MEASURING THE IMPACT OF SITE CONFIGURATIONS ON SITE FINGERPRINTING OVER THE WEB AND TOR

BY

SANJEEV REDDY

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Professor Nikita Borisov

ABSTRACT

As security and privacy on the web become topics of significant concern, there have been increased efforts to expedite the deployment of encryption-based transport- and link-layer protection mechanisms such as HTTPS. Although encryption protects the data being transmitted between a client and a server, site visits generate unique traffic patterns due to contents of the site and the manner in which the server responds to user requests for site resources. These patterns can be learned by an adversary, and then be used to predict which site (or web page within a site) a user is visiting—a technique known as *web fingerprinting*. Web fingerprinting allows an adversary to compromise user privacy even in the presence of encryption mechanisms or anonymity systems, such as the Tor network.

In this thesis, we examine how changes to a site's configuration (i.e., the size of the site, site content, hosting strategies, etc.) can influence an adversary's ability to successfully fingerprint a user's visit to a site over the web and Tor. We pay particular attention to the impact of HTTP/2 and Server Push—new web standards which significantly change network traffic patterns by altering the order in which site resources are served. Additionally, we experiment with padding site sizes, renaming site resources, and hosting sites from both single and multiple servers in order to observe the effect of each of these changes on fingerprinting accuracy.

In order to collect traces from sites that reflect our experimental changes, we create *models* of real-world sites and onion services that capture the resource dependency structures of the original sites. We then modify these models to reflect our desired configuration changes and serve them via HTTP/1.1 and HTTP/2 with server push. We collect traces of visits to these models conducted over the web, as well as the Tor network, and evaluate the performance of state-of-the-art fingerprinting classifiers on both sets of traces. We find that HTTP/2 with server push can successfully reduce fingerprinting accuracy when compared to HTTP/1.1, and that real-world sites visited over the web benefit from single-server hosting, site padding, and constant-length Huffman-encoded resource names. We also find that HTTP/2 with server push reduces the fingerprintability of regular sites and onion services accessed over the Tor network, but inconsistencies in our data prevent us from drawing any conclusions regarding the efficacy of site padding, resource renaming, and single- vs. multi-server hosting when fingerprinting Tor traffic. We suggest future work that should help gather more conclusive results for our Tor experiments.

To my family, for their love and support.
To my friends, for their good company.
To those I lost, for the memories we made.

ACKNOWLEDGMENTS

I want to begin by extending my deepest thanks to my advisor, Professor Nikita Borisov. His guidance, support, and encouragement over the past two years have been instrumental in my success as a graduate student, and I am extremely grateful for the opportunities he has given me as a member of his Hatswitch research group. He has helped me explore my research interests both within and outside of the university setting, and I am thankful for his patience and understanding throughout the process. None of this work would have been possible without him.

I would also like to thank Weiran Lin and Paul Murley for their help in conducting work that was essential to this thesis. I had the pleasure of collaborating with Weiran on the first half of the work documented in this thesis, and I would like to thank him for continuing to answer my questions and remaining a reliable point of contact since. I am also grateful to Paul for answering my questions about the Mida trace collector and expediently implementing certain features that were key to this research.

Next, I would like to thank my peers and the faculty of Security and Privacy Research at Illinois (SPRAI); I thoroughly enjoyed meeting all of them, building working and personal relationships, and delving deeper into the world of security and privacy research. Special thanks to my group mates at the Hatswitch research group: Chester Cheng, Muhammad Haris Mughees, Simran Patil, Qingrong Chen, and Sze Chuen Tan. They were always a pleasure to spend time with, and I am grateful for their friendship and company throughout the past two years.

I am also thankful for the many wonderful professors I had the pleasure of learning from during my time at Illinois. They demonstrated nothing but passion for their fields of research, and being a part of their courses was an enriching experience.

My most heartfelt thanks to my friends from back home and those I made here in Urbana-Champaign. They have given me a sense of belonging and much-needed reprieves from the toil of daily academic work. They never failed to put a smile on my face, and I am thankful for their friendship.

Finally, no words can express my gratitude for my family. Their unconditional love and support have kept me going through thick and thin, and I would not be where I am today without them. To Mom, Dad, and Pramith: lots of love, and thanks for everything.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The protection of user privacy is a major point of concern in the modern web. Protocols such as HTTPS have been seeing widespread deployment as site developers, researchers, and browser vendors attempt to curtail adversarial efforts to tamper with or intercept user data [1]. Virtual private networks are also becoming increasingly popular as a means for users to practice safer browsing[1]. Security through encryption and cryptographic means are now the basis of most modern web protocols and security best-practices—data is encrypted to prevent an unwanted adversary from reading it, and digital signatures have become the de facto standard for proving one's identity on the web.

Anonymity systems—such as the Tor network—have also arisen as an alternative browsing approach for users concerned about remaining anonymous while browsing the web. Tor protects user anonymity by ensuring that no hop along a communication pathway is made aware of the original sender's identity, or the final destination of the communication. Because of its ability to protect the identities of clients, the Tor network has become a crucial tool for proponents of the free-web and users attempting to circumvent state-level censors. It is also host to numerous hidden *onion services*, which are not reachable via the regular Internet and often host sensitive content that may be deemed objectionable by a censoring body.

Link- and transport-layer encryption technologies—such as HTTPS and TLS—are useful for protecting the integrity and confidentiality of network traffic, and anonymity systems—such as Tor—are designed to protect the identity of their users from entities within the network. However, despite the defenses they provide, both mechanisms cannot protect against an inherent aspect of web browsing across any medium: visiting a specific site generates a unique pattern of network traffic that can be used to reveal sensitive information about a user's browsing. This technique—known as *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.

The success of a web fingerprinting attack relies on the fact that each web site being visited has a unique page structure. In order for a browser to display the contents of a page, it must perform a complex set of interactions with one or more web servers to download all of the resources used to specify a site's appearance, structure, and behavior. These client-server(s) interactions create unique patterns of traffic that are identifiable despite the use of encryption and various masking strategies [2]. Factors such as packet directions, inter-

[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.

packet latencies, and the number and size of downloaded resources all contribute towards constructing a discernible profile for a given web page. New protocols—such as HTTP/2 and its Server Push feature [3]—streamline the client-server interactions by introducing optimizations over HTTP/1.1 and changing the resulting traffic patterns. Although these changes are motivated by performance, we aim to investigate what impact they have on web fingerprinting performance. We are particularly interested in the server push mechanism, as it fundamentally alters the request-response pattern for downloading site resources—a change that should reflect quite heavily in the network trace of a site visit. In this study, we examine how non-intrusive changes to a web page's content and hosting strategies can change the profile of client-server traffic patterns, and in turn, impact the success of site fingerprinting over the web and Tor.

Deployment of the HTTP/2 protocol has been gaining speed, but the use of server push in the wild is still relatively rare.[3]. We therefore perform our experiments in a synthetic setting in which we can serve modern websites over HTTP/2 with server push enabled, regardless of whether the original sites support the protocol. To make our simulated setting as realistic as possible, we collect web page *models* from the 99 most popular sites, as ranked by Alexa [4], as well as 20 real-world active onion services. These models capture the dependency structure of the pages, as well as the sizes and types of each individual site resource, thus capturing the core features that influence the network trace of loading a page. We serve these models over the web and Tor, and collect network traces of the model sites using both HTTP/1.1 and HTTP/2 with server push. In the Tor setting, we also evaluate whether hosting site resources on a single server provides any benefit over hosting resources across multiple web servers per site. We use state-of-the-art web fingerprinting techniques [5, 6, 7] to evaluate each model-configuration's susceptibility to fingerprinting.

Throughout the remainder of this thesis, we make the following contributions:

1. **Measure HTTP/2's impact on fingerprinting over the web:** We find that there is a notable drop in classifier accuracy when a site is served via HTTP/2 with server push enabled. Without any other changes to the site's configuration, HTTP/2 with server push reduces fingerprinting accuracy from 80% to 74%.

2. **Site padding and constant-length Huffman-encoded resource names:** We identify three sources of information that contribute to fingerprintability: the length of the site name/URL, the names of the resources that are pushed, and the total size of the page. We investigate the option of padding site models to mitigate the information

[3] `https://http2.netray.io/stats.html`

learned through these sources, and find that with an average padding overhead of 25% of the site's original size, fingerprinting accuracy falls to 68%. In order to address information learned through resource names, we rename all site resources to have names of the same Huffman-encoded length. This further lowers fingerprinting accuracy to 62%. We note that these figures are comparable to existing padding protocols in the literature [8, 9, 10] but can be implemented without changing the web transport and application layer protocols.

3. **Measure fingerprintability of padded, HTTP/2-enabled models over Tor:** We extend our findings from fingerprinting over the web to fingerprinting over the Tor network. We host both regular sites and onion services as models and access them over the Tor network under various site configurations (e.g. padded vs. unpadded, HTTP/1.1 vs HTTP/2 with server push). We find that HTTP/2 with server push seems to marginally lower fingerprinting accuracy over Tor, though not to the same degree as it does over the regular web. We get mixed results regarding the impact of site padding on Tor fingerprinting.

4. **Single- vs. multi-server resource hosting:** We note that real-world sites often host their resources on multiple web servers. We test whether this has any bearing on site fingerprintability by hosting each Tor model from a single server, as well as multiple servers. We find that hosting all site resources from a single server reduces Tor fingerprinting accuracy for regular sites. However, onion services generally seem to be *more* fingerprintable when served from a single server.

We provide a background of web page structure, HTTP/2, Tor, and web fingerprinting in Chapter 2. We then detail our experiments with measuring the impact of HTTP/2 on web fingerprinting in Chapter 3. Chapter 4 discusses takeaways from our initial experiments over the web, and how we might apply them to our next experiments over Tor. Our experiments with fingerprinting regular sites and onion services over Tor are detailed in Chapter 5, along with a discussion of the results and how they can be improved moving forward. Finally, we conclude in Chapter 6.

CHAPTER 2: BACKGROUND AND RELATED WORK

In this section, we discuss various aspects of browsing on the modern web, including web page structure, HTTP/2, and the Tor network. We then describe web fingerprinting and how the aforementioned mechanisms can impact the success of an adversary attempting to classify network traffic.

2.1 WEB PAGE STRUCTURE

Modern web pages are content-rich amalgamations of numerous *resources* that are all rendered together in a browser to provide users access to the sites and web services they are familiar with. Some of these resources include images, scripts, fonts, stylesheets, and frames that may include additional documents. When a user visits a URL, the browser requests and downloads the site's base HTML document, which provides the browser with references to the embedded resources required to fully render the web page. The browser individually requests and downloads each of these embedded resources, and recursively requests any resources that may be present in the embedded resources. This process continues until the browser has fully downloaded all components required to render the site. Figure 2.1 demonstrates a sample web page structure modeled by a dependency graph.

Due to the one-by-one nature of requesting and downloading resources, the time required to download a site grows linearly with the number of site resources (i.e. the size of the site). According to the HTTP Archive, the median desktop WordPress site in February 2020 contained 88 embedded resources, and therefore made 88 resource requests [11]. Each of these requests introduces its own latency, since the size of the resource will directly impact the number of round trips required to download the resource. This delay compounds with the latency caused by parsing resources to identify further requests. Importantly, site resources may be hosted on servers from various domains, meaning that the browser may have to initiate multiple ClientHello's and establish multiple TLS connections—each with its own setup latency. Browsers and web servers have introduced various techniques to streamline the page load process, including connection reuse, simultaneous parallel connections, incremental resource parsing, resource scheduling and prioritization, etc. While these reduce the performance overhead created by multiple site resources, they do not completely eliminate it.

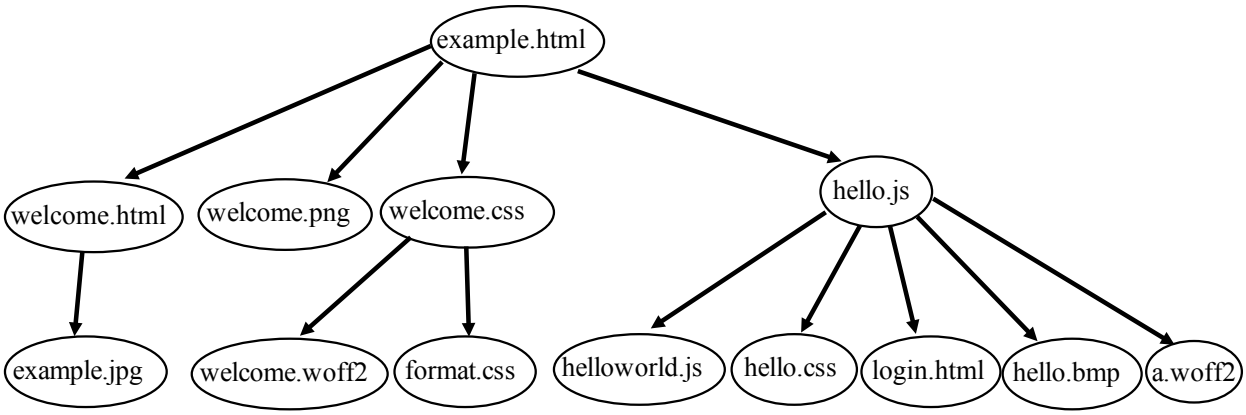

Figure 2.1: The structure of an example web page. The site resources include HTML documents (.html), stylesheets (.css), images (.bmp, .png, .jpg), scripts (.js), and fonts (.woff2). The arrows represent the dependency structure of the page; each resource points to other resources that it references/causes to be loaded.

2.2 HTTP/2 AND SERVER PUSH

One of the primary culprits behind high latency connections is HTTP/1.1's tendency to open multiple TCP connections to a server. Ironically, this solution was engineered in order to reduce latency by enabling concurrent data streams. However, given the large size of most modern web pages, the number of concurrent connections can grow too large and cause network congestion, packet loss, and weaker performance [12]. This performance hit is further compounded by the fact that HTTP/1.1's repetitive headers are prone to clogging the initial TCP congestion window [3]. The SPDY protocol [13]—which has since evolved into HTTP/2—was developed to address these issues.

These protocols establish a single TCP connection to an endpoint, and multiplex all of their HTTP requests and responses over the connection, thereby reducing network load. In order to efficiently pipeline across the single TCP connection, requests can be scheduled asynchronously, and can be prioritized to prevent head-of-line blocking and ensure that the most important resources are handled first. HTTP/2 also compresses headers to truncate redundant header data and allow multiple requests to be compressed into a single packet [3].

2.2.1 Server Push

HTTP/2 also introduces a feature known as *server push*, which enables the preemptive serving of site resources. This is motivated by the fact that certain requests from a client

can allow a server to predict future requests. Consider the site modeled in Figure 2.1. Based on the dependency tree of the site, it is clear that a client requesting example.html will certainly end up requesting resources such as welcome.html, welcome.png, etc. Therefore, when the server receives a request for example.html, it will proactively send resources in the site dependency tree without waiting for explicit requests from the client.

Under the right circumstances, this "pushing" can eliminate some of the latency introduced when a client has to parse an incoming resource and request embedded resources one-by-one. However, server push does not improve performance in all cases. Wang et al. note that server push can induce extra overhead by redundantly pushing unneeded or cached resources depending on push configurations and client latencies [12]. Zimmermann et al. find that suboptimal push strategies can also stall browser rendering pipelines by pushing resources in the wrong order [14, 15]. Recent research suggests that optimal push strategies may be quite complex and involved [16], and determining these strategies is the subject of ongoing work [17]

One of our goals in this work is to identify the *privacy* implications of the HTTP/2 server push mechanism under various web fingerprinting schemes. We posture that HTTP/2 with server push fundamentally changes the behavior of site traffic patterns, and that these changes can somehow impact the success of an adversary attempting to fingerprint users' browsing sessions.

2.3 THE TOR NETWORK

The Tor network is an anonymity communication system built as an overlay network. It aims to improve user privacy on TCP-based applications via *onion routing*—an approach that ensures that each *relay node* (or onion router) in a communication pathway only knows the current packet's previous and next hop nodes. By building a path—or *circuit*—across multiple relays to the endpoint, a client can ensure that no single node in the transmission gains full knowledge of the participants involved in a communication [18]. By maintaining this property, onion routing allows Tor users to remain anonymous by separating their identity from their browsing. As of April, 2020, Tor is used by an estimated 2,000,000 daily users worldwide, and consists of approximately 7,000 volunteer relays [19].

2.3.1 Tor Circuits

Communication over the Tor network occurs over client-constructed circuits, each containing three relays: an entry guard, a middle relay, and an exit relay. In order to send

traffic over the circuit, clients use local software known as an *onion proxy*, which is responsible for communicating with the entry guard. The onion proxy forwards client traffic to the entry guard, which then forwards communication to the middle relay. The middle relay will forward communication to the exit relay, which is responsible for communicating with the endpoint on the client's behalf. Note that the entry guard and middle relay will not know the destination of the communication (provided the client is using end-to-end encryption such as HTTPS), since the destinations they forward to are the middle and exit relays respectively. Similarly, the exit relay will not know the original client from which the communication originates, since its "client" will be the middle relay. Clients cycle between a set of three entry guards for 30 to 60 days at a time, and randomly choose a middle and exit relay for each circuit. Circuits are changed every 10 minutes to prevent deanonymization [20]. Note that randomly selecting the final two relays in the circuit can lead to variable performance over the Tor network, depending on latencies between any two hops in the circuit.

Once a client has established a circuit, they can communicate with the destination via multiple *streams*. Each of these streams corresponds to a single TCP connection between the exit relay and the destination server, and this system allows for multiplexed TCP communication over a Tor circuit. Communication over the Tor network utilizes TLS, and data is sent in fixed-size 512 byte cells [18]. This uniform cell size serves as a rudimentary protection against adversaries attempting to identify traffic based on individual packet sizes. As we will see in section 2.4.3, this protection falls short under more nuanced fingerprinting approaches.

2.3.2 Onion Services

Onion services, also known as location-hidden services, enable users to provide TCP services over Tor without revealing their IP address. These services are only available via the Tor network, and are accessed via self-certifying onion domains ending in `.onion`.

Rendezvous Points

Since onion services never reveal their IP address, clients must have another way of establishing a communication with them. This is achieved via the use of *rendezvous points*. An onion service establishes several onion routers as *introduction points* to which a client can connect to. The client chooses an onion router as their rendezvous point and communicates the rendezvous point to the onion service via the introduction point. The client will also include a hash of the service's public key and an optional authorization token in their message

to the introduction point. The client and onion service will then establish circuits to the rendezvous point and begin communication. The service's hostname will be self-certifying and is represented as `x.y.onion`, where `x` is the authorization token provided by the client, and `y` encodes the hash of the service's public key [18]. While this approach is efficient and secure, it can lead to human-unreadable hostnames that are comprised of random base32 strings.

Content of Onion Sites

Since they protect the location of servers, onion services are generally used to host services that would be censored or restricted by a powerful state-level adversary. These services range from news boards dedicated to free press and anti-censorship, to more illegal operations such as counterfeit document and weapons marketplaces. A study by Biryukov et al. crawled the onion service domain space, and determined that 56% of the active services providing HTTP(S) access were dedicated to politics, anonymity, security, and other generally permitted services or discussions on the regular Internet. However, the remaining 44% was related to drugs, adult content, counterfeit products, identity theft, and weapons [21].

Due to the sensitive and sometimes illegal nature of their content, many onion services are the target of governments attempting to shut them down. Note that even benevolent onion services, such as news and discussion sites dedicated to free speech, may be the target of large or well-funded adversaries who deem the activity of the service detrimental to their goals. As such, fingerprinting onion services and their users is of particular interest to many adversaries worldwide. Recent work has shown that large onion services are easily fingerprinted despite traffic obfuscation attempts [22].

2.4 WEB FINGERPRINTING

Web fingerprinting (WF) is a technique that allows an adversary to predict and/or identify a user's browsing activity by determining patterns or salient features in their web traffic. Useful features for web fingerprinting include packet sizes, counts, directions, and timings—all or most of which can be learned even if the traffic is encrypted or being served over an anonymity system or VPN [23, 24]. As mentioned earlier, web fingerprinting would be useful to adversaries attempting to censor or block traffic to certain sites, or to build user browsing profiles. Such adversaries could exist in the form of state-owned ISP's, or other malicious middlemen within the network.

2.4.1 Closed- vs. Open-World

At its core, web fingerprinting is a classification problem in which an adversary wishes to identify which site—out of a set of possible sites—a user is visiting. In order to do so, the adversary must gather traffic patterns generated by the set of possible sites, identify features that may aid in the classification task, and train a machine learning model on the gathered data. This scenario is referred to as the *closed-world*, since the adversary is assuming that the user will only make site visits within a reasonably finite set of websites. In the closed-world, the adversary has a sufficient number of traffic samples for each of the sites within the set to train their classifier on. The closed-world approach was favored by many pioneering WF studies, as its assumptions were sufficient for identifying which features of a site or web page were susceptible to fingerprinting [25, 26, 27, 28]. However, the approach has come under scrutiny for not being representative of the real world, in which the set of sites a user could visit is far too large for any realistic adversary to collect data on and train a classifier [29, 30]. In order for an adversary to take the closed-world approach and ensure coverage of the entire Internet, they would have to gather traffic data for *every* site/web page on the Internet, which is simply unrealistic.

An alternate strategy which satisfies the shortcomings of the closed-world approach, is known as the *open-world* approach. In this scenario, an adversary trains on traffic samples for a finite set of sites, but assumes that the user may visit sites outside of this set. This places a greater focus on building classifiers that can generalize, but note that it is impossible for a classifier to identify a site or page that it has not been trained on. Rather, open-world classifiers will predict whether a sample occurs within the closed-world on which it was trained (in which case it also predicts the site), or whether it occurs outside of the set altogether.

In this study, we take a closed-world approach to our experiments, as we are interested in identifying how various changes to site configurations can impact the accuracy of a fingerprinting adversary. An open world approach would not render our results invalid or unrealistic, since a closed-world classifier would in fact achieve higher accuracy than an open-world one (due to its finite and known sample set). If changes in accuracy apply to a closed-world scenario, then they will certainly apply to the more general open-world scenario.

2.4.2 Fingerprinting over the Web

It has long been known that object sizes—which are not protected by HTTPS—can reveal sensitive pieces of information that clue an adversary in on the site a user is visiting [31].

These findings led to the development of the first WF attacks, which would pinpoint pages within a single website [25, 32]. Later attempts began to employ the more traditional closed-world approach, and would identify pages from a set of potential sites [26].

A significant amount of work has gone into engineering the features that are used for web classification. As machine learning techniques have evolved and classifiers have become more refined, most approaches have settled on packet direction, order, and size as being the most important features for fingerprinting success. Some studies have also explored variations and combinations of these features, such as cumulative sizes of packets in a specific direction [33]. Recent work has noted the importance of packet *bursts*—a sequence of packets in the same direction—in fingerprinting [5], as they are representative of the request-response pattern of a site and can therefore help recover the site structure. We note that HTTP/2 with server push significantly changes the burst structure by changing the order and frequency at which resources are served, and thus has the potential to impact fingerprinting success.

Defenses

Initial work on defenses against web fingerprinting focused on ad hoc approaches that modified some of the aggregate features used in early fingerprinting work [34, 35, 36]; however, these have been shown to be largely ineffective by more sophisticated machine learning techniques [37, 2]. Recent studies have focused on variants of *constant rate* defenses [9, 8]—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). Certain constant-rate defenses even focus specifically on limiting the information revealed via packet burst sizes [38].

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, one of our goals is to evaluate this high-level intuition using actual HTTP/2 implementations in browsers and web servers.

2.4.3 Fingerprinting over Tor

Since Tor was built for the express purpose of providing user anonymity and privacy, fingerprinting attacks against Tor traffic can be especially dangerous or compromising. Tor users access sensitive data over the network—either by using Tor to circumvent censorship

restrictions, or by accessing onion services—and adversaries who can successfully fingerprint Tor traffic may use the learned information for malevolent purposes.

The first attempt to fingerprint Tor simply attempted to port early WF approaches to Tor traffic [28]. The classification only yielded a 3% accuracy with a set of 775 sites, due to the fact that it focused heavily on packet length frequencies. The use of uniformly sized 512-byte Tor cells means that packet sizes do not serve as a useful classification feature for an adversary. However, researchers have identified other avenues of fingerprinting success over the past decade.

It turns out that some of the more nuanced approaches mentioned in section 2.4.2 apply well to fingerprinting Tor traffic, and can achieve 90-91% accuracy in closed-world scenarios. The *k-NN* attack makes use of packet orderings and burst patterns [5], the *CUMUL* attack measures cumulative packet lengths of *sequences* of traffic (groups of packets moving in the same direction) [33], and the *k-FP* attack makes use of the number of packets in a particular sequence [6]. All of these approaches share some important aspects in their classification features—none of them focus on the size of individual packets, and all of them incorporate packet direction metrics. This strategy has informed the current generation of Tor fingerprinting, which has adopted deep learning to achieve far more accurate results.

Sirinam et al. introduced Deep Fingerprinting (DF) in 2018, as the first deep learning-based fingerprinting approach for Tor that outperformed the previously mentioned WF strategies [7]. This work builds off of earlier attempts to automate feature extraction for WF attacks via deep learning [39, 40], and achieves a 98.3% accuracy in a closed-world scenario. Such performance is achieved by learning features from a simple bit-wise representation of packet directions in the original trace. Later studies have incrementally improved upon DF by providing more scalable, generalizable, and efficient deep learning models [41, 42, 43], but all follow the same approach of extracting features from packet direction representations. Again, we note that HTTP/2 with server push fundamentally changes site packet sequence frequencies—as all resources are pushed at once given a single request from the user—and that this change may impact the accuracy of a deep learning approach that relies heavily on packet direction to make its classification.

Defenses

Proposed WF defenses for Tor also follow a constant-rate philosophy in order to minimize the information leaked via packet bursts. *WTF-PAD* uses adaptive padding to only pad traffic when channel usage is low, thereby making it more difficult to identify concrete packet bursts [10]. *Walkie-Talkie* forces the Tor client to communicate with the server in

half-duplex mode, which causes non-overlapping packet bursts in either direction. Small amounts of padding and packet delay are added to cause similarities between the traces of different sites, therefore making classification more challenging [38].

CHAPTER 3: MEASURING THE IMPACT OF HTTP/2 ON WEB FINGERPRINTING

In this chapter, we build an understanding of how HTTP/2 and server push impact the fingerprinting landscape. We also suggest small changes to site configurations that can make successful fingerprinting of a site more difficult[1].

3.1 EXPERIMENTAL SETUP

To understand how HTTP/2 and server push affect fingerprinting, we perform a head-to-head comparison between HTTP/1.1 and HTTP/2 with server push. Normally this would be done by fingerprinting traces collected from sites served via HTTP/1.1, and comparing the results to traces collected from sites served via HTTP/2. Unfortunately, the deployment of HTTP/2 among web servers is still sparse, and the use of sever push is even more limited. In order to address this issue, we create a synthetic experimental environment in which we can serve the same web pages using both protocols. This allows us to collect traces for fingerprinting that are representative of both HTTP/1.1 and HTTP/2 deployments.

To ensure that our synthetic traces are realistic, we model the structure of our test web pages after the structure of actual popular web sites. The following sections describe our approach for creating and serving these models, as well as the trace collection and fingerprinting techniques we used.

3.1.1 Constructing Web Page Models

As discussed in Section 2.1, a web page is comprised of a number of different types of resources that can be represented as a dependency tree. We wanted the model pages for our synthetic environment to accurately mimic the structure of real web pages, so that they would be representative of real-world sites. To do so, we first perform a web crawl of the the 100 most popular websites—as listed by the Alexa ranking system [4]—with a headless version of the Chromium browser. We use the Chrome DevTools protocol[2] to intercept network events such as requestWillBeSent, responseReceived, dataReceived, and loadingFinished. This allows us to keep track of the requests and responses that are being sent and received by the browser during the loading of the page. From the `responses` field in the request metadata,

[1]The work in this chapter was jointly performed with, and draws from prior work by Weiran Lin (weiranl@andrew.cmu.edu)

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

```
<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>
```

Figure 3.1: 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).

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, by tying request URL's to specific resources, we are able to learn what previously loaded resource is responsible for loading another 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 crawl. In our model, we create "dummy" resources that correspond to each resource in the real page with the same resource type. We also maintain the dependency structure of the real-world site by inserting necessary references to other resources within the model resource. For example, in Figure 2.1, the top-level HTML file, example.html, loads an image, welcome.png, another html page, welcome.html, a stylesheet, welcome.css, and a script, hello.js. Likewise, our top-level model file correspondingly contains references to modeled versions of each of these resources, as shown in Figure 3.1. It also contains a comment that pads the model resource to the same size as the original example.html. The padding comment varies in length depending on the size of the target resource.

Since non-page-root resources can also load dependenciees, we include directives to request dependencies in *any* model resource that loads further resources. For example, script0.js, which models hello.js in Figure 2.1, will include JavaScript commands to load another script, a stylesheet, a document, an image, and a font; it will then be padded to the same size as hello.js using comments. To model images, we create a 1×1 pixel GIF file with a comment section that pads the object to the size of the original image (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 Figure 3.2

We have to use special handling for fonts, as Chromium does not load a font unless it

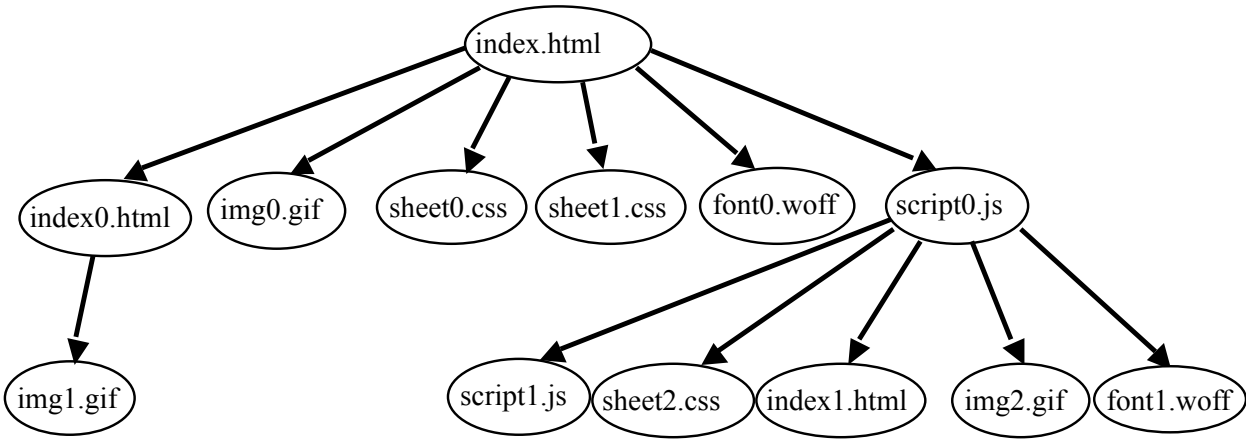

Figure 3.2: A model of the example web page from Figure 2.1. The file types and sizes all match the resources used in original page. The dependency structure closely mimics the original, but note that there is a deviation in how the font0.woff and sheet1.css are loaded.

is actually used to render some text. We therefore add a text element styled to use the loaded font inside an HTML file. In our testing, however, we discovered that Chromium reports that a font load is initiated by the HTML file in which it is used, even if the font is specified in a separate stylesheet (CSS file). This means that our model will miss a dependency when a font is loaded by a stylesheet. For this reason, font0.woff in Figure 3.2, which models welcome.woff2 in Figure 2.1, depends only on the main page and not the stylesheet sheet0.css, which models welcome.css. We found that a similar problem occurs when a stylesheet loads a second sheet using the @import directive. This results in another discrepancy for the dependency structure of sheet1.css in Figure 3.2, which models format.css in Figure 2.1. However, since our goal is to create representative web pages, rather than perfectly model each site, we determined that these slight deviations to the original page were acceptable within reason.

To validate the loading behavior of our models, we load the model web pages through headless Chromium and use the DevTools protocol to note the request and response sizes, types, and order; we find that the models match the original web page behavior.

3.1.2 Trace Collection

We now detail the methods used to host our model sites, as well as those used to collect the traces used for fingerprinting. Our site models are hosted using the Caddy web server[3], pages

[3] `https://caddyserver.com/`

are loaded via headless Chromium. The Caddy web server supports both HTTP/1.1 and HTTP/2 with (or without) server push, which allows us to model the differences between both protocols by making a simple configuration change to the server when hosting our models. Therefore, we configure the server to use HTTPS and HTTP/1.1 or HTTP/2, depending on the test being performed. We note that some of the modeled websites were originally served over HTTP, but we use HTTPS in our experiments, motivated by the trend towards greater HTTPS deployment. In order to best simulate communication between a client and server on isolated machines and wholly separate networks, the Caddy server and Chromium browser are run in separate Docker containers connected by an isolated network. The browser container queries the site model and concurrently runs `tcpdump` to produce a PCAP file that serves as our web trace. We process the PCAP file to filter out DNS packets, omit any TCP packets that carry no data, and record only the sizes, directions, and timings of each packet. These attributes are then used as input for fingerprinting.

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

3.1.3 Classification Techniques

To perform web fingerprinting, we use the features developed by Wang et al. [5] for their k-nearest neighbors classifier. Each packet stream is converted into 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 [6]—hence we use a random forest classifier in our work. Later work has explored using different features [33] as well as deep learning [40] for classification; however, we use the random forest technique because it allows us to perform an analysis of *feature importance*. Having a measure of which features are contributing to fingerprinting success allows us to formulate techniques to mitigate the impact of those features.

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 yields higher classifier performance and thus better highlights the information disparities resulting from protocol changes.

Table 3.1: 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	$62.1\% \pm 4.8\%$

3.2 EXPERIMENTS AND RESULTS

We next discuss the experiments we performed and their results. A summary of the experimental results can be found in Table 3.1.

3.2.1 Sites and Models

We collect 30 traces each from the top 100 web sites as ranked by Alexa [4]. Traces are collected in a round robin fashion to account for changes in a site's content over the duration of the collection phase. However, one of the websites was down during our collection, and we are therefore left with $99 \times 30 = 2970$ traces for our experiment. We note that our random forest classifier, as described in 3.1.3, performs very well on these traces: it has a classification accuracy of $91.3\% \pm 4.3\%$.[4]

We use each of the 30 traces to create a unique model for the relevant site. We note that, as many of the real-world web pages we load are dynamic with constantly changing content, different traces can—and often do—result in different page models. For example, Figure 3.3 shows that many sites exhibit significant variance in the total size of all the objects that are downloaded. This creates significant challenges for fingerprinting, when predictions are partially reliant on classifying resource sizes. For example, when we collect 30 traces using only a single model per site (i.e., using 1×99 models), served over HTTP/1.1, our classifier achieves $99.9\% \pm 0.5\%$ accuracy. On the other hand, using a different model for each trace and each site (i.e., using 30×99 models) causes the accuracy to fall to $80.2\% \pm 3.8\%$.

One point of concern is that the classification accuracy of the synthetic sites is lower than that of the real sites. We hypothesize that this is due to differences in how resources are *served* in our experiment versus the real world. In our experiment, all resources for a site are served from a single Caddy web server instance, whereas real sites typically access

[4]All of the accuracy numbers in this chapter 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.

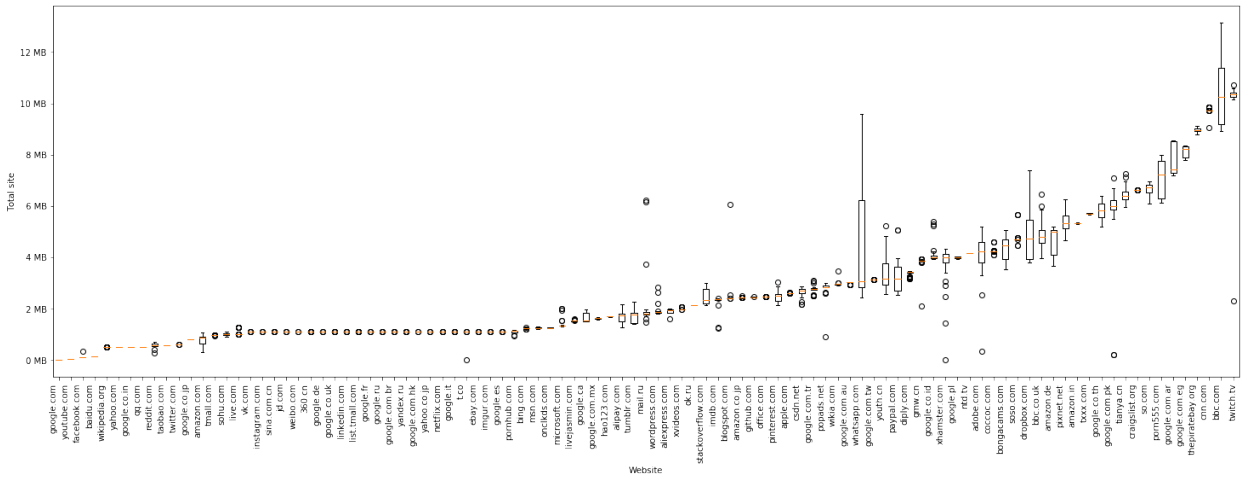

Figure 3.3: The distribution of total page size across 30 traces for each of the 99 sites in our data set. Each distribution is shown as a box plot, where the dashed line represents the median, boxes capture the interquartile range, the whiskers represent the range, and individual points represent absent outliers.

several origins at a time (image servers, ad servers, etc.) in order to load their resources, 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) [44, 45], and CDNs are developing technology such as the HTTP/2 origin frame [46] to allow serving more components via a single server connection. We attempt to address this issue in Chapter 5 by building site models that incorporate the multi-origin nature of most sites.

A second factor that may be contributing to the lower accuracy is that we change the site names of the models 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.

3.2.2 HTTP/2 Server Push

To evaluate the impact of HTTP/2 with server push on fingerprinting, we configure Caddy to use HTTP/2 and to push all of the site resources in a model as soon as the top-level HTML is requested. We then collected 30×99 traces using Chromium and tcpdump. The traces were fingerprinted using the same random forest classifier and yielded a lower accuracy of $74.2\% \pm 4.3\%$. Therefore, we conclude that HTTP/2 with server push is able to reduce the

amount of information available to fingerprinting, most likely due to changes it makes to traffic burst patterns (as hypothesized in 2.2.1).

HTTP/2, of course, introduces a number of other changes over HTTP/1.1 in addition to sever push. To understand the influence those changes have on our measured HTTP/2-with-server-push accuracy, we conduct a second set of experiments that use HTTP/2 but not server push. The resulting accuracy is $80.4\% \pm 2.5\%$, which is virtually identical to the HTTP/1.1 scenario.

3.2.3 Reducing Fingerprintability with Padding

As can be observed in Figure 3.3, there is some variability in the total size of the download between different instances of the same web page. Higher size variability between different instances of the same site makes it harder to accurately fingerprint that particular site. 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 $58.1\% \pm 2.8\%$.

We therefore consider adding padding to reduce the amount of information that can be learned from the site size alone. 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 for each model, and link to it from the main HTML page of the corresponding model. In practice, to avoid the overhead associated with parsing the JavaScript dependency, an empty image could be used. As an alternate approach, the server could possibly be configured to push a padding file not referenced by any resource in the dependency tree. The results of the padding experiments are shown in Figure 3.4 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 drops to $68.5\% \pm 4.6\%$. 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.

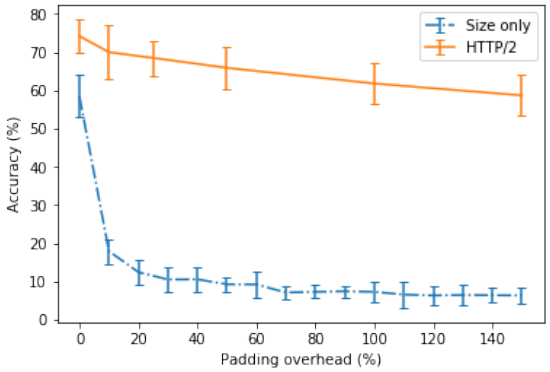

Figure 3.4: Results of adding padding to pages served with HTTP/2 server push, compared to the classification results that only use the total download size.

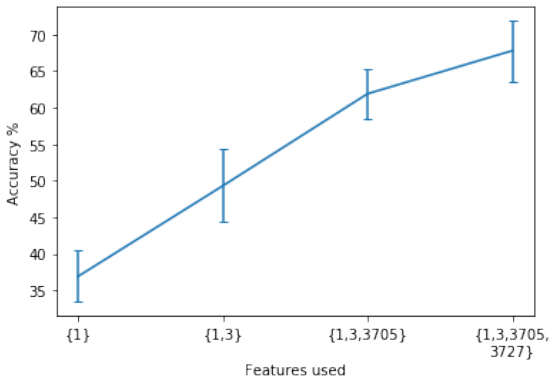

Figure 3.5: Feature selection for HTTP/2 with 25% padding.

3.2.4 Feature Analysis

To understand what features impact the fingerprinting accuracy of our classifier, we perform a feature importance analysis. We first use recursive feature elimination—similarly to the method described in 3.1.3—to reduce the 100 features down to 20 by eliminating one feature at each step. We then perform a greedy search to find the most impactful 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 feature yielding the highest accuracy, $i^* = \arg\max_i \mu_i$, and pair it with all remaining features to calculate μ_{i^*+j} to identify the next most significant feature. We let S_i denote the set of the 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 this 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: maximum burst size

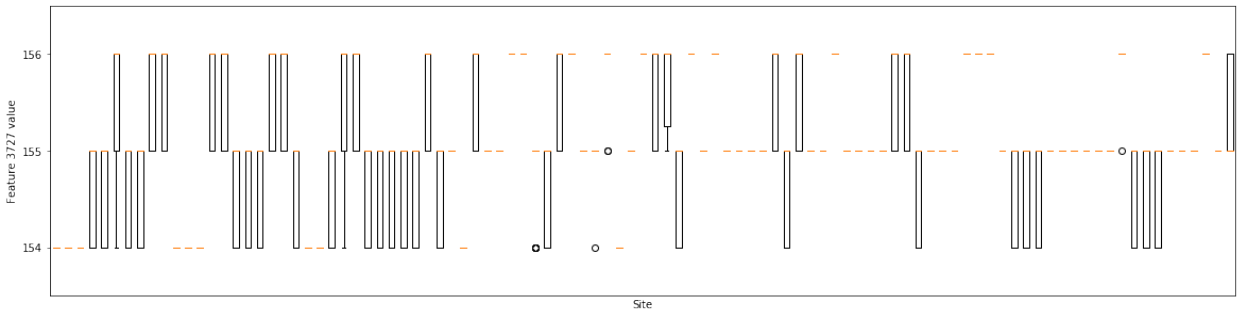

Figure 3.6: Distribution of values for feature 3727: size of the 12th packet

- 3727: size of the 12th packet in the trace

The accuracy of using these features is shown in Figure 3.5. We note that the first three features are strongly correlated to the total size of the downloaded data. The last feature, however, seems quite arbitrary, and 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 Figure 3.6.

Upon further inspection, this packet contains the first Push Promise—an indication to the client that the server intends to push a resource without requiring a request from the client—which includes the file name of the first file pushed by the Caddy server. We note that the name of the first pushed file can range in length from 8 characters (img0.gif) to 10 characters (sheet0.css), depending on the resource file type, which explains the difference in the lengths.

3.2.5 Site Resource Renaming

Having identified the 12th packet as a significant source of fingerprintability, we now detail a remedy to nullify this feature. We observe that the file names of site resources present in HTTP/2 Push Promise packet headers are Huffman encoded. Therefore, it is possible for different Push Promise packets to vary in size depending on the Huffman encoded bit-length of the contained file name.

In order to address this issue, we rename all site resources—except index.html—to be of a constant 61-bit Huffman encoded length. We first pad all of the file extensions (.js, .gif, etc.) to have a 40-bit Huffman encoded representation. This is done by prepending each file extension with a string of characters X, such that X.extension can be represented as a 40-bit Huffman encoding. For example, the .js file extension is padded to aabb.js and the .css extension is padded to bbz.css. Each site resource is then given a unique 3-character

identifier which can be represented by a 21-bit Huffman encoding. Identifiers are generated from a set of characters with 7-bit Huffman encodings. This identifier is prepended to the file's corresponding Huffman-padded file extension in order to generate the final resource file name. As an example, a valid rename for a script file would be PETaabb.js.

We observe that enforcing constant Huffman file lengths has a noticeable effect on fingerprinting success when paired with the padding scheme specified in 3.2.3. Classifier accuracy drops from $68.5\% \pm 4.6\%$ before resource renaming, to $62.1\% \pm 4.8\%$ after renaming.

CHAPTER 4: INFORMING AN EXPLORATION OF TOR FINGERPRINTING

After completing the experiments discussed in Chapter 3, we wanted to shift our focus towards performing fingerprinting on Tor traffic. More particularly, we wanted to see whether our findings would hold true in the Tor setting as well. In this chapter, we discuss how the results of the previous chapter inform the direction and framing of our experiments with Tor.

4.1 CHANGES TO SITE CONFIGURATION

We start by observing how different changes in site configuration impacted classification accuracy for our random forest classifier. These variations included padding site names to the same length, serving the site via HTTP/2 with server push, adding padding to each model, and renaming model resources to constant-length Huffman-encoded names. All of these adjustments managed to decrease classification accuracy compared to the model baseline of $80.2\% \pm 3.8\%$. Our most significant decrease in accuracy ($62.1\% \pm 4.8\%$) came when all of these site configuration changes were applied simultaneously. Given that Tor traffic patterns exhibit different behavior than regular web traffic, we frame our next experiments to understand whether the aforementioned changes can provide the same (or similar) benefits under a Tor browsing context. Essentially, we aim to conclude whether a change that reduces fingerprinting accuracy over the regular Internet will also reduce fingerprinting accuracy over Tor.

4.1.1 Onion Services

Tor also provides a unique opportunity to measure our site configuration changes under a fundamentally different model—onion services. As detailed in 2.3.2, connecting to an onion service is a more involved process than connecting to a regular service over the Tor network, as a rendezvous point must be established for traffic to funnel through. We are curious to see whether the idiosyncrasies of onion services impact their susceptibility to our configuration changes, and if so, to what degree. Since the safety of onion services is heavily tied to their ability to remain anonymous and unidentifiable, we believe that it is important to understand what strategies can be used to make fingerprinting onion services as difficult as possible. By looking at the results of Chapter 3, we hypothesize that the site changes we introduced may make onion services harder to fingerprint.

4.1.2 Gaming the State-of-the-Art

All state-of-the-art Tor fingerprinting techniques rely on deep learning approaches that focus primarily on a single source of feature extraction: *packet direction*. We note that frequencies of packets heading in either direction are heavily tied to the traffic burst patterns of a given site. The more bursts present in a packet trace, the more transients that can be picked up by the classifier to build a profile of the site. Importantly, HTTP/2 with server push drastically changes the packet burst patterns exhibited during a site load by pushing all resources at once, rather than waiting for individual requests for each resource. This should, in theory, reduce the number of bursts in a packet trace, thereby making it more difficult for the classifier to extract features. We therefore want to test whether HTTP/2 with server push adversely affects the accuracy of state-of-the-art Tor classifiers by nature of their own classification strategies.

4.2 HOSTING SITE RESOURCES ACROSS MULTIPLE SERVERS

As noted in 3.2.1, our original site models are all served from a single server per model. This is not representative of real-world sites, which often host resources across multiple servers and require clients to request these resources from their respective servers. Therefore, we attempt to remedy this issue moving forward by creating site models that host resources across multiple server instances. We also see whether this change in how sites are hosted has an impact on our classification success across both HTTP/1.1 and HTTP/2 scenarios.

CHAPTER 5: MEASURING THE IMPACT OF HTTP/2 AND SITE HOSTING CONFIGURATIONS ON FINGERPRINTING OVER THE TOR NETWORK

This chapter covers our experiments with fingerprinting site models over the Tor network under conditions engineered in Chapter 3. We also build site models that serve resources from multiple virtual servers in order to more closely represent real-world site hosting. Additionally, we examine how these conditions affect fingerprinting performance on onion services.

5.1 EXPERIMENTAL SETUP

Our general approach to setting up our Tor testbed remains relatively similar to the previous experiment. Real-world sites are still modeled in order to be re-hosted under our engineered configurations. However, certain nuances of multi-host site representations, onion services, and communication over the Tor network cause us to make slight changes to the way we model, host, and communicate with our site models.

5.1.1 Constructing Multi-host Web Page Models

In this experiment, we construct site models for the 20 most popular sites—as listed by the Alexa ranking system [4]—under two representations: single-host (where all resources are served from the same server) and multi-host (where resources are served from multiple servers as deduced from the original site traces). We create 30 models per site under each representation, and single-host site models are constructed in the same manner detailed in 3.1.1.

Constructing multi-host site models requires us to pay closer attention to the originating server of each resource, as detailed by the `server` value in the `responses` field of the request metadata. For each site model, we separate resources based on the domain names of their hosting server. Consider a model of the homepage for the Chinese web portal QQ, qq.com. All resources served from the homepage will be denoted by a `server` value of `qq.com`. Other resources from the homepage may be served from different servers, such as apis.map.qq.com, btrace.qq.com, and view.inews.qq.com among other locations. The `server` value of these resources reveals them as such.

A new model Caddy server is created for each host accessed throughout the page load, and each is responsible for hosting the resources served by the corresponding host. Importantly, the only model server that has an index.html file is the model homepage server, since this

```
<head>
<link rel="stylesheet" href="sheet0.css">
<script src=
    "https://www-gstatic-com.torsites.ttat.xyz/script0.js">
</script>
<body>
<img src="https://i-yt-img-com.torsites.ttat.xyz/img0.gif">
<iframe src="index1.html"></iframe>
<!--AAAAA[...]AAA-->
</body>
```

Figure 5.1: The top-level HTML file in a web page model, which loads a stylesheet (sheet0.css) and another HTML file (index1.html) from the top-level server. The script (script1.js) and image (img0.gif) are sourced from different model servers as denoted by their URLs.

is the server that will be handling the incoming site request. The site dependency tree is built in the same manner as our single-host models, but references to resources hosted on servers other than the requesting server are made via complete URLs indicating the server and filename of the target resource. This is necessary because the client must be able to request each foreign resource from a separate server with its own unique URL. An example index.html snippet for a multi-host model is given in Figure 5.1.

In this example, the model consists of 3 servers—the top level responding server hosting index.html, index1.html, and sheet0.css; and 2 external hosts, www-gstatic-com.torsites.ttat.xyz and i-yt-img-com.torsites.ttat.xyz, responsible for hosting script0.css and img0.gif respectively. www-gstatic-com.torsites.ttat.xyz is the model representation of the real-world host, www.gstatic.com, and i-yt-img-com.torsites.ttat.xyz represents real-world host, i.yt.img.com.

Note that resources loaded from external hosts may also reference resources stored on other external hosts. These resources must specify the *full* URLs of their target resources. This is especially important when a script is using the document.write directive to modify the top-level index.html to request an external target resource. Even if the target resource is hosted locally from the perspective of the writing script (i.e. on the same server as the script performing the document.write), it is an external resource from the perspective of the client, since it is hosted on a different server than the top-level responding host. As such, a full URL must be specified to ensure that the client requests the resource from the correct destination after parsing the modified index.html.

5.1.2 Modeling Onion Services

In order to test our configuration changes on more critical and Tor-unique browsing, we create single- and multi-host models for 20 different onion services. Once again, we create 30 models per site under each representation. While there are onion directories which provide lists of various onion services by category, they are not reliable for maintaining up-to-date lists of *active* services. Onion services often periodically change their hostname, or are subject to being taken down or moved, and as a result, many entries in the aforementioned directories point to inactive links. Instead, we use the regularly updated and filtered Real-World Onion Sites repository[1] to select 20 active onion services to model. We note that this repository performs content filtering to only include "clean" services (i.e. no nudity, exploitation, drugs, etc.), and excludes sites with an "onion-only" presence. Although this biases the content of our example sites, we argue that it makes no impact on the validity of our experiments, which are focused on fingerprinting onion services based on their structure and mechanisms.

Since onion services are only accessible via the Tor network, we cannot use the Chromium browser as-is to communicate with them. Instead, we modify the Mida web measurement tool[2] to access the Tor network via the Tor SOCKS proxy. Each Mida crawl is executed in its own Docker container, and outputs crawl metadata that is used to build a site model. Single-host site models are built as described in 3.1.1, and multi-host models are built under guidelines similar to those outlined in 5.1.1. There is, however, one key distinction: since we want to gather traces that are representative of traffic to an onion service, we must *host* the models as onion services themselves. This involves establishing $20 \times 30 = 600$ unique onion services—one for each of our site models—and pointing each onion service to the server responsible for hosting its corresponding model's homepage. We achieve this by adding the snippet of code shown in Figure 5.2 to the `torrc` of our host system and restarting the Tor daemon. This causes the Tor daemon to establish onion service descriptors for each service—detailing introduction points and the service's public key—and publish them to the directory server, thereby making our onion services publicly accessible.

We note that our multi-host onion service models make a few digressions from their original representations. It is possible for an onion service site external resource to be hosted on another onion service rather than a regular web server, and we observed this behavior in a few of the services that we modeled. However, due to time constraints, we were not able to host external resources on onion services, since doing so would cause a significant increase

[1] `https://github.com/alecmuffett/real-world-onion-sites`
[2] `https://mida.sprai.org/`

```
HiddenServiceDir /var/lib/tor/<model-name>
HiddenServicePort 443 127.0.0.1:<model-port>
```

Figure 5.2: The configuration lines added to the `torrc` for each onion service we wish to establish. For each model, we replace `<model-name>` with the model's name, and `<model-port>` with the port on which the local server is hosting the model. This allows us to tie a public key/onion hostname to each model, and to redirect incoming HTTPS requests to the required server port.

in the number of onion services we needed to establish, and would greatly increase the time required for our Tor daemon to publish the onion service descriptors. Therefore, we only host the model homepage and resources served by the home server on our model onion services; all external resources are hosted by regular HTTPS server instances.

5.1.3 Trace Collection

In order to gather our model traces over the Tor network and model a Tor browsing context, we use a Dockerized version of the Mida web measurement tool, which uses headless Chromium to contact sites via a Tor SOCKS proxy. The proxy establishes a connection with a Tor entry guard before serving as the browser's access point to the Tor network. Each trace is collected from a separate Dockerized Mida instance, meaning that a fresh Tor connection is established for each crawl. This ensures that our trace collection gathers examples across a wide variety of Tor circuits and provides greater coverage of potential user connection characteristics. After the crawl has completed, each Docker container outputs a PCAP file which we use as our web trace. The trace is filtered to exclude any non-IPv4 and non-TCP packets before being processed for classification.

We once again host our site models using the Caddy web server over HTTPS. This allows us to serve models over both HTTP/1.1 and HTTP/2 with server push. Site models each have their own unique URL, are hosted on a separate machine from the trace collector, and are hosted simultaneously via a single Caddy server which parses incoming requests and serves the desired site. This means we are able to collect traces from model A as well as model B without having to start a fresh server instance for model B. Since Tor exit relays must resolve each unique URL and each crawl is conducted from a separate Docker container with a unique Tor circuit, our hosting method should adequately model real-world communication to separate URLs in both single- and multi-host settings.

5.1.4 Classification Techniques

We perform closed-world fingerprinting of our traces with the Deep Fingerprinting (DF) classifier[3], as developed by Sirinam et al. [7]. The neural network is run with the hyperparameters specified in the original work, and we process our packet traces to satisfy the input constraints of the DF classifier. Each trace is divided into packets of 1500 bytes each, and the directions of these packets are recorded as a vector with elements in the range $[-1, +1]$. Outgoing packets from the client (the Mida Docker container) are recorded as a $+1$, and incoming packets to the client are recorded as a -1. If the trace exceeds 5000 packets in length, only the first 5000 packets are recorded; likewise, traces that do not reach a length of 5000 packets have their vectors zero-padded to a length of 5000 elements. This data representation is noticeably simpler and more lightweight than what was used for our random forest classifier in Chapter 3, since the DF classifier relies entirely on packet directions in order to perform feature extraction. As a result, we can ignore metrics such as packet size.

More advanced deep learning-based Tor fingerprinting approaches have been designed in the year following DF's release, including *Var-CNN* [42] and *triplet fingerprinting* [41]. These approaches focus on constructing deep learning classifiers that can achieve strong performance with more realistic adversarial models, i.e. less training data gathered over a much more limited set of sites. However, we use DF as our classifier for this work since it is the best documented and most accessible implementation available, and since we are focused on recognizing trends in fingerprinting given site configurations rather than commenting on the outright accuracy or usability of our classifier.

5.2 EXPERIMENTS AND RESULTS

We conduct a total of 12 classifications split evenly between one of two scenarios: classifying regular sites and classifying onion services. Each scenario is tested with 6 experiments, as detailed below. A summary of the experimental results can be found in Table 5.1.

1. **Unpadded models, single host, HTTP/1.1:** The control models for single-host testing. Models are served from a single host and *do not* have any of the anti-fingerprinting mechanisms introduced in Chapter 3. Models do not possess 25% padding or resource filenames padded to a constant Huffman length. Models are served via HTTP/1.1.

2. **Padded models, single host, HTTP/1.1:** Models are served from a single host

[3] `https://github.com/deep-fingerprinting/df`

Table 5.1: Summary of results. Experiments are numbered.

Model	Experiment	Accuracy
Regular sites	**1.** Unpadded, single host, HTTP/1.1	$58.0\% \pm 3.8\%$
	2. Padded, single host, HTTP/1.1	$54.9\% \pm 7.8\%$
	3. Padded, single host, HTTP/2 with server push	$54.7\% \pm 5.8\%$
	4. Unpadded, multiple hosts, HTTP/1.1	$67.1\% \pm 7.5\%$
	5. Padded, multiple hosts, HTTP/1.1	$68.2\% \pm 6.3\%$
	6. Padded, multiple hosts, HTTP/2 with server push	$65.7\% \pm 5.4\%$
Onion services	**7.** Unpadded, single host, HTTP/1.1	$76.7\% \pm 7.2\%$
	8. Padded, single host, HTTP/1.1	$78.3\% \pm 7.2\%$
	9. Padded, single host, HTTP/2 with server push	$56.7\% \pm 2.7\%$
	10. Unpadded, multiple hosts, HTTP/1.1	$75.2\% \pm 5.3\%$
	11. Padded, multiple hosts, HTTP/1.1	$65.53\% \pm 13.0\%$
	12. Padded, multiple hosts, HTTP/2 with server push	$65.5\% \pm 7.8\%$

with 25% padding and resource filenames padded to a constant Huffman length. Models are served via HTTP/1.1.

3. **Padded models, single host, HTTP/2 with server push:** Models are served from a single host with 25% padding and resource filenames padded to a constant Huffman length. Models are served via HTTP/2 with server push enabled.

4. **Unpadded models, multiple hosts, HTTP/1.1:** The control models for multi-host testing. Models are served from multiple hosts and *do not* have any of the anti-fingerprinting mechanisms introduced in Chapter 3. Models do not possess 25% padding or resource filenames padded to a constant Huffman length. Models are served via HTTP/1.1.

5. **Padded models, multiple hosts, HTTP/1.1:** Models are served from multiple hosts with 25% padding and resource filenames padded to a constant Huffman length. Models are served via HTTP/1.1.

6. **Padded models, multiple hosts, HTTP/2 with server push:** Models are served from a single host with 25% padding and resource filenames padded to a constant Huffman length. Models are served via HTTP/2 with server push enabled.

5.2.1 Collecting Classification Data

Similarly to our experiments in Chapter 3, we collect 30 traces for each of the sites we wish to model (both regular and onion services). Traces are once again collected in a round-robin

fashion to account for site content changes over the duration of the collection period. Once the sites have been modeled and hosted using the configurations required for the desired experiment (e.g. single- vs. multi-host, HTTP/1.1 vs HTTP/2, etc.), we collect 5 traces per site model—each from a separate Mida run—giving us a total of $30 \times 5 = 150$ traces per model. Over a set of 20 sites per experiment, this yields a final dataset of $150 \times 20 = 3000$ traces per experiment with which to train and test the DF classifier. Since each model crawl establishes a new Tor circuit, we have the opportunity to model communication through a wide range of Tor entry guards; in fact, throughout our experiments across both regular and onion services, we establish Tor circuits through 2493 unique entry guard relays. Considering that the Tor network only consists of about 7000 total relays, this likely means our models were collected over circuits spanning a fair portion of the overall Tor network.

Following the methodology used by Sirinam et al. to evaluate their DF classifier [7], we split each dataset into training, testing, and validation sets based on an 80%/10%/10% split. For each of the 20 sites being classified, we randomly sample 120 out of its 150 traces and add them to the overall training set for the experiment. Out of the remaining 30 traces, we randomly sample 15 for the experiment's validation set and add the remaining 15 to the test set. After this has been done for all 20 sites, we are left with a training set of 2400 traces, and 300-trace testing and validation sets.

In order to mitigate any biases caused by unfavorable or favorable data splits, we create 10 separate datasets for each of the 12 classifications. Each of these datasets is independently sampled from the same set of traces, and allows us to evaluate classifier performance across a wider range of training and testing data. The accuracy scores reported throughout the remainder of this chapter are calculated by taking the mean accuracy of the 10 trials for each classification $\pm$ a single standard deviation of the accuracies.

5.2.2 Unpadded vs. Padded Site Models

The first goal of these experiments was to see whether padding the site models as we did in Chapter 3 would cause any decrease in fingerprinting accuracy when sites were visited over Tor. To this end, we compare the results of unpadded models served over HTTP/1.1 to those of padded models served over HTTP/1.1. We note that for regular sites served via a single host, and onion sites served via multiple hosts, adding padding manages to decrease classification accuracy; in the case of regular sites, accuracy drops from $58.0\% \pm 3.8\%$ to $54.9\% \pm 7.8\%$, and in the case of onion services, we see a drop from $75.2\% \pm 5.3\%$ to $65.5\% \pm 13.0\%$.

However, regular sites served from multiple hosts and onion services served from a single

host show little difference in accuracy when comparing padded and unpadded models. In fact, the padded models in both of these cases yield a marginally higher classification accuracy than the standard unpadded models; we see an accuracy increase from $67.1\% \pm 7.5\%$ to $68.2\% \pm 6.3\%$ for regular site models, and an increase from $76.7\% \pm 7.2\%$ to $78.3\% \pm 7.2\%$ for onion services.

Based on these results, we cannot draw any confident conclusions about the efficacy of site padding as a fingerprinting defense over Tor. While we observe decreased classification accuracy in some cases, we also see relatively unchanged and slightly higher accuracies in other cases. This could be due to faults in our data (discussed in 5.3), or more complex factors such as the relationships between Tor cells and site content, or idiosyncrasies in the traffic patterns of onion services.

5.2.3 HTTP/1.1 vs. HTTP/2 with Server Push

When comparing the results of padded models collected over HTTP/2 with server push, to those of padded models collected over HTTP/1.1, we see a general trend indicating that HTTP/2 with server push could make Tor fingerprinting slightly less accurate. In the case of single-host regular site models, we see a reduction in accuracy—albeit a small one—from $54.9\% \pm 7.8\%$ when served via HTTP/1.1, to $54.7\% \pm 5.8\%$ when served via HTTP/2 with server push. Likewise, multi-host regular site models exhibit an accuracy drop from $68.2\% \pm 6.3\%$ to $65.7\% \pm 5.4\%$.

The largest drop in classification accuracy is seen with onion services hosted via a single host. While serving these models over HTTP/1.1 yields a classification accuracy of $78.3\% \pm 7.2\%$, using HTTP/2 with server push sees the classifier fall to an accuracy of $56.7\% \pm 2.7\%$. This accuracy drop, however, is not reflected as dramatically in the multi-host onion service models. The classification accuracies for these models are virtually identical regardless of which protocol they are served with. Serving them via HTTP/1.1 yields an accuracy of $65.53\% \pm 13.0\%$, which is negligibly higher than the $65.5\% \pm 7.8\%$ accuracy achieved with HTTP/2 and server push. Regardless, we believe that the general trend demonstrates that HTTP/2 with server push can provide a certain degree of fingerprinting protection over regular HTTP/1.1.

5.2.4 Single- vs. Multi-host Models

Coming into this experiment, we believed that serving site models over multiple hosts would more closely represent real-world site structure, and would therefore yield higher

accuracies from fingerprinting techniques designed for fingerprinting real-world sites. If this were the case, then we would be able to recommend serving sites from a single host as a potential fingerprinting defense. We see this assumption reflected in the case of our regular site models—experiments 4, 5, and 6 all achieve higher classification accuracies than 1, 2, and 3 (see Table 5.1).

However, we do not see the same results in our onion service models. With the exception of models served via HTTP/2—which see single-host models yield a classification accuracy of $56.7\% \pm 2.7\%$ compared to multi-host models' $65.5\% \pm 7.8\%$—our onion services are, in fact, less fingerprintable when served via multiple hosts.

Once again, we cannot make a decisive claim as to whether serving sites from a single host is an effective fingerprinting defense given the discrepancies in results. It appears as though regular sites benefit from being served via a single host, and perhaps testing non-Tor models with multiple hosts (as suggested in Chapter 3) would shed further light on this hypothesis.

5.2.5 Regular Sites vs. Onion Services

As a general observation, we note that our onion service models are far more fingerprintable than our models of regular sites. We see 3 onion service experiments with classification accuracies exceeding 70%, while the most accurate regular site experiment has an accuracy of roughly 68%. Similarly, the onion service experiment with the lowest classification accuracy of $56.7\% \pm 2.7\%$ is still more fingerprintable than the 2 least fingerprintable regular site experiments.

This increased fingerprintability is concerning given the highly sensitive nature of many onion services, and the risky stakes under which their users operate. Given that the world of onion services and their users is far smaller than that of regular sites, an adversary who can successfully fingerprint a user's connection to an onion service may be able to do significant and actionable damage. While the observed accuracies could be due to issues with our data, if they are indeed representative of the fingerprintability of real-world onion services compared to regular sites, they may have some interesting implications.

5.3 EVALUATION OF RESULTS

In this section, we take a closer look at our results to determine why they exhibit certain patterns. We address the issues present in our experiments and make suggestions for future work.

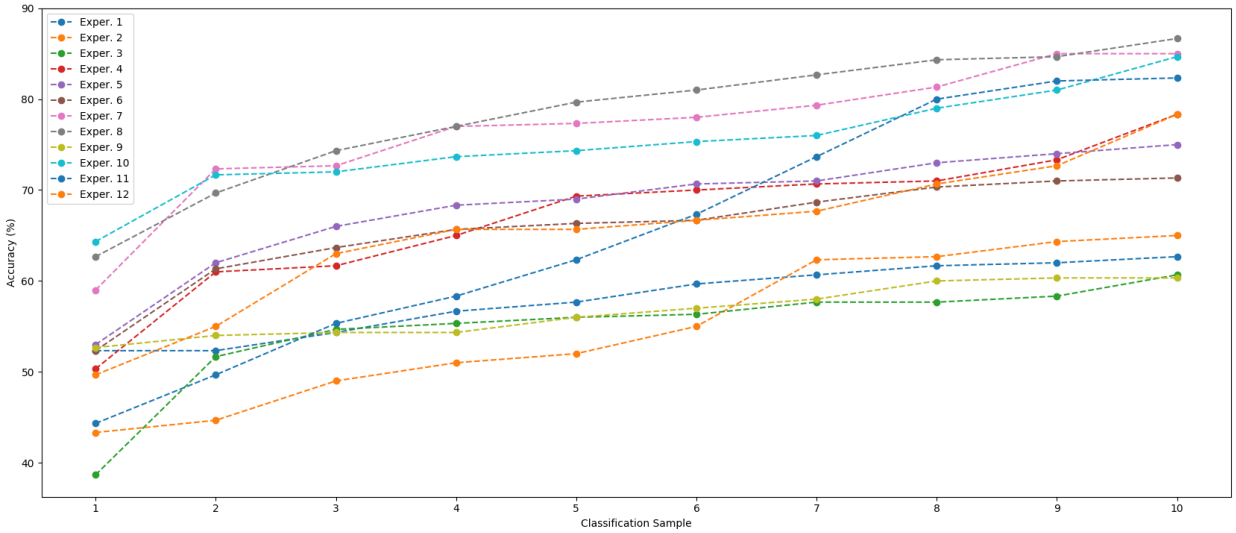

Figure 5.3: Distribution of experiment accuracies across 10 classification trials. Accuracies for each experiment are sorted in increasing order

5.3.1 Baseline Classifier Accuracy

After running the DF classifier on 10 datasets for each of the 12 experiments, we get the results listed in Table 5.1. We note that our accuracies for the unpadded models (which serve as our base representations of sites) range from $58.0\% \pm 3.8\%$ to $76.7\% \pm 7.2\%$ depending on the experiment. Strangely, these are far lower than the reported baseline accuracies for the DF classifier; Sirinam et al. demonstrate that the DF classifier achieves 90% accuracy in their experiments with just 50 traces per site, and with 1000 traces per site, can achieve 98.3% accuracy [7].

We believe that this discrepancy in overall accuracy is likely due to faults in our model construction, which are making them inconsistent with real world sites and onion services. As an example, our multi-host onion service models did not host external resources on other onion services, even though we observed this behavior in our crawls of real-world onion services. Given more time, we would re-create our models to remedy this issue. Future work on this project should certainly involve a closer examination of our model-construction strategies, as well as our site-hosting methodologies to ensure that they are as realistic and representative as possible.

5.3.2 Accuracy Spreads Within Experiments

We observe that the accuracies of numerous experiments fluctuate heavily across their spread of trials, as demonstrated in Figure 5.3. For example, Experiment 11 achieves a wide

spread of accuracies, ranging from as low as 45% to as high as 82%. These large variations across such a small sample space are likely the reason for our inconclusive data.

We believe that the demonstrated distribution is caused by poorly constructed datasets for certain trials. Since we are randomly sampling traces into training, validation, and testing sets, the lower accuracy trials occur when the test set is constructed with classes that are less present in the training set. Since we are reporting the classification accuracy as the mean of the 10 trials per experiment, a few poorly constructed datasets—and therefore, poor classifications—will skew the final results far more heavily than if we were to conduct more trials. Future work in this space would involve conducting more classification trials with more varied datasets in order to smooth out the results shown in Figure 5.3.

5.3.3 Additional Future Work

For future revisions of this experiment, we would begin by collecting more traces over a wider variety of sites (both regular and onion services). We limited ourselves to 20 sites for each type of model due to time restraints, but it would be far more valuable to test Tor fingerprinting over larger closed- and open-world scenarios which are more representative of real-world adversarial scenarios. Furthermore, we would evaluate our models under more recent deep learning-based classifiers, such as *Var-CNN* [42], and *triplet fingerprinting* [41].

Knowledge gained through these experiments could also be used to conduct a more thorough investigation of onion service fingerprintability. It would be interesting to explore how changes in site configuration impact onion services of different categories, such as news services vs. video hosting services. Understanding these implications could inform which types of services are most at risk, and what defenses they can employ to remain anonymous.

Finally, we would experiment with more nuanced HTTP/2 server push configurations. As suggested by previous work [14, 15, 17], push configurations can be extremely involved and require significant engineering to perform optimally. We would therefore attempt to further understand server push best-practice, and would create our site models respectively.

CHAPTER 6: CONCLUSION

Despite efforts to bolster web security via improved cryptographic measures and the adoption of more secure protocols, web fingerprinting poses a serious threat to the privacy and safety of users online. Not only does it allow an adversary to learn what sites and services users are accessing, but it gives malicious adversaries the knowledge required to target these services in more coordinated efforts. A web fingerprinting attack can even be successful against the traffic of anonymity systems such as Tor, which are host to users accessing critical and sensitive data. Early ad hoc defenses against web fingerprinting were easily circumvented with more advanced fingerprinting strategies, and newer defenses focus on constant-rate solutions that often require changes to transport and application layer protocols—changes that make adopting the defenses costly and difficult to deploy at a wide scale. In this thesis, we explored the feasibility and effectiveness of using changes to a site's *configuration* as a lightweight, easy-to-deploy fingerprinting defense.

We examined the impact of various factors—including HTTP/2 with server push, site padding, and single- vs. multi-server resource-hosting—on the accuracy of fingerprinting over the web and Tor. To achieve this, we created *models* of websites and onion services by crawling their real-world counterparts and creating site models that reflected the original sites' dependency structures, resources, and sizes. We then modified and hosted these models as required to test the impact of each of the aforementioned factors on fingerprinting success.

In order to conduct our experiments, site models were hosted via the Caddy server, which allowed us to serve them via HTTP/2 with server push. Our first experiments involved measuring fingerprinting performance on the web, and we used a random forest classifier to fingerprint site visits to each of our models. We found that serving a model via HTTP/2 with server push caused a significant reduction in fingerprinting accuracy when compared to the same model being served via HTTP/1.1. Additionally, we found that adding a small-to-moderate amount of padding to a site's content, as well as renaming all site resource names to be of a constant Huffman-encoded length, can further decrease fingerprinting accuracy. When we paired HTTP/2 with server push with our padding and resource-renaming approaches, we were able to reduce fingerprinting accuracy from 80% to 62%.

In our second set of experiments, we used the Deep Fingerprinting classifier to evaluate fingerprinting performance against our models when they were served via the Tor network. In addition to conducting experiments with regular sites, we modeled, hosted, and fingerprinted 20 real-world onion services. We also evaluated fingerprinting performance when sites were served from a single server, as opposed to multiple servers (which more closely resembles

real-world hosting practices). We found that HTTP/2 with server push marginally lowered fingerprinting accuracy over Tor, and that serving regular site models from a single server made them more difficult to fingerprint. However, onion services seemed more fingerprintable when served from a single server. We were also not able to make a conclusive statement in regards to the impact of site padding on fingerprinting over Tor; due to inconsistencies present in our Tor data, we believe that refinements need to be made in our site modeling, and that more fingerprinting data needs to be gathered as part of future work.

Excluding our Tor inconsistencies, however, we can issue a set of recommendations to most non-onion service websites that wish to reduce their users' susceptibility to fingerprinting over the web:

1. Serve as many site components as possible from a single server
2. Use HTTP/2 and server push on that server (also applicable to onion services)
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
5. Rename site resources to be of a constant Huffman-encoded length

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] 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)*, 2017, pp. 1323–1338.

[2] 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 *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 332–346.

[3] M. Belshe, R. Peon, and M. T. Ed, "Hypertext transfer protocol version 2 (http/2)," Tech. Rep., 2015, rFC 7540.

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

[5] 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)*, 2014, pp. 143–157.

[6] J. Hayes and G. Danezis, "k-fingerprinting: A robust scalable website fingerprinting technique," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 1187–1203.

[7] P. Sirinam, M. Imani, M. Juarez, and M. Wright, "Deep fingerprinting: Undermining website fingerprinting defenses with deep learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1928–1943.

[8] 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*, 2014, pp. 227–238.

[9] 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*, 2014, pp. 121–130.

[10] 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.

[11] "Report: State of the web," 2020. [Online]. Available: https://httparchive.org/reports/state-of-the-web

[12] 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)*, 2014, pp. 387–399.

[13] "Spdy: An experimental protocol for a faster web." [Online]. Available: https://www.chromium.org/spdy/spdy-whitepaper

[14] T. Zimmermann, J. Rüth, B. Wolters, and O. Hohlfeld, "How http/2 pushes the web: An empirical study of http/2 server push," in *2017 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 2017, pp. 1–9.

[15] T. Zimmermann, B. Wolters, and O. Hohlfeld, "A qoe perspective on http/2 server push," in *Proceedings of the Workshop on QoE-based Analysis and Management of Data Communication Networks*, 2017, pp. 1–6.

[16] A. Frömmgen, D. Stohr, B. Koldehofe, and A. Rizk, "Don't repeat yourself: seamless execution and analysis of extensive network experiments," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, 2018, pp. 20–26.

[17] T. Zimmermann, B. Wolters, O. Hohlfeld, and K. Wehrle, "Is the web ready for http/2 server push?" in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, 2018, pp. 13–19.

[18] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," Naval Research Lab Washington DC, Tech. Rep., 2004.

[19] "Tor metrics," 2020. [Online]. Available: https://metrics.torproject.org/

[20] T. Wang and I. Goldberg, "Improved website fingerprinting on tor," in *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, 2013, pp. 201–212.

[21] A. Biryukov, I. Pustogarov, F. Thill, and R.-P. Weinmann, "Content and popularity analysis of tor hidden services," in *2014 IEEE 34th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2014, pp. 188–193.

[22] 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*, 2017, pp. 2021–2036.

[23] M. Liberatore and B. N. Levine, "Inferring the source of encrypted http connections," in *Proceedings of the 13th ACM conference on Computer and communications security*, 2006, pp. 255–263.

[24] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website fingerprinting in onion routing based anonymization networks," in *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*, 2011, pp. 103–114.

[25] H. Cheng and R. Avnur, "Traffic analysis of ssl encrypted web browsing," *URL citeseer. ist. psu. edu/656522. html*, 1998.

[26] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu, "Statistical identification of encrypted web browsing traffic," in *Proceedings 2002 IEEE Symposium on Security and Privacy*. IEEE, 2002, pp. 19–30.

[27] A. Hintz, "Fingerprinting websites using traffic analysis," in *International workshop on privacy enhancing technologies*. Springer, 2002, pp. 171–178.

[28] D. Herrmann, R. Wendolsky, and H. Federrath, "Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, 2009, pp. 31–42.

[29] M. Juarez, S. Afroz, G. Acar, C. Diaz, and R. Greenstadt, "A critical evaluation of website fingerprinting attacks," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 263–274.

[30] M. Perry, "A critique of website traffic fingerprinting attacks," 2013. [Online]. Available: https://blog.torproject.org/critique-website-traffic-fingerprinting-attacks

[31] D. Wagner, B. Schneier et al., "Analysis of the ssl 3.0 protocol," in *The Second USENIX Workshop on Electronic Commerce Proceedings*, vol. 1, no. 1, 1996, pp. 29–40.

[32] S. Mistry and B. Raman, "Quantifying traffic analysis of encrypted web-browsing," *Project paper, University of Berkeley*, 1998.

[33] A. Panchenko, F. Lanze, J. Pennekamp, T. Engel, A. Zinnen, M. Henze, and K. Wehrle, "Website fingerprinting at internet scale." in *NDSS*, 2016.

[34] X. Luo, P. Zhou, E. W. Chan, W. Lee, R. K. Chang, and R. Perdisci, "Httpos: Sealing information leaks with browser-side obfuscation of encrypted flows." in *NDSS*, vol. 11, 2011.

[35] M. Perry, "Experimental defense for website traffic fingerprinting," 2011. [Online]. Available: https://blog.torproject.org/experimental-defense-website-traffic-fingerprinting

[36] C. V. Wright, S. E. Coull, and F. Monrose, "Traffic morphing: An efficient defense against statistical traffic analysis." in *NDSS*, vol. 9. Citeseer, 2009.

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

[38] T. Wang and I. Goldberg, "Walkie-talkie: An efficient defense against passive website fingerprinting attacks," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1375–1390.

[39] K. Abe and S. Goto, "Fingerprinting attack on tor anonymity using deep learning," *Proceedings of the Asia-Pacific Advanced Network*, vol. 42, pp. 15–20, 2016.

[40] V. Rimmer, D. Preuveneers, M. Juarez, T. Van Goethem, and W. Joosen, "Automated website fingerprinting through deep learning," *arXiv preprint arXiv:1708.06376*, 2017.

[41] P. Sirinam, N. Mathews, M. S. Rahman, and M. Wright, "Triplet fingerprinting: More practical and portable website fingerprinting with n-shot learning," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1131–1148.

[42] S. Bhat, D. Lu, A. Kwon, and S. Devadas, "Var-cnn: A data-efficient website fingerprinting attack based on deep learning," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 4, pp. 292–310, 2019.

[43] S. E. Oh, S. Sunkam, and N. Hopper, "p1-fp: Extraction, classification, and prediction of website fingerprints with deep learning," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 3, pp. 191–209, 2019.

[44] 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," *arXiv preprint arXiv:1806.08420*, 2018.

[45] 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*, 2017, pp. 677–684.

[46] M. Nottingham and E. Nygren, "The origin http/2 frame," Tech. Rep., 2018, rFC 8336.