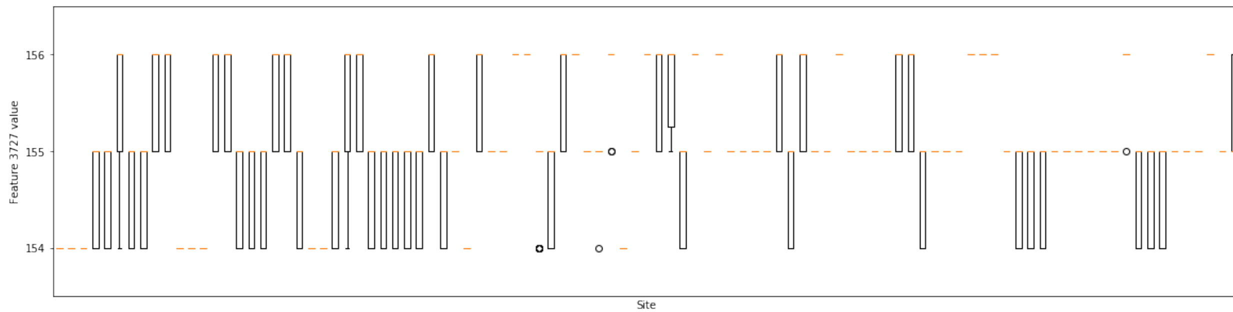


Figure 5: Distribution of sizes of the 12th packet, i.e. feature 3727

5. Conclusion

We measured the influence of HTTP/2 server push and padding on website fingerprinting. Based on test results on our models, we can tell that the combination of these three methods could significantly reduce the accuracy of website fingerprinting. We thus recommend websites with user privacy requirements to implement these methods as countermeasures against website fingerprinting. It is also notable that all these three methods could be implemented with no changes on client side and using existing web server software.

References

1. Alexa: Alexa top 500 global sites. https://www.alexa.com/topsites (2018)
2. Rimmer, V., Preuveneers, D., Juarez, M., Van Goethem, T., Joosen, W.: Automated website fingerprinting through deep learning. In: Network and Distributed System Security Symposium (NDSS). Internet Society (2018)
3. Hayes, J., Danezis, G.: k-fingerprinting: A robust scalable website fingerprinting technique. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 1187–1203. USENIX Association, Austin, TX (2016), https://www.usenix.org/ conference/usenixsecurity16/technical-sessions/presentation/hayes
4. Luo, X., Zhou, P., Chan, E., Lee, W.: HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. In: In Proceedings of the 18th Annual Network & Distributed System Security Symposium (2011)
5. Perry, M.: Experimental Defense for Website Traffic Fingerprinting. https://blog.torproject.org/blog/experimental-defense-website-traffic-fingerprinting (2011)
6. Wright, C.V., Coull, S.E., Monrose, F.: Traffic morphing: An efficient defense against statistical traffic analysis. In: In Proceedings of the 16th Annual Network & Distributed System Security Symposium (2009)
7. Cai, X., Zhang, X., Joshi, B., Johnson, R.: Touching from a distance: Website fingerprinting attacks and defenses. In: In Proceedings of the 19th ACM conference on Computer and communications security. pp. 605–616 (2012)
8. Dyer, K.P., Coull, S.E., Ristenpart, T., Shrimpton, T.: Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. pp. 332–346. Ieee (May 2012)
9. Cai, X., Nithyanand, R., Johnson, R.: CS-BufLO: A congestion sensitive website fingerprinting defense. In: Proceedings of the 13th Workshop on Privacy in the Electronic Society. pp. 121–130. ACM (2014)
10. Cai, X., Nithyanand, R., Wang, T., Johnson, R., Goldberg, I.: A systematic approach to developing and evaluating website fingerprinting defenses. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 227–238. ACM (2014)
11. https://chromedevtools.github.io/devtools-protocol/