Measuring the Impact of HTTP/2 and Server Push on Web Fingerprinting

Weiran Lin, Sanjeev Reddy, and Nikita Borisov
University of Illinois at Urbana-Champaign
{wlin40,streddy2,nikita}@illinois.edu

***Abstract*—The growing deployment of transport- and link-layer encryption mechanisms helps to protect users' security. However, privacy attacks are still possible due to patterns present in the network traffic. Web fingerprinting, in particular, can reveal what web site or page someone is visiting despite encryption. In this paper, we set out to study what impact new web standards—in particular, HTTP/2 and Server Push—have on the ability of adversaries to perform web fingerprinting, as these technologies significantly change network traffic patterns.**

We created web page models of top Alexa sites that captured the dependency structure of the resources on the site. We then captured network traces loading these models using both HTTP/1.1 and HTTP/2 with Server Push, and evaluated their susceptibility to state-of-the-art web fingerprinting attacks. Our results show that HTTP/2 presents a smaller fingerprinting surface for an adversary than HTTP/1.1. Additionally, it admits a simple padding scheme that can further reduce web fingerprinting success. This scheme is competitive with other state-of-the-art defenses, and only presents a small amount of bandwidth overhead.

I. Introduction

A growing fraction of Internet traffic is protected with end-to-end and link-layer encryption. HTTPS is representing an ever-growing fraction of web traffic [8]. Likewise, the usage of virtual private networks has seen significant growth.[1] Although these technologies protect the integrity and confidentiality of network traffic, they leave visible patterns of traffic that may still reveal sensitive information. In particular, *web fingerprinting*[2] can be used to learn what site—or page within a site—someone is visiting, by analyzing traffic generated by their web browser.

There are a number of traffic features that contribute to the success of fingerprinting. An important one that has been identified by prior work is the *burst* feature; the number of packets that are sent in one direction before a packet in the other direction. This feature is dependent on the protocol "dialogue" between the server and client, and it is precisely this dialog flow that is intended to be streamlined by the HTTP/2 server push feature. [2, §8.2] Server push is designed to eliminate the round-trip to request site resources that are referenced by a web page by having the server push the referenced resources preemptively. We therefore investigate the impact of server push on the amount of data that is leaked to fingerprinting.

As the deployment of HTTP/2 is still in its infancy, we perform our evaluation in a synthetic setting. To make it realistic, we collect web page *models* from the 99 most popular sites, as ranked by Alexa [1]. These models capture the dependency structure of the pages, as well as the sizes of each individual site resource, thus capturing the features that influence the network trace of loading a page. We then serve these models over both HTTP/1.1 and HTTP/2, and use state-of-the-art web fingerprinting techniques [11], [27] to evaluate each protocol's susceptibility to fingerprinting.

We found that there was a notable decrease in fingerprinting accuracy—from 80% to 74%—when HTTP/2 with server push was used, although fingerprinting was still possible. We identified three sources of information that contributed to this: the length of the site name/URL, the names of the page dependencies that are pushed, and the total size of the page. We therefore investigated whether padding could be used to reduce the information available from these sources. With an average padding overhead of 25%, fingerprinting accuracy falls to 64%. We note that these figures are comparable to existing padding protocols in the literature [4], [5], [14] but can be implemented without changing the web transport and application layer protocols.

II. Background

In this section we review some background on web page structure, HTTP/2 and server push, and web fingerprinting.

A. Web page structure

A common web page is comprised of a large number of resources, such as images, scripts, stylesheets, and frames. When a user navigates to a URL, the browser first requests and downloads the main HTML page. It then parses it to find references to embedded resources and makes new requests for them. Each resource may recursively reference other resources, causing even more requests to be generated. Figure 1 shows the various components of an example site, arranged in a dependency graph. According to the HTTP Archive, in September 2018 a median (desktop) web page consisted of 75 site resources and correspondingly required 75 requests.[3]

The complex structure of pages has the potential to significantly slow down page load times. In addition to the large total size of all the objects (≈ 1.5 MB, according to the HTTP Archive), each individual request adds latency: each additional request adds a round-trip latency, compounded by any processing time the browser needs to parse the response and

[1] https://www.vpnmentor.com/blog/vpn-use-data-privacy-stats/

[2] We use the term *web fingerprinting* in place of the popular *website fingerprinting* as the underlying techniques apply both within and across sites.

[3] https://httparchive.org/reports/state-of-the-web, fetched Oct 17, 2018.

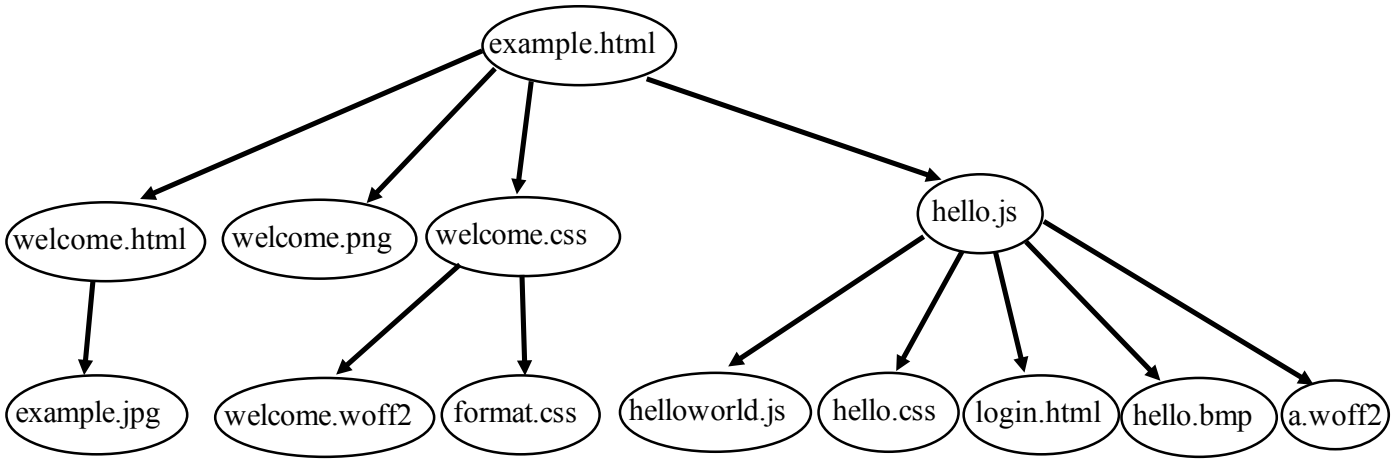

Fig. 1. The structure of an example web page. The site resources include HTML documents (.html), stylesheets (.css), images (.bmp, .png, .jpg), scripts (.js), and fonts (.woff2). The arrows represent the dependency structure of the page; each resource points to other resources that it references/causes to be loaded.

identify new requests. Browsers and web servers use a number of techniques to speed up this process, including connection reuse, simultaneous parallel connections, incremental resource parsing, resource scheduling and prioritization, etc. Note that HTTP request pipelining [9, §6.3.2] was designed to reduce round-trip latency associated with site dependency requests; however, it has since been disabled by major web browsers due to head-of-line blocking and other issues.[4] These partially mitigate, but do not eliminate, the performance overhead created by multiple site resources.

a) HTTP/2: The SPDY protocol [10] and its successor, HTTP/2 [2], were designed in part to address these issues. They define a number of features, such as multiplexing multiple objects in one stream, the ability to asynchronously schedule requests (similar to pipelining), as well as prioritize and cancel them (to avoid head-of-line blocking), and server push. The latter is motivated by the fact that a server can predict that a user loading example.html in Figure 1 will also want to load welcome.html, welcome.png, etc. The server can therefore proactively start sending these objects to the client without waiting for an explicit request.

Note that server push does not always result in faster page load times; Wang et al. found that server push might introduce extra overhead by pushing unneeded (e.g., cached) resources, depending on push configurations as well as client latencies [30]. Our present goal is to understand the *privacy* implications of the push mechanism to help inform its future deployment.

B. Web Fingerprinting

It has long been known that object sizes, not protected by HTTPS, can reveal sensitive information [26]. This gave rise to web fingerprinting, where someone who observes encrypted web traffic uses patterns of traffic, such as packet sizes, counts, directions, and timings, to discern what web site or web page a user is visiting. Web fingerprinting applies in two contexts. First, when using an anonymous communication tool, such as Tor [6], a local observer can try to use web fingerprinting to infer the destination website [17], [23]. Alternately, someone who is observing a direct HTTPS connection will know the destination web *site*, but can use fingerprinting to learn which *page* within a site is being visited [19].

The techniques for fingerprinting are similar in both cases: consider a stream of packets with their times, sizes, and directions, and use machine learning to classify them into one of several potential destinations. The major difference is that in Tor, traffic is sent over equal-sized cells, and therefore packet sizes carry no information. Classification can be done in a closed-world setting, where a packet trace is assumed to be visiting one of a *known* set of web destinations, or open-world, which adds the possibility that a visit is to a previously unknown site. A full review of web fingerprinting techniques is beyond the scope of this paper, but we will discuss the particular techniques we use in section III-C.

A significant amount of work has gone into engineering the features that are used for web classification. Recent work has noted the importance of packets *bursts*—a sequence of packets in the same direction—in fingerprinting [27], as they are representative of the request-response pattern of a site and can therefore help recover the site structure; some recent defenses focus specifically on burst sizes [29]. We note that HTTP/2 with server push significantly changes the burst structure and thus has the potential to impact fingerprinting success.

We should note that many assumptions behind web fingerprinting research have been questioned [13] as being unrealistic. Recent work, however, tries to address some of these criticisms [28]; furthermore there are contexts, such as onion (hidden) services in Tor [21] where fingerprinting is particularly dangerous.

a) Defenses.: Initial work on defenses against web fingerprinting focused on ad hoc approaches that modify some of the aggregate features used in early fingerprinting

[4] See, for example, https://www.chromium.org/developers/design-documents/network-stack/http-pipelining

work [18], [24], [31]; however, these have been shown to be largely ineffective by more sophisticated machine learning techniques [3], [7]. Recent exploration has focused on variants of *constant rate* defenses [4], [5]—where the server and client communicate with each other using a constant-rate stream—as this approach has a provable bound on the amount of information available to a web fingerprinting attacker, i.e., the total amount of data transferred in each direction (plus the rate, in cases when it is adjustable).

In fact, an idealized version of an HTTP/2 page load with server push resembles a constant-rate download, since a single request from the client is satisfied by a download of all the resources comprising a page, and as such, should present a limited amount of information for fingerprinting. Therefore, our goal is to evaluate this high-level intuition using actual HTTP/2 implementations in browsers and web servers.

III. Experimental Set Up

To understand how HTTP/2 and server push affect fingerprinting, we wanted to do a head-to-head comparison between HTTP/1.1 and HTTP/2 with server push. Since the deployment of HTTP/2 is still sparse and the use of sever push is limited, we created a synthetic experimental environment in which we served the same web pages using both protocols and collected traces for fingerprinting.

To ensure that our synthetic traces were realistic, we modeled the structure of our test web pages on the structure of actual popular web sites. We next describe our approach for creating and serving these models. Finally, we discuss the trace collection and fingerprinting techniques used.

A. Web Page Models

As discussed in section II-A, a web page is composed of a number of different types of resources. We wanted our model pages to accurately mimic the structure of real web pages so that they would be representative of real-world sites. To do so, we first performed a web crawl of the the 100 most popular websites as listed by Alexa [1] using a headless version of Chromium. We used the Chrome DevTools protocol[5] to intercept network events such as requestWillBeSent, responseReceived, dataReceived, and loadingFinished to keep track of the requests and responses that were sent and received by the browser during the loading of the page. From the metadata in these requests, we can extract the type of the resource that is being requested (document (i.e., HTML file), stylesheet, script, image, or font) as well as its size. Additionally, we are able to learn what previously loaded resource is responsible for loading this resource—e.g., a script might load a font, a stylesheet might load an image, etc.—through the initiator information in the responseReceived event.

We then create a web page *model* that replicates the structure attributes captured during our initial load. We create a main HTML page, called index.html, that includes all the resources directly loaded from the main page. For example, in Figure 1, the example.html page loads an image, welcome.png, another html page, welcome.html, a stylesheet, welcome.css, and a script, hello.js. We would therefore create resources of the corresponding type and load them from the index.html file, as shown in fig. 2. We then add a comment to pad the file to the same size as the main file of the original site.

```
<head>
<link rel="stylesheet" href="sheet0.css">
<script src="script0.js"></script>
<body>
<img src="img0.gif">
<iframe src="index1.html"></iframe>
<!—AAAAA [...] AAA—>
</body>
```

Fig. 2. The top-level HTML file in a web page model, which loads a stylesheet (sheet0.css), a script (script1.js), an image (img0.gif), and another HTML file in an iframe (index1.html).

Likewise, we include directives in the other objects to load their dependencies. For example, script0.js, which models hello.js in fig. 1, will include Javascript commands to load a font, another script, a stylesheet, an image, and a font; it will then be padded to the correct size using comments. For images, we create a 1×1 pixel GIF file with a comment section that pads the object to the correct size (we found that serving an invalid image file would cause the browser to abort the load). The dependency structure of the resulting model is shown in fig. 3

We have to use special handling for fonts, as Chromium does not load a font unless it is actually used to render some text. We therefore add a text element styled to use the loaded font inside an HTML file. We also discovered in our testing that Chromium reports that the font load is initiated by the HTML file where it is used, even if the font is specified in a separate stylesheet (CSS file). This means that our model will miss a dependency of a font on a stylesheet; in fig. 3, font0.woff, which models welcome.woff2 in fig. 1, depends only on the main page and not the stylesheet sheet0.css, which models welcome.css. We found a similar problem when a stylesheet loads a second sheet using the @import directive, resulting in another discrepancy for the dependency structure of sheet1.css in fig. 3, which models format.css in fig. 1. Since our goal is to create representative web pages, rather than perfectly model each site, we felt this was an acceptable deviation.

To validate our models, we loaded the model web pages with headless Chromium and used the DevTools protocol to note the request and response sizes, types, and order; we found that the models matched the original web page behavior.

B. Trace Collection

To collect traces used for fingerprinting, we used the Caddy web server[6] to serve the model web pages and headless Chromium to load them. The Caddy server was configured to use HTTPS and HTTP/1.1 or HTTP/2, depending on the test (we note that some of the modeled websites were served over HTTP, but we used HTTPS, motivated by the trend towards

[5]https://chromedevtools.github.io/devtools-protocol/

[6]https://caddyserver.com/

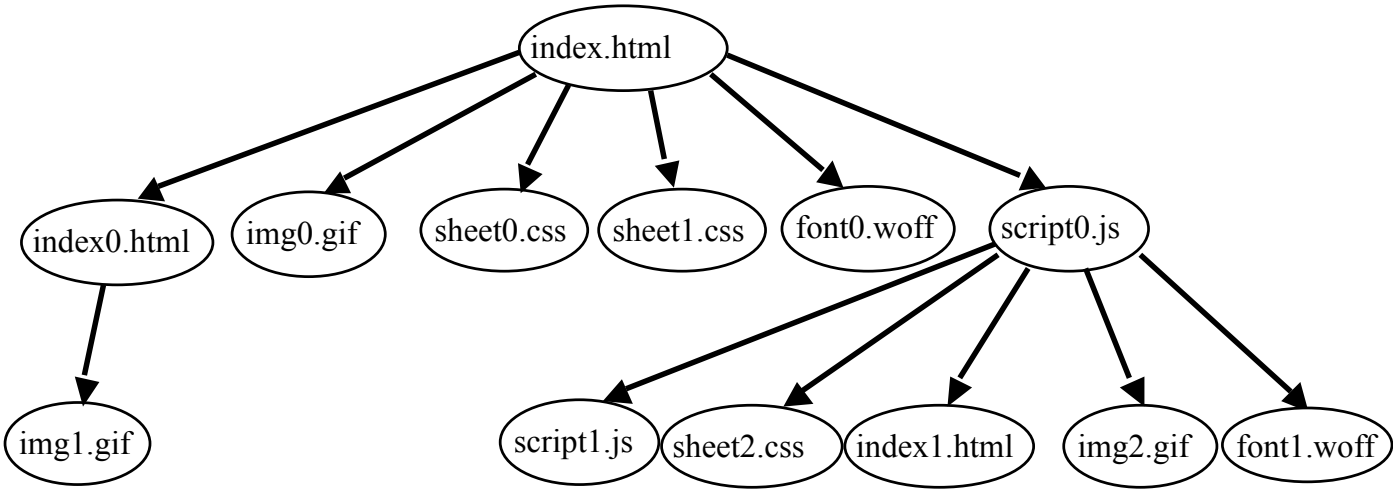

Fig. 3. A model of the example web page from fig. 1. The file types and sizes all match the original page. The dependency structure closely mimics the original, but note that there is a deviation in how the font0.woff and sheet1.css are loaded.

greater HTTPS deployment). The server and browser were run in separate Docker containers connected by an isolated network. The browser container also ran tcpdump to produce a PCAP file that served as our web trace. We processed the PCAP to filter out DNS packets, omit any TCP packets that carry no data, and recorded only the sizes, directions, and timing of each packet. These attributes were then used as input for fingerprinting.

As points of comparison, we also collected packet traces of headless Chromium loading the original sites.

C. Classification

To perform web fingerprinting, we use features developed by Wang et al. [27] for their k-nearest neighbor classifier. Each packet stream is converted to a vector of 3766 features capturing distributions of packet counts, sizes, bursts, etc. However, Hayes and Danezis demonstrate that a random forest classifier using the same set of features achieves a higher classification accuracy [11], hence we use a random forest classifier in our work. Later work has explored using different features [22] as well as deep learning [25]; however, we use the random forest technique because it allows us to perform an analysis of feature importance.

In our classification, we first perform recursive feature elimination with a step size of 100 to prune the 3766 features down to the 100 most relevant ones. We also compute the accuracy using all 3766 features, but in our experience the classifier performs better on the reduced feature space. In our experiments, we use closed-world classification, as it generally has higher performance and thus better highlights the information disparities resulting from protocol changes.

IV. Experiments and Results

We next discuss the experiments we performed and their results. A summary of the experimental resuls can be found in table I.

TABLE I. Summary of results.

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

A. Sites and Models

We collected 30 traces each from the top 100 web sites as ranked by Alexa [1]. One of the websites was down during our collection, thus we were left with $99 \times 30 = 2970$ traces for our experiment. We note that our random forest classifier, as described in section III-C, performs very well on these traces: it has a classification accuracy of $91.3\% \pm 4.3\%$.[7]

We use each of the 30 traces to create a model. We note that, as many of the web pages we load are dynamic, different traces can—and often do—result in different page models. For example, fig. 4 shows that many sites have significant variance in the total size of all the objects that are downloaded. This creates significant challenges for fingerprinting. For example, when we collected 30 traces from a single model per site (i.e., using 1×99 models), served over HTTP/1.1, our classifier was able to achieve $99.9\% \pm 0.5\%$ accuracy. On the other hand, using a different model for each trace and each site (i.e., using 30×99 models), the accuracy fell to $80.2\% \pm 3.8\%$.

Why is the classification accuracy of the synthetic sites lower than the real sites? The biggest difference is that in our experiment, all resources are served from a single web server, whereas real sites typically access several origins at a time (image servers, ad servers, etc.), using separate connections with potentially varying latencies. We do note that there is a trend towards serving a large fraction of a site's resources via content distribution networks (CDNs) [15], [16], and

[7] All of the accuracy numbers are obtained using 10-fold cross-validation; we use two standard deviations of the accuracy scores across the folds to approximate a 95% confidence interval.

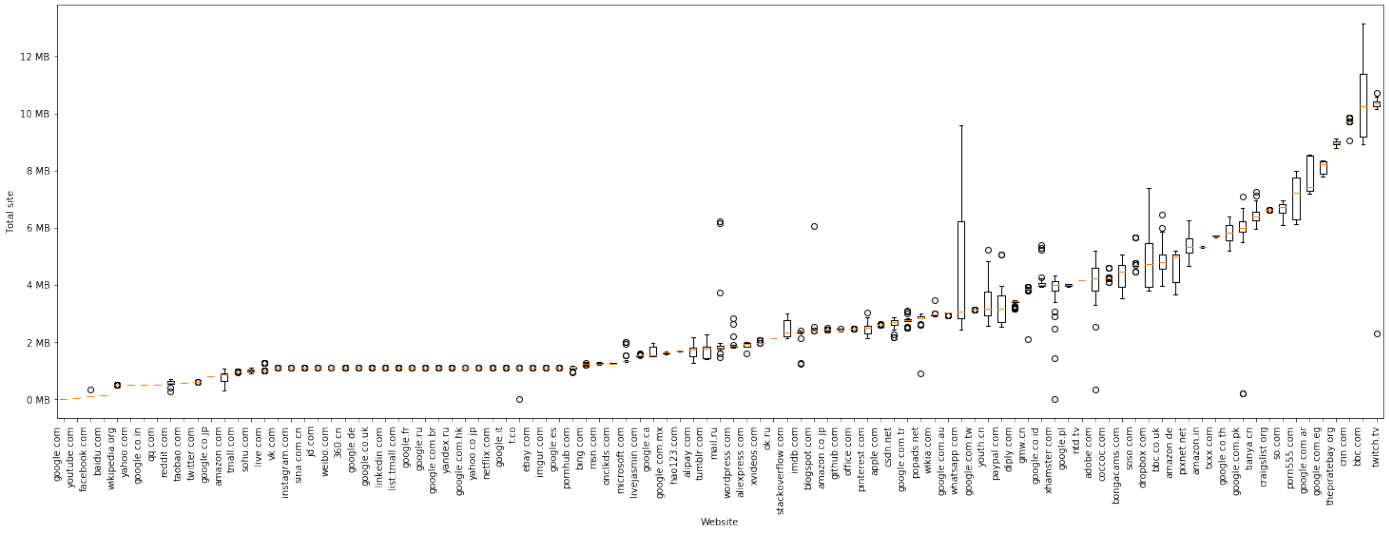

Fig. 4. The distribution of total page size across 30 traces for each of the 99 sites in our data set. Each distribution is shown as a box plot, where the dashed line represents the median, boxes capture the interquartile range, the whiskers represent the range, and individual points represent absent outliers.

CDNs are developing technology, such as the HTTP/2 origin frame [20] to allow serving more components via a single server connection.

A second factor is that we change the site names to be of a uniform length. We discovered that the size of the ClientHello packet is directly proportional to the length of the site name, due to the name being embedded inside a Server Name Indication (SNI) extension. In our data set, the site name lengths have 3.2 bits of entropy. By padding the names to the same length, we are able to eliminate this extra source of fingerprinting and focus only on the impact of the protocol. We note that this result also suggests that the proposed SNI encryption [12] should consider padding the SNI as well.

B. HTTP/2 Server Push

To evaluate the impact of HTTP/2 server push on fingerprinting, we configured Caddy to use HTTP/2 and to push all of the site resources in a model as soon as the top-level HTML was requested. We then collected 30×99 traces using Chromium. The corresponding accuracy was $74.2\% \pm 4.3\%$. Therefore, we conclude that HTTP/2 with server push is able to reduce the amount of information available to fingerprinting.

HTTP/2, of course, makes a number of changes in addition to sever push. To understand whether those changes have an impact, we conduct a second set of experiments that use HTTP/2 but not server push. The resulting accuracy is $80.4\% \pm 2.5\%$, or virtually identical to the HTTP/1.1 scenario.

C. Padding

As can be observed in fig. 4, there is some variability in the total size of the download between different instances of the same web page. However, some pages are significantly less variable; 4 out of 99 sites have exactly the same total size across all page loads. These sites are therefore easy to fingerprint based on total size alone. Moreover, there are large disparities in the range of total site sizes: the smallest site transfers less than 500 KB, whereas the largest transfers over 10 MB. In fact, if we train a random forest classifier on a single feature—the total number of bytes downloaded—we get an accuracy of $58.1\% \pm 2.8\%$.

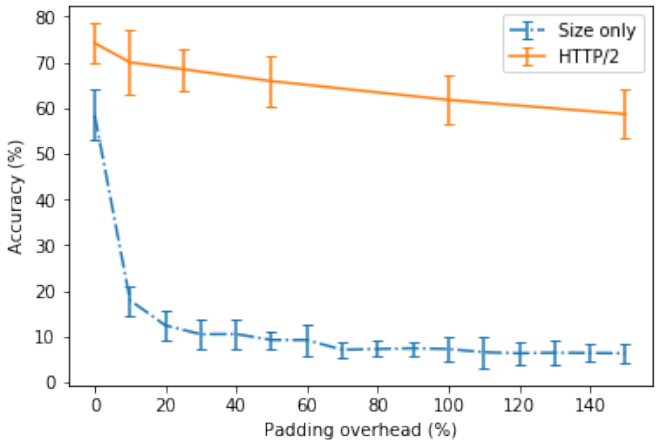

Fig. 5. Results of adding padding to pages served with HTTP/2 server push, compared to the classification results that only use the total download size.

We therefore consider adding padding to reduce the amount of information that can be learned from the site size. We use a simple padding scheme: if a site transfers k bytes, we pick x uniformly at random from the range $[0, \alpha]$ and add $k \cdot x$ bytes of padding, resulting in an average of $k\alpha/2$ extra bandwidth overhead. To implement this in our test setting, we add an extra Javascript file that contains $k \cdot x$ bytes of comments to the main HTML page of each model. In practice, to avoid the parsing overhead, an empty image could be used. In addition, the server could possibly be configured to push a padding file not referenced by any object in the dependency tree. The results of the padding experiments are shown in fig. 5 for various average padding sizes.

We see that even a small amount of padding results in a reduced accuracy: at 10% average padding, the accuracy is

$70.0\% \pm 7.1\%$, and with 25%, the accuracy is $68.5\% \pm 4.6\%$. For comparison, we experiment with serving the padded web page over HTTP/1.1. The resulting accuracy (at 10% padding) is $79.1\% \pm 4.3\%$. The web page structure available for fingerprinting ensures that padding in HTTP/1.1 has little to no effect.

We also try training a classifier on the total padded size alone. We note that the accuracy is dramatically lower, suggesting that other factors are contributing to fingerprinting success.

D. Feature Analysis

To understand what features impact the fingerprinting accuracy, we perform a feature importance analysis. We first use recursive feature elimination, similar to section III-C, to further reduce the 100 features down to 20, eliminating one feature at each step. We then perform a greedy search to find the best features among those 20. We start by using each individual feature to classify the data set and using 10-fold cross validation to compute the classification accuracy μ_i and standard deviation σ_i.

We then pick the best performing feature $i^* = \arg\max_i \mu_i$ and pair it with all remaining features to calculate μ_{i^*+j} to pick the next best feature. We let S_i denote the set of best i features picked in this way; i.e., $S_1 = \{i^*\}, S_2 = \{i^*, j^*\}$. We continue the algorithm until the improvement from adding a new feature is less than one standard deviation, i.e., $\mu_{S_k} - \sigma_{S_k} < \mu_{S_{k+1}}$.

Since some feature choices result in similar classification accuracies, instead of picking the unique set S_i at each step, we also consider any sets S_i' such that $\mu_{S_i} - \sigma_{S_i} < \mu_{S_i'} + \sigma_{S_i'}$. This creates a small combinatorial increase in the number of sets considered, but the approach remains largely greedy without considering all $\binom{20}{i}$ combinations.

When we apply the analysis to the previous classifier that uses HTTP/2 with no padding, the four features that are selected are (in order):

- 1: total number of packets received
- 3: total page load time
- 3705: the maximum burst size
- 3727: size of the 12th packet in the trace

The accuracy of using these features is shown in fig. 6. We note that the first three features are strongly correlated to the total size of data downloaded. The last one is a little unexpected; upon inspection, this packet in the trace generally ranges from 154 to 156 bytes. However, the distribution is different for different sites, as shown in fig. 7.

Upon further inspection, this packet contains the first push promise, which includes the file name of the first file pushed by the Caddy server. We note that the name of the first pushed file ranges in length from 8 characters (img0.gif) to 10 characters (sheet0.css), depending on the file type, which explains the difference in the lengths.

We therefore create a new experiment where all pushed files have names that are padded to the same length. With a 25% average overhead, the classification performance drops to $63.5\% \pm 8.0\%$. We note that this is still much higher than the accuracy resulting from classifying on total sizes alone—$11.0\% \pm 1.4\%$. Thus there is still a difference between HTTP/2 with server push and the idealized constant-rate download of the entire site. We will investigate the sources of this difference in future work.

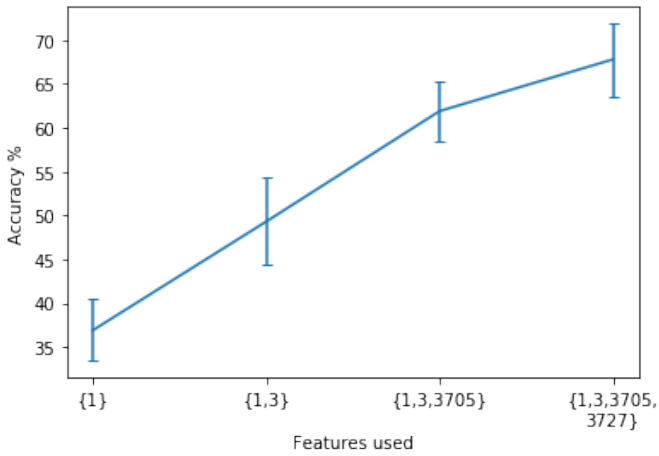

Fig. 6. Feature selection for HTTP/2 with 25% padding.

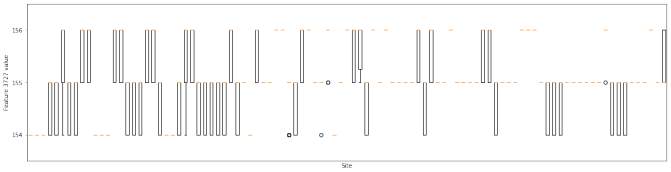

Fig. 7. Distribution of values for feature 3727: size of the 12th packet

V. Conclusion

We examined the impact of HTTP/2 and server push on web fingerprinting accuracy by creating a synthetic set of web pages that model real-world web page structure. We found that HTTP/2 with server push creates a significant reduction in fingerprinting accuracy. We also found that adding a small to moderate amount of padding further reduces fingerprinting accuracy.

We can therefore issue a set of recommendations to websites that wish to reduce their users' susceptibility to fingerprinting:

1) Serve as many site components as possible from a single server
2) Use HTTP/2 and server push on that server
3) Normalize/pad the length of a site name (SNI) and the request URLs of all site resources
4) Add a small-to-moderate amount of randomized padding in the form of an unused, pushed web page component

We note that all of these steps can be implemented using existing tools, while allowing users to browse with off-the-shelf browsers that support the HTTP/2 protocol.

References

[1] Alexa, "Alexa top 500 global sites," https://www.alexa.com/topsites, 2018.

[2] M. Belshe, R. Peon, and M. Thomson, "Hypertext transfer protocol version 2 (HTTP/2)," RFC7540, May 2015. [Online]. Available: https://httpwg.org/specs/rfc7540.html

[3] X. Cai, X. Zhang, B. Joshi, and R. Johnson, "Touching from a distance: Website fingerprinting attacks and defenses," in *In Proceedings of the 19th ACM conference on Computer and communications security*, 2012, pp. 605–616.

[4] X. Cai, R. Nithyanand, and R. Johnson, "CS-BufLO: A congestion sensitive website fingerprinting defense," in *Proceedings of the 13th Workshop on Privacy in the Electronic Society*. ACM, 2014, pp. 121–130.

[5] X. Cai, R. Nithyanand, T. Wang, R. Johnson, and I. Goldberg, "A systematic approach to developing and evaluating website fingerprinting defenses," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 227–238.

[6] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *In Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 303–320.

[7] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, "Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. Ieee, May 2012, pp. 332–346.

[8] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz, "Measuring HTTPS adoption on the Web," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1323–1338. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/felt

[9] R. Fielding and J. Reschke, "Hypertext transfer protocol (HTTP/1.1): Message syntax and routing," RFC7230, Jun. 2014. [Online]. Available: https://httpwg.org/specs/rfc7230.html

[10] Google, "SPDY: An experimental protocol for a faster web," https://www.chromium.org/spdy/spdy-whitepaper, 2010.

[11] J. Hayes and G. Danezis, "k-fingerprinting: A robust scalable website fingerprinting technique," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 1187–1203. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/hayes

[12] C. Huitema and E. Rescorla, "Issues and Requirements for SNI Encryption in TLS," Internet Engineering Task Force, Internet-Draft draft-ietf-tls-sni-encryption-03, May 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-tls-sni-encryption-03

[13] M. Juarez, S. Afroz, G. Acar, C. Diaz, and R. Greenstadt, "A critical analysis of website fingerprinting attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[14] M. Juarez, M. Imani, M. Perry, C. Diaz, and M. Wright, "Toward an efficient website fingerprinting defense," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 27–46.

[15] A. Kashaf, C. Zarate, H. Wang, Y. Agarwal, and V. Sekar, "Oh, what a fragile web we weave: Third-party service dependencies in modern webservices and implications," 2018.

[16] D. Kumar, Z. Ma, Z. Durumeric, A. Mirian, J. Mason, J. A. Halderman, and M. Bailey, "Security challenges in an increasingly tangled web," in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 677–684.

[17] M. Liberatore and B. N. Levine, "Inferring the source of encrypted HTTP connections," in *In Proceedings of the 13th ACM conference on Computer and Communications Security*. New York, New York, USA: ACM Press, 2006, p. 255.

[18] X. Luo, P. Zhou, E. Chan, and W. Lee, "HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows." in *In Proceedings of the 18th Annual Network & Distributed System Security Symposium*, 2011.

[19] B. Miller, L. Huang, A. D. Joseph, and J. D. Tygar, "I know why you went to the clinic: Risks and realization of https traffic analysis," in *Privacy Enhancing Technologies Symposium*, 2014.

[20] M. Nottingham and E. Nygren, "The ORIGIN HTTP/2 frame," RFC8336, IETF, Mar. 2018. [Online]. Available: https://tools.ietf.org/html/rfc8336

[21] R. Overdorf, M. Juarez, G. Acar, R. Greenstadt, and C. Diaz, "How unique is your. onion?: An analysis of the fingerprintability of tor onion services," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2021–2036.

[22] A. Panchenko, F. Lanze, J. Pennekamp, T. Engel, A. Zinnen, M. Henze, and K. Wehrle, "Website fingerprinting at Internet scale," in *Network and Distributed System Security Symposium*. Internet Society, 2016.

[23] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website fingerprinting in onion routing based anonymization networks," in *In Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*. New York, New York, USA: ACM Press, 2011, p. 103.

[24] M. Perry, "Experimental Defense for Website Traffic Fingerprinting," https://blog.torproject.org/blog/experimental-defense-website-traffic-fingerprinting, 2011.

[25] V. Rimmer, D. Preuveneers, M. Juarez, T. Van Goethem, and W. Joosen, "Automated website fingerprinting through deep learning," in *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2018.

[26] D. Wagner and B. Schneier, "Analysis of the SSL 3.0 protocol," in *USENIX Workshop on Electronic Commerce*, 1996.

[27] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg, "Effective attacks and provable defenses for website fingerprinting," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 143–157. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/wang_tao

[28] T. Wang and I. Goldberg, "On realistically attacking tor with website fingerprinting," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 4, pp. 21–36, 2016.

[29] ——, "Walkie-talkie: An efficient defense against passive website fingerprinting attacks," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1375–1390. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-tao

[30] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "How speedy is SPDY?" in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 387–399. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/wang

[31] C. V. Wright, S. E. Coull, and F. Monrose, "Traffic morphing: An efficient defense against statistical traffic analysis," in *In Proceedings of the 16th Annual Network & Distributed System Security Symposium*, 2009.