PERFORMANCE AND SECURITY TRADEOFFS OF PROVABLE WEBSITE TRAFFIC FINGERPRINTING DEFENSES OVER TOR

BY

GIANG TRUONG KHOA NGUYEN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Associate Professor Nikita Borisov, Chair
Associate Professor Matthew Caesar
Associate Professor Philip Brighten Godfrey
Research Assistant Professor Rob Johnson, Stony Brook University

Abstract

The Internet has become an integral part of modern life. At the same time, as we spend increasingly more time online, our digital trails, including the identities of the websites we visit, can reveal sensitive personal information. As a result, researchers have devised schemes that seek to enable users to obfuscate the *network traffic fingerprints* of the websites they visit; however, being ad hoc attempts, these schemes have all been later found to be ineffective against more sophisticated attacks. Thus, researchers have recently proposed a family of provable defenses called BuFLO, or Buffered Fixed-Length Obfuscator, that provides strong privacy guarantees at the expense of high overhead.

Orthogonal to these defenses, the popular Tor anonymity network provides some protection against these attacks but is nonetheless susceptible. In this dissertation, we propose a simple design that uses BuFLO to protect web browsing traffic over Tor: tunnel the BuFLO channel through Tor. In order to evaluate the design, for both live experiments as well as large-scale simulations, we need precise models of the traffic profiles generated by a browser's visiting websites. This in turn requires us to obtain a fine-grained model of the web page loading process, two key components of which are the browser and the web page. After diving into the immensely complex web page loading process, we instrument the browser in order to extract bits of information as it loads a web page; this enables us to obtain the models for 50 top Alexa-ranked global websites. Following that, we build a traffic generator framework to generate network traffic according to the models. Next, we design and implement from scratch CS-Tamaraw, a congestion-sensitive version of Tamaraw, the most secure member of the BuFLO family.

With all the pieces in hand, we perform live experiments to confirm that CS-Tamaraw provides the predicted gains in privacy as in the original study. However, when CS-Tamaraw is tunneled through Tor as we propose, its de-

fense degrades significantly. We then conduct experiments to determine whether CS-Tamaraw is at fault. Both CS-Tamaraw and a simple, bare-bone, application-layer defense work largely as expected without Tor but are similarly afflicted when tunneled through Tor. Further investigations suggest that the unexpected results are due to artifacts in network conditions and not due to flaws in the design or implementation of CS-Tamaraw. We end after discussing the large-scale simulation studies with various levels of adoption of CS-Tamaraw.

To my family, for their love and support.

Acknowledgments

First and foremost, I want to thank my advisor Nikita Borisov; his guidance and support over the past eight years have been invaluable. I have benefited tremendously from being able to take advantage of his technical expertise and guidance. I appreciate that he allowed me the freedom to pursue my research, and to do so in a manner that is important to me. He has been patient with my slow research progress and numerous delays and extensions, and he was empathetic and supportive when I was going through a difficult period outside of research. This dissertation otherwise would not have been possible, and for all of that, I am extremely grateful.

I am also grateful to other members on my doctoral committee, Philip Brighten Godfrey, Matthew Caesar, and Rob Johnson. They have helped guide this dissertation with insightful questions and feedback. I also truly appreciate their understanding of and accommodating my numerous last-minute requests—I cannot imagine it has been easy being on my doctoral committee. Furthermore, I have also had the pleasure of collaborating with Brighten and Matt on other successful research projects.

Next, I would like to thank past and present members of the Hatswitch research group: Amir Houmansadr, Anupam Das, Gaurav Aggarwal, Joshua Juen, Prateek Mittal, Qiyan Wang, Sonia Jahid, and Xun Gong. It was a pleasure discussing security and privacy research and working with them; I am also thankful for their assistance on my projects. Outside of Hatswitch, I enjoyed collaborating with Ankit Singla, Chi-Yao Hong, Rachit Agarwal, and Virajith Jalaparti. Outside of research, I have to thank Carol Wisniewski and the staff at Coordinated Science Laboratory and the Computer Science Department for helping me with administrative processes.

Outside of work, I was fortunate to have found a sense of social belonging with other members of the Vietnamese Student Community at Illinois, with friends who enjoy soccer and music as much as I, and with members of the

Urbana-Champaign Vietnamese community at large. The have enriched my experience in Urbana-Champaign the last eight years.

Last but not least, I cannot be where I am today without my family. Their sacrifices and hard work have allowed me to focus on my education and pursue my interests, academic or otherwise. Their unconditional love and support have been a solid foundation and have helped push me through frustrating and tough times. I am eternally grateful for everything they have done for me. Thanks, and much love, to Ba Ngoai, Ba, Me, and Truong.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The World Wide Web has revolutionized many aspects of our daily lives; we can now communicate, learn, teach, work, socialize, etc. over the web platform. However, all these activities can reveal vast amounts of information about our individuality, preferences, and habits. This has attracted pervasive monitoring of our online activities by Internet Service Providers (ISPs) for commercial purposes and governments in the name of national security. This collection of data is clearly a violation of our privacy, but there are numerous other problems it presents: the data can be leaked or stolen, potentially causing personal and financial harm; it can be a threat to democracy if used by governments to squelch the freedom of expression and dissent.

In the past decade, the anonymity network Tor [12] has proven a powerful tool in the fight to protect Internet users' privacy: it provides unlinkability between a user's IP address and her online activities. However, Tor is not perfect, and due its popularity, researchers have studied and published many attacks against the network; AlSabah and Goldberg have compiled a survey of these attacks [4]. One class of traffic analysis attacks, called *website traffic fingerprinting*, enables a passive local adversary that can observe the network traffic generated by a user's web browsing session to determine which web pages the user is visiting.

Herrmann et al. were the first to present a website traffic fingerprinting attack against Tor, albeit with little success [17]. However, two years later, Panchenko et al. demonstrated a successful website traffic fingerprinting attack against Tor [30], prompting Tor developers to implement a number of network- and browser-level defenses into the Tor Browser Bundle [3]. The Tor Project cautioned that they did not expect those defenses to be foolproof; indeed, the following year, Cai et al. published a successful attack against Tor with those defenses in place [6]. In fact, the broader website traffic fingerprinting literature previously had been in a similar situation: proposed

defenses foiled specific known attacks—by hiding features exploited by said attacks—but were later found to provide little protection against a newer attack. With that in mind, researchers recently started to turn their attention towards *provably secure* defenses that can provide tunable levels of security against some theoretically optimal attacker.[1] What these provably secure defenses have in common is they use a pair of proxies to essentially provide a *single communication channel* through which the web browsing traffic is tunneled, enabling researchers to enforce strict rules on the observable properties of the channel—message sizes, counts, timing, and ordering—in order to analyze its theoretical security properties.

The first such provable defense is called BuFLO (**Bu**ffered **F**ixed-**L**ength **O**bfuscator), due to Dyer et al. [13]. The BuFLO channel sends fixed-length packets at a constant rate in both directions for a minimum duration, with padding as necessary; however, BuFLO incurs high bandwidth overheads and still leaks substantial information about the total sizes of web pages that require more than the fixed minimum duration to finish loading. Extending BuFLO, Cai et al. proposed CS-BuFLO to address several performance and practicality issues—including congestion insensitivity—while maintaining the security-overhead tradeoff [7]. Orthogonal to CS-BuFLO, Cai et al. also proposed another BuFLO extension called Tamaraw that achieves a significantly better security/efficiency tradeoff [8].

A second family of provable defenses achieves better performance than the BuFLO family by using *super-traces* that can cover clusters of packet traces of web pages to be protected [29, 34]. They rely on a training phase that obtains unprotected packet traces generated by visiting the web pages without the defense. Then, with this knowledge on the web pages to be protected, these defenses can compute super-traces tailored specifically for these specific pages, enabling them to reduce the overhead compared to the BuFLO-based defenses. During the online protection phase, when a web page is loaded, its traffic is morphed—with packet padding, splitting, delaying, etc.—to appear like its corresponding super-trace.

Because of their provable security properties, these defenses are compelling

[1] Most recently, Juarez et al. show that real-world settings—e.g., when the victim uses a different browser than the attacker or simultaneously loads two web pages—will significantly degrade many existing attacks [23]. While this result does cast doubt on the severity of the threat of website traffic fingerprinting, it does not eliminate the threat outright.

candidate solutions to address Tor's website traffic fingerprinting weakness. Indeed, in this dissertation work, we aim to adapt these defenses for use with Tor. Specifically, we will consider the BuFLO-based defenses: CS-BuFLO and Tamaraw. The super-trace-based defenses, on the other hand, are hindered by practical deployment issues including the required training phase that involves collecting many traces of the web pages and the intensive super-trace computation [29]; therefore, we will not consider them here.

1.1 Contributions

We explore a design that adds the defenses on top of Tor by using a pair of proxies to tunnel the BuFLO channel through Tor, with one proxy at either end of the Tor circuit. Then we implement the design and evaluate using live experiments and simulations. In particular, we seek to answer the following questions:

- **A deep dive into the web page loading process, and modeling it**

  We take a close yet incomprehensive look at the complex web page loading process. The lessons from this study help guide the creation of a framework to extract a model of the process and to generate network traffic according to the model.

- **Propose a design to use BuFLO-defenses to protect web traffic through Tor**

  - **How will single clients perform?**

    We want to understand how the high latencies and low bandwidth of the Tor environment affect the defenses. Also, this will in fact be the first study of Tamaraw in action, for the original paper only simulated the defense by applying transformations to unprotected packet traces to produce protected ones.

  - **How will the whole network behave?**

> As with any network protocol, it is crucial to understand how the broader network will behave with various levels of adoption. This is especially true for the Tor network, with many relays being run by individual users contributing their bandwidth to the network, because these defenses incur significant bandwidth overhead. Furthermore, the defenses also have an inherent time overhead. Therefore it is important to understand how the defenses will affect web browsing clients, the key user population of Tor.

Previous simulation studies, including those related to Tor, have used coarse models of web traffic, e.g., generate traffic according to certain distributions or represent a web page as a single file. Website traffic fingerprinting attacks and defenses, however, are designed to detect and conceal, respectively, tiny differences in the network profiles of different web pages; we must, therefore, create detailed models of web pages for our simulation studies. Furthermore, even with live experiments over the real Tor network, because real web pages change over time, it is also desirable to eliminate that source of variability by capturing snapshots of the pages for use in repeated experiments. In this thesis, we will thus also contribute a tool to create such web page models.

The rest of the thesis is organized as follows. In Chapter 2 we will provide background on Tor and website traffic fingerprinting. Next we discuss modeling of web pages in Chapter 3. We then describe provable website traffic fingerprinting defenses and a simple design to bring them to Tor in Chapter 4. Finally, we conclude in Chapter 5.

CHAPTER 2

BACKGROUND

2.1 Tor

Tor is a low-latency anonymity network that supports any TCP application. The network consists of proxies called Tor relays that register themselves with Tor's public central directory servers. Starting with just a handful of relays in 2005, today the network has grown to over 6000 relays, serving 2 million users at any given time . In order to use Tor, a client first downloads from the public Tor directory servers information (IP address, public key material, etc.) about available relays in the network. Then, to make a connection to a destination server, the client selects three[1] relays, setups a circuit through them, and connects to the destination server through the circuit. The three relays are called the *entry* (or *guard*), *middle*, and *exit* relays based on their positions on the circuit in the perspective of the client.

All communication between the client and the relays are protected using layered encryption, i.e., a relay on the circuit can read only messages between itself and the client, not those messages between the client and other hosts. This ensures that, except for the client, each host along the circuit knows of only the (IP addresses) of host(s) directly connected to it. For example, the middle relay knows of only the entry and exit relays, and the destination server knows of only the exit relay. It is important to note, though, that the exit relay can potentially see the plaintext traffic between the client and the destination server; that is, if the client chooses not to use end-to-end encryption (e.g., HTTPS) to protect its communication with the destination server.

To be suitable for interactive applications such as instant messaging and

[1] Though the client can configure shorter path lengths, the security of two-hop paths is an open question .

web browsing—the most important application carried over the network, at least evidenced by the number of connections and implicitly the number of users [27]—Tor relays do not intentionally delay traffic. In fact, the term "low-latency" is only with respect to other anonymity systems, such as Babel [15] and Mixminion [11], that achieve higher levels of security at the cost of inducing high and variable delays to messages. When compared to direct connections, Tor has significantly higher latencies because of the three additional hops on the path between the client and server, in addition to other factors such as network capacity and congestion. From ∼1 000 tests during a 24-hour period in November 2015, the RTT over Tor is 0.41 ±0.33 seconds.[2]

2.2 Website traffic fingerprinting

We now describe website traffic fingerprinting and survey the attacks and defenses in research literature.

2.2.1 Attacks

Website traffic fingerprinting, or website fingerprinting for short, refers to the attack where the attacker observes the encrypted channel over which a web browsing session takes place and attempts to identify visited websites by analyzing the traffic pattern. The encrypted channel can be a direct HTTPS connection between the browser and the web server, in which case, the identity of the website is already available (e.g., in the IP addresses of the connections); in these scenarios, the attacker attempts to identify the visited pages within the site [10]. The encrypted channel can be a connection (e.g., SSH or HTTPS) between the browser and a proxy, and the website is downloaded over the proxied connection. In these cases, the IP addresses of the destination web servers are not available to the attacker, making the fingerprinting attack harder but not impossible.

Since the content of the downloaded bytes is not available to the attacker, she relies on other observable features of the traffic, such as packet counts, directions, sizes, and inter-arrival times, etc. First, she builds "profiles" of web pages by browsing those pages and extracting the relevant features of the

[2] We ignore ∼15 samples that are over 5 seconds.

traffic her visits generate; this is the *training* phase, from which the attacker creates a classification model. Later, given a new *test* traffic trace, she uses machine learning classification algorithms and the created model to identify the website that generated the trace in question.

Two attacks scenarios are considered: the "closed-world" experiment where the entire world of web pages the client can visit is known *a priori*, and the attacker can train on all the pages; and the more realistic "open-world" experiment where the attacker optimizes her resources and/or techniques to correctly detect visits to a small set of "monitored" pages (for which she builds extensive profiles) while the client is allowed to also visit a much larger set of "unmonitored" pages (for which the attacker builds less extensive profiles). We will denote by M-v-U the size of an open-world experiment, where M and U are the sizes of the sets of monitored and unmonitored pages, respectively.

The earliest attacks in the literature are due to Hintz [18] and Sun et al. [31]. These rely on the strong assumption that individual requested resources can be differentiated in the captured traffic. This is possible, for example, when each resource is requested using a separate non-persistent HTTP connection or in a serial (non-pipelined) fashion on a persistent one. Knowing the number and approximate sizes of downloaded resources helps the attacker identify the loaded web page. In particular, for a 2 191-v-98 496 open-world experiment, Sun et al.'s attack based on the Jaccard similarity coefficient is able to achieve an accuracy rate of 75% with a false positive rate of 1.5%.

Bissias et al. are the first to present an attack on encrypted channels (SSH) that do not reveal individual resource sizes, achieving a 23% accuracy using cross-correlation of packet sizes and inter-arrival times in a 100-site closed-world experiment [5]. Liberatore and Levine consider unique packet lengths (with directions) in proposing two attacks, one using the Jaccard coefficient and the other using the Naive Bayes classifier [25]. For a closed-world attack on 1 000 pages, using only 4 training instances per page, the classifier based on the Jaccard coefficient achieves up to 73% accuracy, and the Naive Bayes classifier 67%. (False positive rates are not presented.)

Subsequently, Herrmann et al. [17] improved on the Naive Bayes attack by using a multinomial naive Bayes classifier and applying optimization techniques from text mining: term frequency transformation, inverse document

frequency transformation, and cosine normalization. Neglecting false positives, their attack achieves 94.2% success rate against SSH tunnels in a 775-site closed-world experiment; however, in a 78-by-697 open-world experiment optimized for minimum false positives—incorrectly classifying an unmonitored page as a monitored one—their technique achieves a false positive rate of 1.4% at the expense of dropping the true positive rate to 40%. Herrmann et al. also evaluate their attack against Tor, which presents the toughest challenge: they achieve a success rate of only 2.96%.

However, Panchenko et al. demonstrate a successful attack against Tor clients by using Support Vector Machines (SVM) [30]. For the same 775-page closed-world scenario and features used by Herrmann et al., simply using SVM improves the attack accuracy from 3% to 31%. With more features designed to capture traffic burstiness, main HTML document size, and ratios of incoming and outgoing packets, etc., they achieve a success rate of 54.6%, presenting a significant privacy threat for Tor web browsing clients. For a 5-by-5 000 open-world experiment where the attacker trains on 4 000 of the unmonitored sites but is tested on the other 1 000 sites, their true positive rates are 56-73%, and false positive rates are less than 1%. Dyer et al. show that a Naive Bayes classifier that uses only coarse-grained features of total bandwidth, total download time, and burst sizes, which are much simpler than Panchenko et al.'s SVM and advanced features, can achieve similar attack accuracies [13].

In 2012, Cai et al. further improve web page fingerprinting attack accuracy on Tor by training SVM on optimal string alignment distances (OSAD) between packet traces [6]. This distance metric captures packet ordering, which is useful because it reveals information about the order in which the browser requests embedded resources, and the up-down patterns of the traffic, which is a key contributor to their attack success. They are able to achieve an 83.7% success rate for a closed-world attack of 100 web pages, an improvement from the 65% success rate achieved by Panchenko et al.'s attack against the same data set. By using a modified OSAD metric as well as more advanced data processing, Wang and Goldberg improve Cai et al.'s attack accuracies to 91% from 87% and 96.9% from 86.9% for a 100-site closed-world experiment and a 4-v-860 open-world experiment, respectively [33].

Most recently, Wang et al. publishes an attack based on k-Nearest Neighbors that best Cai's method in open-world experiments [34]. They employ

a large feature set of 4 000 features (75% of which are for unique packet lengths); however, the features are assigned different weights that represent how well they are able to differentiate websites during the training phase. In a 100-by-5 000 open-world experiment, they achieve an 85% success rate at 0.006% false positive rate, compared to Wang and Goldberg's [33] 83% and 0.06%, respectively. Their 100-site closed-world results are not significantly different than Wang and Goldberg's.

2.2.2 Defenses

Fingerprinting defenses address the recognizable low-level features of encrypted web traffic. To obfuscate unique packet sizes, a common technique is to pad packets [13, 25, 31] or always send fixed-size packets. There are many possible padding schemes: pad to the maximum transmission unit (MTU), or to the next multiple of some fixed value, or with a random amount, etc. To obfuscate packet size distribution, Wright et al. propose traffic morphing [37]. This technique first learns the packet size distribution (the *target* distribution) generated by some web page. Later, when a different page is visited, the system will pad or split packets generated by the application such that their sizes appear to be drawn from the target distribution. Wright et al. demonstrate that their morphing technique achieves better security and overhead than a pad-to-MTU defense.

Panchenko et al. propose using background traffic to obfuscate the traffic pattern [30]. Called Camouflage, the idea is load a decoy web page in the background at the same time as the real requested page. Their experiments show that their closed-world attack accuracy on Tor drops to 3% from 54% at the expense of 85% bandwidth overhead.

Defenses can also be implemented at the HTTP application layer: modify the default HTTP behavior to induce changes at the network layer. A defense can split an original request into multiple requests by using range requests; randomize orders of requests when pipelining; inject dummy request headers that the server will ignore, etc. Luo et al. implement these obfuscation measures in HTTPOS, a client-side proxy [26], and report significant success defending against several attacks in a 100-site closed-world experiment. For example, for SSH tunnels, Liberatore and Levine's Naive Bayes classifier can

identify all of the 100 sites with at least 90% accuracy without HTTPOS; when HTTPOS is applied, 98 of the sites are fully unidentifiable to the classifier, i.e., 0% accuracy. Another implementation of randomized pipelining is the Tor Browser Bundle's customized version of Firefox [3].

Sun et al. [31] discuss using pipelining to achieve "two-chunk transfer" if all resources are on the same server: the first downloaded chunk is the page HTML, and the second is all the embedded resources downloaded together. This transfer scheme obscures the boundaries among the resources downloaded in the second chunk and thus hides their individual sizes. Furthermore, a "one-chunk" transfer, which further hides the size of the page HTML, can be achieved if a proxy downloads all files and sends to the client in one chunk.

After showing that known efficient packet padding-based defenses and Wright et al.'s morphing defense are still leaking coarse features like total time and bandwidth that their simple classifier can exploit, Dyer et al. propose the first provable defense BuFLO (Buffered Fixed-Length Obfuscator) to hide those coarse features [13]. BuFLO inspired other provably secure defenses: CS-BuFLO and Tamaraw; we will describe them in details in Section 4.1.

CHAPTER 3

FINE-GRAINED MODELING OF THE WEB PAGE LOADING PROCESS

In many modeling applications, researchers have used coarse-grained models of web pages, e.g., the total download size of a page [19, 22], the numbers and sizes of objects of a page, etc. This information is readily available from a packet capture on the network. However, web pages are becoming more complex and dynamic, resulting in ever more complex inter-dependencies among a page's embedded resources. These *dependency graphs* influence the timing of the browser's discovery of and issuing requests for the embedded resources, and thus play an increasingly larger role in determining the page load times. Yet, it is non-trivial to extract a web page's dependency graph from a network packet capture. As a result, as a part of efforts to optimize page load times, researchers have created tools to extract web pages' resource dependency graphs, with varying levels of precision [24, 28, 35].

Nevertheless, the focus of these previous efforts have been the web page, i.e., create the model of the web page. In our context, however, where website traffic fingerprinting algorithms are adept at picking out small differences in network traffic, we need to have precise models of the entire **web page loading process**. Figure 3.1 shows that conceptually, this process takes as inputs the web page, the browser,[1] and the network; its output is the sequence HTTP requests—or more generally the sequence of byte write requests—it makes to the underlying communication layer. This is the communication layer with which the process interacts directly; it could be the operation system's standard TCP/IP stack, or it could be a proxy channel, such as Tor, or BuFLO, that interacts with the operating system's TCP/IP stack. Note that the communication layer itself feeds back into the web page loading process by, for example, rejecting or rate-limiting the page loading process's write requests.

[1] this includes any user-specific customization, browser extensions, etc. that modify the behavior of the browser.

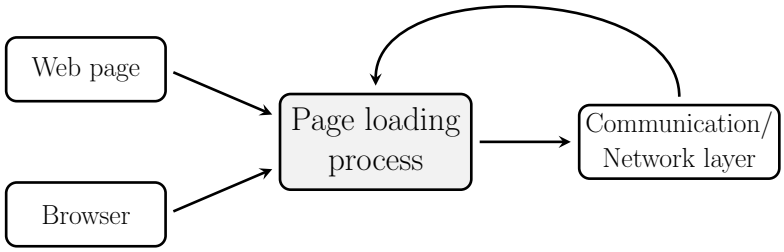


Figure 3.1: The web page, browser, and communication/network layer are inputs into the web page loading process, which then outputs a sequence of write requests into the underlying communication/network layer.

Therefore, in addition to modeling the web page, we need to also model the web browser. This means not only understanding the pertinent parts of web standards such as HTML, DOM, JavaScript, and XMLHttpRequest, etc.,[2] but also a browser's implementations of those standards, as well as its own browser-specific designs and idiosyncrasies. Specifically, we are working with the open-source Chromium browser[3] (at version 38.0.2125.122), which is at the core of Google Chrome,[4] which has a more than 50% share of the web browser market [2]. In the rest the thesis, we use Chromium and Google Chrome interchangeably, because the additional branding and features that Google Chrome adds on top of Chromium do not affect the core web page loading process.

Next, we will give a general overview of web page loading process, which is not specific to any web browser. We then take a closer look at the process, as well as some design and implementation details that are specific to Google Chrome. Since the full specifications of web standards are beyond the scope of this thesis, we assume the reader is somewhat familiar with these topics, and we only discuss aspects that are of interest to our modeling task.

3.1 The high-level mechanics of web page loading

Web pages A modern web page is conceptually a dynamic document described by an HTML file and associated external resources. First, the page's

[2] https://spec.whatwg.org/
[3] https://www.chromium.org/Home
[4] https://www.google.com/chrome/

content, e.g., text and images, etc., is primarily specified in the HTML markup. Second, its visual presentation or style, e.g., colors and fonts, etc., is specified in cascading style sheets (CSS), which can be embedded directly in the HTML markup (*internal* style sheets) or linked to other files (*external* style sheets). Finally, the page's dynamism and interactability are enabled by JavaScript: scripts can alter many aspects of the page's content and style as well as enable interactions with the user. Like style sheets, scripts can also embedded in the HTML file (*inline* scripts) or linked to other files (*external* scripts).

Browsers Notwithstanding differences in their feature sets and architectures, modern browsers share a common fundamental task: to render web pages. To perform this task, browsers contain at their core a standards compliant web layout engine that provides the foundation to display a web page on screen. The layout engine's key components that are relevant to our discussion are an *HTML parser*, a *CSS evaluator*, and a *JavaScript engine*. The layout engine delegates the task of fetching resources to a separate *resource loader*, which abstracts away the network and potentially the HTTP cache; the layout engine submits URL requests to and receives response data from the resource loader.

Loading a web page To load a web page, a browser first loads the page's HTML file by submitting a request to the resource loader. HTML data can be processed incrementally instead of requiring the entire file, and the HTML parser does just that: it parses the HTML response data as it becomes available and incrementally constructs a *Document Object Model (DOM)* to represent the content of the HTML. As the parser encounters external resources such as images and style sheets, it requests them with the resource loader and generally continues parsing and building the DOM without blocking on the responses.

Concurrently with the DOM construction, the browser also builds the *CSS Object Model (CSSOM)* when it encounters style sheets, either in-line or after fully receiving them from the resource loader. The CSSOM contains rules on how to visually style the page's content. Once the CSSOM has been fully constructed, the layout engine periodically combines it with the DOM, even if not fully constructed, to create a render tree to be displayed on the screen. Note that this operation might cause additional resources to be loaded (e.g.,

from the network) because some content elements are displayed with CSS styles that specify external fonts and/or background images. This represents a dependency relationship: these external resources can only be downloaded sometime after the style sheet referring to them has been.

As we have mentioned, scripts have the power to alter the page's content and styling: they have full read write access to the DOM and CSSOM. Additionally, using `document.write()`, a script can write HTML data directly into the current input of the HTML parser. Thus, there are potential race conditions where the HTML parser, CSS evaluator, and JavaScript engine access the DOM and CSSOM at the same time. And because generally scripts are executed at the exact location they are referenced in the document, the browser prevents the race conditions by adopting the following rules:

- Script download (if external) and execution block the HTML parser (though, this does not affect the resource loader's active resource downloads, including of the HTML file).
- Script execution is blocked if there is ongoing CSS activity including external style sheet downloads and CSSOM construction.

These rules introduce dependencies that can significantly slow the page load; for example, if a script takes a long time to download and/or execute, the HTML parser will take that much longer to parse the rest of the page, delaying discovery and download of embedded resources later in the HTML markup, etc. (To address some of the issues, the HTML spec has introduced an `async` attribute for external scripts: if marked `async`, their download and execution occur asynchronously with the rest of the page, i.e., they do not block the parser.) Furthermore, a script can cause the browser to load new resources, e.g., it can add a new image to the DOM, or it can change the style of an existing element to a previously unused style that defines a background image, requiring the browser to request that background image for the first time. In both cases, the newly requested image depends on the script, and in the latter example, also on the style sheet.

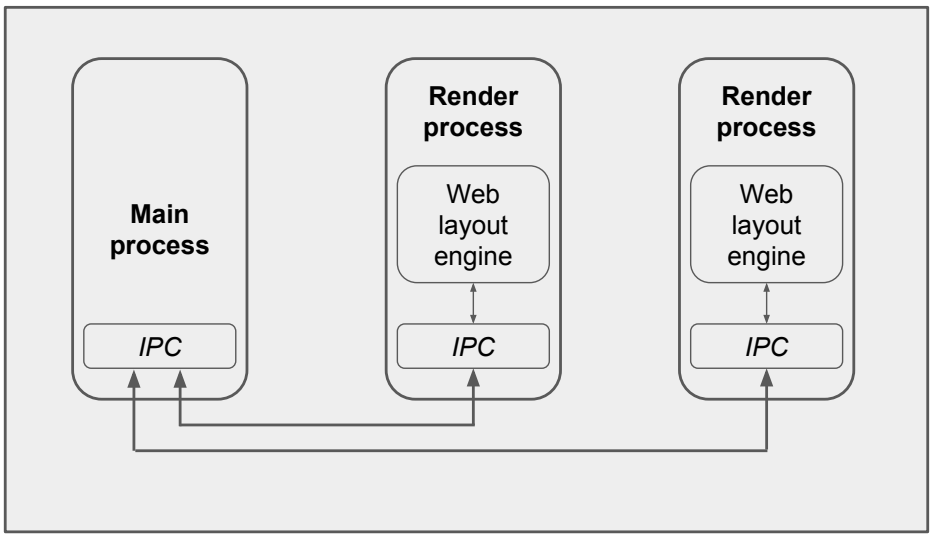


Figure 3.2: High level view of Google Chrome's multi-process architecture.

3.2 Deep dive into web page loading process

In this section we go into the details of the web page loading process as well as the Chrome browser implementations that are relevant to our modeling task. Since these topics are inherently intertwined, oftentimes we cannot avoid referring to concepts/components that we have not yet fully defined. We start with describing Google Chrome's architecture, as depicted in Figure 3.2.

3.2.1 Google Chrome architecture

The web layout engine is a CPU-bound component of a browser: parsing arbitrary HTML and CSS as well as executing JavaScript can take significant amount of time. In order to prevent these operations from blocking the browser's IO, for concurrency, browsers use a separate thread or process dedicated to network IO. Google Chrome uses a multi-processing architecture, where the **main process** (also called the **browser process**) handles IO, including the network and the disk cache, and each web page being loaded—either in a tab or window—is handled by a separate instance of the **render process**. An important advantage of this multi-processing architecture is isolation: critical errors in one tab or window does not affect other tabs or windows or the main browser.

The only part of the main process that is relevant for our task is the net-

work subsystem that handles network IO. This subsystem is responsible for creating and maintaining connections to web servers, implementing networking protocols such as HTTP, SPDY, etc., servicing the renderer's requests for resources.

In the renderer process, the main thread handles inter-process communication (IPC) with the browser process, while the web layout engine is run in another thread called render thread. The main thread provides an interface through which the layout engine can submit requests for resources and receive response data. The layout engine is Blink,[5] which is a fork of the popular layout engine WebKit.[6]

3.2.2 Layout engine processing model

The web layout engine handles many different things: HTML parsing and DOM construction, receiving data from the network, executing JavaScript, interacting with the user, rendering the page, etc. It uses an event-based processing model: events from these sources of work are placed into task queues, and the layout engine runs an event loop to process these tasks. An example task is when the main process sends to the render process an IPC message containing a new chunk of data for the page's HTML; the render process's main thread would queue a task on the layout engine's task queues to handle the new data. Another example is when the main process has finished receiving the HTTP response for a resource, it will send the render process a "ResourceFetchFinish" IPC message, which the render will add a task onto the queue to be processed.

The HTML standard specifies this fully,[7] but at a high level, what's relevant to us can be summarized as follows:

1. select a task from one of the task queues
2. run the task
3. update rendering

[5] http://www.chromium.org/blink
[6] https://webkit.org/
[7] https://html.spec.whatwg.org/multipage/webappapis.html#event-loop-processing-model

4. repeat

The use of multiple task queues allow the layout engine to prioritize certain tasks over others; for example, it might give higher priority to the task queues that contain mouse and keyboards events over other task queues.

3.2.3 Resources and HTML elements

As discussed previously, the layout engine uses the DOM as the internal representation of a web page. The DOM is a tree of "nodes". The root of the DOM corresponds to the `html` element in the HTML markup; other nodes in the tree correspond to other HTML elements in the page, as well as text nodes that are not elements but are children of other elements.

Resources, on the other hand, are not part of the DOM tree. Generally, a resource refers to some data that is used by the web page but is (usually) external from the HTML. A resource might not be associated with any HTML element at all, or it can be referenced by multiple elements; we will later see why this is relevant. Resources can be one of several types, such as HTML, image, script, font, etc. For example, the main HTML of the page is a resource ("main resource") that the layout engine has to download. Each resource is identified by a Uniform Resource Identifier (URI), which in the case of web pages, is most often a Uniform Resource Locator (URL) that specifies the remote location of the resource.

In the common case, a resource is fetched by sending a single HTTP GET request and receiving the response that fulfills the request. In other cases, additional requests are needed. For example, the response from the server can redirect the browser to request the resource at another URL. Suppose that the layout engine "follows" the redirect and makes another request at the new URL, this request itself can also be redirected, and so on. The sequence of HTTP requests made by the layout engine in order to fetch a resource is commonly referred to as the redirect chain. Another example is for `XmlHttpRequest`s, for which the layout engine might send an OPTIONS request before sending the actual request. These requests can also be redirected.

The layout engine keeps an in-memory "cache" of the resources, keyed by their URIs, and their data. This allows multiple elements that reference the

Listing 1 Simple web page at `http://example.com`

```
<html>
  <body>
    <img id='img1' src='cat.jpg'>
    <img id='img2' src='...'>
    <img id='img3' src='http://example.com/cat.jpg'>
  </body>
</html>
```

same resource URI to use the same resource, i.e., the layout engine does not have to fetch the same resource repeatedly for every reference. Note that this cache is in the renderer process and is separate from the memory-only HTTP cache maintained by the browser process and shared by multiple renderers.

A less commonly used URI is the Data URI, which contains the actual resource data itself, and thus the layout engine does not need to separately fetch the resource from the remote server and can just parse the data from the URI itself. This is useful for small resources, for which the HTTP request-response round-trip and potentially TCP connection setup can dwarf the data transfer time.

Consider a toy example web page at hypothetical site `http://example.com` with the HTML markup in Listing 1. In the body of the page, there are three image elements. Not counting the main HTML resource, what are the resources that the layout engine will load when loading this page? The two resources loaded by the layout engine are:

- The image resource `http://example.com/cat.jpg`: this resource is referenced by the two image elements `img1` and `img3`. In particular, element `img1` has a relative `src` attribute, and thus the layout engine determines that its absolute URI is `http://example.com/cat.jpg`, which is the same URI referred to by element `img3`. As mentioned above, the layout engine only needs to fetch this resource once, and both elements can be rendered using the same data.

- The image resource with a Data URI referred to by element `img2`; this resource does not require a network fetch.

Key relevant elements Since we are only interested in aspects of the web page that can induce network traffic and/or significant computation time,

we are mostly interested in these HTML elements: `img`, `link` (those that are style sheets), `script`, and `body` elements. (The `body` element is interesting because the layout engine will skip the rendering step if the parser has not encountered the `body` element.)

3.2.4 HTML parsing

The HTML resource (e.g., the main resource) data constitutes a stream of input bytes that is fed to the HTML parser. The parser consists of a module—the "tokenizer"—that extracts HTML tokens from the input stream and sends each extracted token to a separate module—the "tree builder"—to construct the DOM tree. For some elements such as images, in normal operation, insertions into the DOM tree immediately result in the layout engine's issuing requests to fetch the corresponding resources.

Incremental parsing The layout engine tries to parse whatever HTML data that has arrived; it does not wait until receiving the entirety of the main resource before parsing. This means that not only the relative ordering of elements on the page, but also the byte location of each of the key elements is an important property that we need to model because it affects the timing of the parser's discovery of the element, and thus the corresponding network request and/or computation. For example, consider the example page in Listing 2. The two image elements `topimg` and `bottomimg` are near the beginning and the end of the 100 KB HTML file, respectively. Due to incremental parsing, the layout engine will discover and request `top.jpg` almost immediately after starting to load the web page, regardless of the download bandwidth. On the other hand, the delay between the start of the page load and the layout engine's discovery of and requesting for `bottom.jpg` depends on the download bandwidth: the lower the bandwidth, the larger the delay. Therefore, it is important for the page models to capture this property, especially when loaded over Tor, where network quality is highly variable.

Preloading When the parser encounters a closing script tag (`</script>`), if the script is a regular script element (without `async` and `defer` attributes), then the layout engine pauses the parser, and thus DOM construction, until the script has executed. If the script is inline, the layout engine can immediately execute the script; since the layout engine runs in a single thread, this

Listing 2 Example web page illustrating the importance of the elements' locations. The two image elements are near the beginning and the end of an HTML file that is approximately 100 KB.

```
<html>
   <img id='topimg' src='top.jpg'>

   <!-- ... 100 KB of markup ... -->

   <img id='bottomimg' src='bottom.jpg'>
</html>
```

does not represent any idle time. If the script is external, however, while waiting for the script resource to download, the layout engine is essentially idle. To take advantage of this idle time, if there is more HTML data available following the script element, the parser gives the HTML to a *preload scanner*, which tokenizes the HTML and looks for external resources, such as images, style sheets, etc., to fetch. This way, once the script resource has finished loading and has finished executing, then parser resumes and encounters the subsequent resources, they will have been in-progress or might have even finished downloading.

Yielding Even if the entirety of the HTML data is available, the parser might not parse it in one go: due to the single-threaded nature of the layout engine, the parser self-imposes a time limit on each parsing session, and yields control back to the event loop, so that the layout engine can perform other tasks, such as a timer, or parsing a script-blocking CSS resource that has finished downloading, allowing any script blocked by the CSS to execute.

Background parser By default, chrome uses a separate thread to run a background parser. This parser is particularly useful when the render thread is blocked, e.g., waiting for a script to execute: the background parser can receive the HTML data that might be arriving during this time and tokenize it. Note that the background parser does not modify the DOM tree; it only sends parsed HTML tokens back to the render thread, which uses the HTML tokens in the DOM tree construction. Also note that, this is different than preloading scanning, which also tokenizes the HTML but only to look for resources to preload; these tokens are otherwise discarded.

Listing 3 Example page `http://example.com/` where JavaScript directly and indirectly induces resource fetches

```
1  <html>
2    <head>
3      <style>
4        @font-face { font-family: myfontface; src: url('myfont.woff'); }
5        .myclass { font-family: myfontface; }
6      </style>
7    </head>
8    <body>
9
10     <img id='myimg' src='orig-img.jpg'>
11     <div id='mydiv' class='myclass'>   </div>
12
13     <script>
14       document.getElementById('myimg').src = 'NEW-img.jpg';
15       document.getElementById('mydiv').innerHTML = 'some text!';
16     </script>
17   </body>
18 </html>
```

3.2.5 JavaScript

JavaScript is the main method to provide dynamism and interactibility for web pages. JavaScript code can modify the page in many ways. For example, since the DOM tree construction happens incrementally, JavaScript code can modify the current DOM tree at the time the script runs: it can add and remove elements, modify attributes of existing elements, etc. There are two main ways for web page authors to have their JavaScript code executed: script elements and event handlers.

Script elements By default, script elements are executed in the order they appear on the page. For more flexibility, the HTML standard recently allows script elements to specify `async` and/or `defer` attributes, which affect when a script can be executed and relaxes the ordering constraint. The possible ways JavaScript can interact with the web page and the layout engine are numerous and complex; it is not a goal of this project to study this area comprehensively.

Still, we discuss here an example web page in Listing 3 that illustrates a basic way and a subtle way that JavaScript can induce network fetch of resources. First, the page defines on line 4 a font face named `myfontface` that

uses the remote font resource at `http://example.com/myfont.woff`. Line 5 declares that any element that is of class `myclass` will be rendered using the font face specified in `myfontface`. In the body of the page, the image element `myimg` refers to resource `http://example.com/orig-img.jpg`, which the layout engine will fetch. The body of the div[8] element `mydiv`, which is of class `myclass`, contains no text, only a few blank spaces; since there is no text to be displayed the font resource is not needed and thus not fetched. Next, we encounter the script element on lines 13-16. When executed, the script does two things:

1. modifies the image element's `src` attribute; the image element now refers to another resource `http://example.com/NEW-img.jpg`, which means the layout engine initiates a fetch for the new resource, while the JavaScript engine is still running.

2. adds the text `some text!` to the div element. This action does not immediately cause any resource fetches but does invalidate the style of the div element. In a subsequent *style recomputation* step, *after* the JavaScript engine has finished running, the layout engine updates the style of the div and determines that `some text!` needs to be rendered with the font face `myfontface`, causing the layout engine to initiate a fetch for font resource `http://example.com/myfont.woff`. Effectively, the script has indirectly induced network traffic, i.e., the font resource depends on the script element.

Event handling JavaScript code can be registered to execute when certain events occur, e.g., as part of the page load, or in response to user's actions (such as clicking on a button, entering text in an input field, etc.). *Event listeners*, as they are called, can be specified in HTML markup, and they can be added and removed in JavaScript code. Two key events associated with the page load are:

- `DOMContentLoaded`: the layout engine fires this event when the parser has finished parsing the page's HTML markup and constructing the DOM. This does not depend on the status of embedded resources: it might be the case that all embedded resources are still being downloaded.

[8] A `div` tag defines a section of the document

Listing 4 Example page to illustrate JavaScript event handling

```
1  <html>
2    <body>
3      <script>
4        function domLoaded() { console.log("Done parsing markup"); }
5        function pageLoaded() { console.log("Done loading page"); }
6
7        document.addEventListener("DOMContentLoaded", domLoaded);
8        window.onload = pageLoaded;
9
10       function img1LoadSuccessHandler() {
11         console.log("img1 loads successfully");
12         var img = document.getElementById("img1");
13         img.src = "non-existent.png";
14         img.addEventListener("error", img1ErrorHandler);
15       }
16
17       function img1ErrorHandler() { console.log("img1 error (JS)"); }
18     </script>
19
20     <img id="img1" src="good.png"
21          onload="img1LoadSuccessHandler()"
22          onerror="console.log('img1 error (markup)')" >
23
24   </body>
25 </html>
```

- `load`: the layout engine fires this event when there are no more pending requests embedded resources (`pendingRequestCount == 0`), i.e., each embedded resource has finished loading, whether successfully or not. In other words, as far as the layout engine is concerned, the initial page load is considered complete at this point. At the same, not all requests are taken into consideration by the layout engine (i.e., the request does not affect `pendingRequestCount`) when determining whether to fire this event, as we will see in the example below.

Consider the example web page in Listing 4. First, `console.log()` is a built-in JavaScript function that logs the message to the browser's JavaScript console.

- On lines 7 and 8, `domLoaded()` and `pageLoaded()` functions are registered to log the `DOMContentLoaded` and `load` events, respectively.

Listing 5 JavaScript console output when browser loads page in Listing 4

```
1 Done parsing markup
2 img1 loads successfully
3 Done loading page
4 img1 error (markup)
5 img1 error (JS)
```

- On line 21, in HTML markup, using the `onload` attribute, we specify that the `img1LoadSuccessHandler()` function will be called when the image successfully loads.
- On line 22, similarly, in HTML markup, we register a listener to log an error message to the JavaScript console when the image fails to load.
- When the image element successfully loads, we log to the JavaScript console, and then also modify its `src` attribute to point to a non-existent image (line 13) and register an *additional* listener for when the image fails to load (line 14).

When the browser loads the web page, assuming that the image `good.png` loads successfully, the sequence of events logged to the JavaScript console is shown in Listing 5. After initiating a fetch for image `good.png`, while it is still downloading, the parser reaches the end of the HTML markup and is thus finished parsing; at this point, the layout engine fires `DOMContentLoaded` and `Done parsing markup` is logged. When the image finishes loading, its `load` event listener is called and logs `img1 loads successfully` as well as modifies its `src` attribute, which will cause the layout engine to schedule a request for the non-existent resource. This request, however, does not increment`pendingRequestCount`, and thus at this time, the page is considered finished—no more pending resources—and the layout engine fires the page's `load` event. Soon after, the request for `non-existent.jpg` comes back with an error response, and the two event listeners registered for the image's `error` event are executed and log the corresponding messages.

In the following section 3.3, we will discuss how we produce a model of the page loading process. Then in section 3.4, we describe the design and implementation of our traffic generator.

3.2.6 When is a page load done?

As discussed earlier, at the moment the layout engine fires the `load` event, there might still be pending requests being downloaded. Furthermore, at or after the time of the `load` event, JavaScript can create additional requests that are very much part of the initial page load. For example, of the ∼9 400 page loads we performed on 94 of the top 100 Alexa-ranked web pages (100 loads per page) early November 2016, in over 14% of the page loads, we saw at least one request being started after the page `load` event. It is plausible that more page loads had pending requests that were started before the `load` event. This means that the firing of the `load` event is not a reliable method for website traffic fingerprinting defenses to be certain that the page load is truly done. Therefore, in our experiments, we rely on heuristics to determine when a web page is done loading: when an idle period of two seconds has elapsed without any pending request after the page's `load` event.

3.3 Extracting the model of the web page loading process

To extract a model of the web page loading process, we instrument the Chrome browser to log key information during the page load, and implement a tool to process the log to produce the model. There are other less intrusive methods to extract models of web pages without modifying the browser, but as mentioned before, for our modeling task, fine-grained details matter, so we need to instrument the browser in order to gain visibility into the intricate operations of a browser while it loads the web page. (The WProf authors took this approach as well [35].)

At the most abstract level, our objective is to generate the right amount of network traffic at the right moment in time. Resources and the requests needed to fetch them are the primary source of network traffic,[9] so we need to model whatever components and activities of the web page and browser that cause resources to be fetched, as well as the dependencies and timing of such activities. Parts of the page that can cause resources to be fetched are DOM

[9] Web sockets are another source of network traffic, but they enable the web page to communicate non-HTTP application-specific data, so it is beyond the scope of this thesis to model these connections.

elements (which are either parsed from the page's HTML markup, or can be added or modified by JavaScript); style calculations (as part of rendering updates, and which can fetch font and image resources); JavaScript execution, which can add or modify DOM elements, as well as create `XmlHttpRequest`s.

Our models contain information about the **entities** on the web page such as elements, resources, timers, etc., and for some of those entities, such as script elements and timers, what they "do." In other words, we also model *computations* since they contribute to the timing of the page load. For example, when a script runs, how long does it take? And what relevant operations does it perform? Does it create resources and initiate network fetches? Does it schedule timers? etc. These same questions are applicable to other event listeners, and timer expiration handling functions, and so on. We generalize this to a concept we call **execution scopes** to model computations. The simplest execution scope represents just a delay, i.e., it does not create any resources, schedule any timers, etc. We will discuss execution scopes in more detail, but first we go over how we instrument Chrome to handle other relevant aspects of the page load.

3.3.1 Instrumenting Chrome

Resources: For each web page load, the resources in memory are given unique identifiers (monotonically increasing *instance numbers*). Note that resources with the same URI might be different resources; for example, an HTTP OPTIONS request for `http://example.com/path` and an HTTP GET request for `http://example.com/path` are two different resources. We are able to detect this since we are operating in the Chrome code. Essentially the two resources have two unique identifiers. In general, the layout engine submits requests for the resources to the browser process's network subsystem, which is not aware of the resource identifiers used by the layout engine. In order for us to associate a resource in the layout engine with the chain of HTTP requests performed by the network subsystem, we propagate the resource's identifier down into the network subsystem, which can log the resource's identifier when logging the HTTP requests, including the hostname and port, and the total amounts of upstream bytes and downstream bytes.

Not all resources contribute to the `pendingRequestCount` that is part of

the check to see if the page has been loaded. We can also log this information for each resource that is fetched. For style sheet resources, we also log how long it takes to parse the style sheet; this can be a significant amount time, with the main style sheet often taking tens and sometimes more than a hundred milliseconds to parse.

Elements: Instrumenting the layout engine's HTML parser, we can see the relevant attribute of each element of the main page's HTML markup that refers to resources, e.g., `src` attribute for image and script elements, `href` attribute for link elements that are external style sheet elements (i.e., `rel` attribute is `stylesheet`), and so on. We log the byte locations within the main page's HTML of the key HTML elements such as images, style sheets, the `body` element, etc. These are byte locations within the uncompressed HTML data, which most likely do not accurately correspond to their locations within the compressed HTTP response, which is what the network subsystem deals with. For simplicity, not having to implement decompression, we use proportional mapping between the two set of offsets. For example, if the uncompressed HTML data and the compressed HTTP response are 1000 and 200 bytes respectively, then an element at location 700 in the HTML data is assumed to be at location 140 in the HTTP response, i.e., when the layout engine's HTML parser reaches the 140th byte of its input stream, then it encounters the said element. We also log the execution scopes that run when the element's `load` event fires.

Besides being created by the parser when parsing the page's HTML resource, elements can also be created by the JavaScript engine using other methods such as:

- JavaScript code can use DOM API such as `document.createElement()` to create an element and then insert it into the DOM.
- JavaScript code can insert HTML markup into the parser's input stream with `document.write()`, or insert *HTML fragments* using the DOM API such as assigning or appending to an element's `innerHTML` attribute; in either case, the created elements, though directly created by parser, are created while the JavaScript engine is active, so for simplicity we can consider them to be created during an execution scope of the script code.

For these elements, which are not from the page's HTML markup, we do not need their byte offsets, as they are not relevant. Elements are also identified by their unique monotonically increasing instance numbers, though from a difference space than resources. (All elements, documents, `XmlHttpRequest`s, etc.—entities that can have DOM events—share the same (*event target*) instance number space.)

CSS style sheets: Not all style sheets block JavaScript, e.g., a style sheet that is only used for laying out the page in "print" view. Instrumenting the style engine we can log this information for style sheet elements.

`XmlHttpRequest`s: `XmlHttpRequest`s are similar to elements in that they refer to resources and thus induce network requests, and they also fire the `load` and `error` events that JavaScript can listen to. Different than elements, the layout engine also supports notifying applications of fine-grained progress for `XmlHttpRequest`s, including when the response headers have been received, and whenever a chunk of response data has been received. Besides logging about the `load` event listener, we also log about these fine-grained events if there are listeners registered for them. `XmlHttpRequest`s by default are asynchronous, i.e., the layout engine can perform other tasks while waiting for the resource to download; they can, however, be configured to be synchronous, which means the JavaScript engine and consequently the entire layout engine block until the resource request finishes or errors. Synchronous `XmlHttpRequest`s are not recommended and thus should be rare; therefore we do not handle these `XmlHttpRequest`s.

Execution scopes: Execution scopes allow us to logically link the operations performed by the layout engine during some interesting/relevant sequence of activities. They generally correspond to "block scopes" and "function scopes" in C/C++ programming, but only for logging purposes. An execution scope, when created, will log its unique identifier and the current timestamp (currently we use the timestamp at construction as the identifier because of it has sufficiently high resolution, and the fact that the layout engine runs in a single thread); when destructed, the scope will also log the timestamp. In fact, all log messages associated with an execution scope are marked up with the scope's identifier and the current timestamp. The execution scope identifiers are most useful in cases of recursion.

Listing 6 Example code to illustrate execution scopes logging.

```
void functionA(int x)
{
  int tmp = 0; // some dummy statement
  tmp = x + 1; // some dummy statement

  ExecutionScope scope("msg for outer scope");

  scope.logMsg("x is %d", x);
  if (x == 0) {
    scope.logMsg("reached base case! no more recursion.");
    return;
  } else {
    ExecutionScope scope("this is inner scope");
    functionA(x - 1);
  }
}
```

Listing 7 Hypothetical log output of example code in Listing 6 when `functionA(1)` is called. `ts` is the current system time in seconds, and `scopeStart` identifies the execution scope that logs the message

```
ts= 1234.320: scopeStart:1234.320: msg for outer scope: begin
ts= 1234.350: scopeStart:1234.320: x is 1
ts= 1234.400: scopeStart:1234.400: this is inner scope: begin
ts= 1234.525: scopeStart:1234.525: msg for outer scope: begin
ts= 1234.555: scopeStart:1234.525: x is 0
ts= 1234.560: scopeStart:1234.525: reached base case! no more recursion.
ts= 1234.570: scopeStart:1234.525: msg for outer scope: done (dur= 0.055)
ts= 1234.610: scopeStart:1234.400: this is inner scope: done (dur= 0.210)
ts= 1234.630: scopeStart:1234.320: msg for outer scope: done (dur= 0.310)
```

Consider the example code in Listing 6, that shows a recursive `functionA()` using an execution scope for logging a function scope as well as a block ("inner") scope. The (simplified) hypothetical log output when `functionA(1)` is called is shown in Listing 7. The `ts` field is the current system time in seconds, and the `scopeStart` field identifies the execution scope that logs the message. The `begin` and `done` lines are automatically logged by the execution scope, enabling us to reliably know exactly when the layout engine enters and exits the scope.

For brevity, we have left out of the log output the other key information that is logged with each line: the source filename, and function name, and the line number. This information is usually sufficiently unique for us to determine during the log processing phase what kind of execution scope we are

processing; for example, if this is a execution scope for running a JavaScript element, or for processing a resource that finished downloading, etc. This, along with the automatically logged messages on scope entrances and exits simplify the task of recursion detection. It is simple to spot the recursion from the log of this simple example, but in general recursive calls are harder to detect when many other activities happen and messages are logged in between the recursive calls. Detection of recursive calls enable us to make certain assertions during the log processing phase; for example, we might not support web pages where a certain function recurses more than, say, two levels, because that could imply our lack of full understanding of the function's usage, or suggest the complexity of the web page is beyond our current ability to reasonably model it.

The `ts` fields enable us to determine the duration of the execution scope, and when the various interesting activities happen relative to the beginning of the execution scope. For example, we can say, "*this script element E takes 100 milliseconds to run, and it creates and causes the fetch of resource R 15 milliseconds after it starts.*"

3.3.2 Constructing the model from the log

To construct the model of a web page, we first load the page using our instrumented Chrome browser. Because we have not instrumented the background HTML parser, we disable it during all of our page loads. We also disable the use of SPDY[10] in Chrome because we have not implemented SPDY in our traffic generator (discussed in Section 3.4).

We have developed a Python program—we call it the **model extractor**—to parse the log produced by our instrumented Chrome browser to produce a combined model of the web page and the loading process. The objective of the model extractor is to automate the process of model extraction, so that we can scale this study to hundreds if not thousands of web pages. However, this is a challenging task, for the very simple reason that browsers as well as web standards are complex, and our understanding and coverage of them have only merely scratched the surface. We liberally add assertions in the instrumented browser as well as the model extractor to protect against

[10] https://developers.google.com/speed/spdy/

unexpected code paths, conditions, and uncovered features, and many real web pages fail at one or more of these assertions.

Specifically, we do not support *web workers*, which enable long running JavaScript code to be run in a separate thread; *message channels*, which enable communication among different frames, web workers, etc. on the page; *web sockets*, which allow the page to create full-duplex connections to and communicate application-specific (non-HTTP) data with a remote server; CSS imports; request priorities; policies regarding preloaded requests (e.g., preloaded requests have low priorities, and limits on the number of concurrent preloaded requests); abortion of `XmlHttpRequests`; shadow DOM tree; iframes; `async` or `defer` scripts, etc.

Example model: It is best to describe our web page loading models using a concrete example. In Listing 8, we show an example model of http://www.wikipedia.org that has been simplified for brevity. There are several "sections" in the model: main HTML (lines 39-49), elements (lines 2-19), resources (lines 50-72), and execution scopes (lines 20-37).

- For the page's main resource (always resource 1), we see there is only one request in the request chain (i.e., no redirects); the HTTP request to `www.wikipedia.org:443` is 334 bytes, and the HTTP response is 19379 bytes.

- The main page (`"main_html"`) contains 4 elements—element 2222 is not part of the page's HTML, as we will see below. Each element in the `element_byte_offsets` array is a tuple of byte offset and element identifier, e.g., at byte offset 19278, the HTML parser encounters the closing tag `</script>` and constructs element 2185.

  The execution scopes for the document's `DOMContentLoaded` and `load` events are 111 and 129 respectively.

- Each resource's `part_of_page_loaded_check` field specifies whether it contributes to the `pendingRequestcount`.

- Each script element has `run_scope_id` that specifies the execution scope that captures the execution of the script. For simplicity, we also include in this scope the time it takes to compile the script before it is run.

Listing 8 A simplified version of a model of http://www.wikipedia.org/

```
{
  "elements": {
    "22": { "exec_immediately": true,
            "is_parser_blocking": false,
            "run_scope_id": 9,
            "tag": "script"
    },
    "44": { "tag": "body" },
    "48": { "initial_resInstNum": 2,
            "tag": "img"
    },
    "2185": { "exec_immediately": false,
              "initial_resInstNum": 4,
              "is_parser_blocking": true,
              "run_scope_id": 101,
              "tag": "script"
    },
    "2222": { "tag": "img" }
  },
  "exec_scopes": {
    "9": [ "  __msleep(3.80);" ],

    "17": [ "  __msleep(2.85);" ],

    "101": [ "  __msleep(13.69);",
             "  __msleep(3.68);" ],

    "111": [ "  __msleep(7.40);",
             "  add_elem(2222);",
             "  __msleep(0.68);",
             "  sched_render_update_scope(124);",
             "  __msleep(0.85);" ],

    "124": [ "  __msleep(2.82);" ],

    "129": [ "  __msleep(0.02);" ]
  },
  "initial_render_tree_update_scope_id": 17,
  "main_html": {
    "element_byte_offsets": [ [ 106, 22 ],
                              [ 9843, 44 ],
                              [ 9932, 48 ],
                              ...
                              [ 19278, 2185 ]
    ],
    "event_handling_scopes": [ [ "DOMContentLoaded", 111 ],
                               [ "load", 129 ]
    ]
  },
  "resources": {
    "1": {
      "part_of_page_loaded_check": false,
      "req_chain": [
        {
          "host": "www.wikipedia.org",
          "method": "GET",
          "port": 443,
          "req_total_size": 334,
          "resp_body_size": 19379,
        }
      ],
      "type": "main"
    },
    "2": {
      "part_of_page_loaded_check": true,
      ...
    },
    "4": {
      "part_of_page_loaded_check": true,
      ...
    }
  },
}
```

- If an element has `initial_resInstNum`, then that refers to a resource

identifier, and it means this reference is parsed from the page's HTML markup, e.g., `src` attribute for images and scripts. This field is not specified for inline script element 22 and also for image element 2222, which is added by JavaScript.

As previously discussed, each execution scope captures a block of computation in the browser. In our models, each execution scope is represented as a snippet of source code, containing a list of commands for the layout engine to interpret. Take execution scope 111 (line 28) for example. The first command is `__msleep(7.40)`, which instructs the layout engine to sleep for 7.40 milliseconds. The second command is `add_elem(2222)`, which instructs the layout engine to add image element 2222 into the DOM; note that element 2222 never references any resource in this example. The fourth command, `sched_render_update_scope(124)`, instructs the layout engine to schedule the render tree update execution scope 124, which will be executed at the next opportunity, i.e., step 3 of the layout engine processing model (Section 3.2.2). The other supported execution scope commands are:

- `fetch_res(resInstNum)`: the layout engine will initiate a fetch for the resource with identifier `resInstNum` if the resource is not already being fetched or available. The `resInstNum` must be a known resource in the model. This is used, for example, when a font resource is fetched during a render tree update.

- `set_elem_res(elemInstNum, resInstNum)`: the layout engine makes the element `elemInstNum` reference the resource `resInstNum`. Both `elemInstNum` and `resInstNum` must be known in the model. The engine will start fetching the resource if it has not been fetched. This is used, for example, when JavaScript code modifies (or adds) the `src` attribute of an image element.

- `send_xhr(xhrInstNum)`: the layout engine will start sending the asynchronous `XmlHttpRequest xhrInstNum`, which must be known in the model.

- `start_timer(timerID)`, `cancel_timer(timerID)`: the layout engine will start or stop the DOM timer with identifier `timerID`, which must be known in the model.

Not illustrated in the example above are DOM timers, `XmlHttpRequest`s, CSS style sheets, and event handling scopes associated with elements, such as images, CSS style sheets, and scripts. Also, note that the model does not specify anything about preloading: the preloading logic will be entirely implemented in the layout engine.

3.4 Traffic generator

To generate network traffic, we need a pair of cooperative applications: the web server and the browser.

3.4.1 Browser simulator

As mentioned earlier, an important advantage of Chrome's multi-processing architecture is isolation among different browser windows and tabs. For our purposes, since we are not a full-fledged browser—no support for multiple concurrent web pages needed—there is no need for isolation, which means that a multi-threading architecture is more desirable because we would not need to use IPC, which can save development time. However, it is important for us to be able to use the same code base as a native application as well as a Shadow plugin; this leaves us with only one option: to use a multi-processing architecture. This is because Shadow does not fully support multi-threading: many of the threading system calls are not supported. Moreover, some threading system calls are missed by Shadow—i.e., Shadow does not intercept them—and thus will fail silently when used by a plugin. We discover this after trying to use `folly`,[11] a popular open-source C++ library developed at Facebook, that provides many functionalities useful for networking, threading, as well as event-based programming. Concerned that we might lose significant time investing into using such libraries only to find out that more system calls are missed or incorrectly simulated by Shadow, we decide that we cannot use multiple threads and have to use a multi-processing architecture.

[11] https://github.com/facebook/folly

Furthermore, we also discover issues with Shadow's handling of some aspects of C++, as well as a buggy simulation of the `writev()` system call,[12] thus rendering even the networking parts of the popular library `libevent`,[13] a relatively low-level library, suspect. Due to the uncertainties, we have to expend a significant amount of engineering effort re-inventing the wheel, starting with writing our own networking library on top POSIX sockets, so that we use only the minimal set of system calls that can get the work done, to reduce our exposure to issues in Shadow. Then we implement our own IPC on top of our networking library; again, we cannot be confident that other IPC frameworks, such as Apache Thrift[14] and Google gRPC,[15] which are multi-threaded, will work correctly in Shadow.

Similar to Chrome, the browser simulator uses an **IO process** to handle network IO on behalf of the web layout engine, which runs inside a **render process**. Unlike Chrome, however, the browser simulator's processes cannot run separate threads to handle IPC; all of our processes use `libevent` for the event loop. Each process has a single event loop to handle all events: for IPC and networking-related events for the IO process, and for IPC and web layout engine's processing model (3.2.2) for the render process. Also, similar to Chrome's Debugging Protocol,[16] the render process exposes an API that allows another process to control it via IPC: our **controller process** uses this API to send commands to the render process and to receive notifications of various events that the renderer fires, such as when a request is about to be sent and when it is finished, and when the `load` event is fired. To handle execution scopes, we embed the AngelScript engine[17] in the render process to execute the execution scope commands by calling back into the render process' functions.

Because of bugs in our page models or browser simulator, or both, sometimes some resources in a page model are not fetched. We implement a work-around for this by allowing the render process to force the loads of these resources when the layout engine is about to go idle without any scheduled activity, i.e., has finished parsing and there are no pending requests and no

[12] https://github.com/shadow/shadow/issues/317
[13] http://libevent.org/
[14] https://thrift.apache.org/
[15] http://www.grpc.io/
[16] https://developer.chrome.com/devtools/docs/debugger-protocol
[17] http://www.angelcode.com/angelscript/

pending timers. This ensures that our traffic generator at least generates the expected amount of network traffic, even though the timing is likely way off.

For IPC, it is often easier to use more verbose and readable formats such as JSON for message serialization, but for better performance, a popular library used for serializing messages is Google's Protocol Buffers,[18] which supports binary message formats. However, since we want to run Shadow simulations with as many nodes as possible, we use `flatbuffers`,[19] an even faster and more memory efficient relative of Protocol Buffers.

Our common C++ library implementation, including classes for networking, IPC, etc. are 3 200 lines of code. The IO process and the render process total more than 7 000 lines of code, and the controller process is about 1 400 lines of code.

3.4.2 Server simulator

The server's job is fairly straightforward: it receives requests from the client that specify how much data to send back, and it can just send back dummy data, since we're only concerned with the amount of traffic, not the content. Note that we are not modeling timing at the server, i.e., we assume the server sends back data as fast as possible, not exhibiting unique timing patterns due to application logic.

3.5 Extracting models of top Alexa-ranked web pages

In the website traffic fingerprinting literature, experimental studies are performed on datasets containing hundreds if not thousands of the most popular web pages from the Alexa rankings.[20] Ideally we can match or approach that scale. However, as we started to dive deep into the web page loading process and began to better understand its complexity when we tried to construct the model of the very first test web page, `http://www.berkeley.edu`, which is a simple web page relative to many of the top Alexa-ranked sites, it became clear that we had to make a tradeoff call: quality versus quantity. We

[18] https://developers.google.com/protocol-buffers/
[19] https://google.github.io/flatbuffers/
[20] http://www.alexa.com/topsites

could construct thousands of low-quality page models, or focus on extracting higher-quality models of a smaller number of websites. With website traffic fingerprinting, small differences in traffic patterns matter, so the quality of the page models is more important than in other contexts.

Since several earlier papers have used 100 web pages for the closed-world experiments, we decided to extract the models of the top 100 web pages from the Alexa global ranking. It turned out that accomplishing even that humble goal is a challenge. Some of these pages use web workers or message channels, which our tools do not support. Furthermore, most of the other web pages cause our tools to fail the numerous assertions we have added, in either the Chrome code or in the model extractor. For each individual case, we manually inspected the log to understand the root cause, and then we either relaxed the requirements that the code was insisting on, or sometimes removed the assertion altogether. In some cases, our fix would result in the code hitting other errors, and so on. Overall, we feel the tools have not been able to automate the page model extraction process, and, given the complexity, they are most likely not usable by those without intimate knowledge of the web page loading process in general and Chrome's design and implementation in particular.[21]

In the end we settled with 50 page models, the absolute minimum number of page models we considered acceptable (Hayes and Danezis's k-fingerprinting attack uses 50 web pages for one of their closed-world experiments). In Table 3.1, we list the web pages, and the various statistics about the page model that we extracted and the 100 load instances of each of the real web pages, which we will discuss next in the validation section.

Table 3.1: Statistics comparing the 50 page models to 100 load instances of the corresponding real web page.

	Real page (100 loads)		Page model	
Page	Up size	Down size	Up size	Down size
`ameblo.jp`	38.7 MB	2.9 MB	32.5 MB	3.7 MB
`aws.amazon.com`	90.3 MB	1.9 MB	88.3 MB	2.0 MB
`diply.com`	15.4 MB	145.9 MB	24.9 MB	692.0 MB
`english.china.com`	49.2 MB	943.0 MB	35.4 MB	973.8 MB

[21] We suspect this is one reason why WProf authors released their Chrome patches but not the tool that analyzes the logs to produce the page models.

Table 3.1 cont'd

go.com	21.5 MB	1.6 MB	10.8 MB	1.6 MB
imgur.com	55.5 MB	1.1 MB	46.2 MB	1.2 MB
mail.ru	64.1 MB	316.3 MB	29.9 MB	269.7 MB
stackoverflow.com	14.2 MB	298.2 MB	8.7 MB	289.5 MB
twitter.com	48.2 MB	2.7 MB	25.9 MB	3.8 MB
vk.com	24.3 MB	835.2 MB	14.8 MB	796.1 MB
wordpress.com	26.6 MB	288.5 MB	8.7 MB	256.9 MB
www.360.com	92.8 MB	2.4 MB	51.1 MB	1.9 MB
www.alibaba.com	14.5 MB	567.0 MB	12.6 MB	573.7 MB
www.aliexpress.com	62.9 MB	5.9 MB	26.7 MB	6.3 MB
www.amazon.com	29.6 MB	509.1 MB	31.4 MB	500.9 MB
www.ask.com	7.0 MB	69.8 MB	6.3 MB	67.6 MB
www.baidu.com	7.6 MB	162.6 MB	3.9 MB	157.2 MB
www.bankofamerica.com	27.7 MB	313.8 MB	130.9 MB	492.1 MB
www.bing.com	5.5 MB	46.4 MB	2.9 MB	42.4 MB
www.booking.com	29.9 MB	1.1 MB	28.2 MB	1.6 MB
www.buzzfeed.com	13.1 MB	291.1 MB	4.6 MB	344.6 MB
www.chase.com	27.6 MB	482.5 MB	12.7 MB	449.9 MB
www.cnn.com	18.3 MB	863.8 MB	47.5 MB	1.5 MB
www.craigslist.org/about/sites	12.1 MB	218.2 MB	8.0 MB	963.6 MB
www.dropbox.com	51.3 MB	736.9 MB	22.7 MB	690.6 MB
www.ebay.com	44.6 MB	438.4 MB	66.0 MB	2.1 MB
www.google.com	8.4 MB	189.2 MB	2.3 MB	179.4 MB
www.hao123.com	69.1 MB	933.1 MB	35.1 MB	1.3 MB
www.indeed.com	7.1 MB	24.0 MB	7.4 MB	107.7 MB
www.instagram.com	6.9 MB	205.0 MB	8.2 MB	583.4 MB
www.jd.com	6.6 MB	230.6 MB	5.6 MB	253.6 MB
www.linkedin.com	21.0 MB	295.2 MB	14.1 MB	301.4 MB
www.microsoft.com/en/us	43.6 MB	924.1 MB	33.3 MB	897.9 MB
www.nytimes.com	46.1 MB	609.7 MB	29.8 MB	747.6 MB
www.office.com	33.3 MB	412.1 MB	19.1 MB	624.1 MB
www.ok.ru	49.0 MB	732.8 MB	23.8 MB	682.3 MB
www.outbrain.com	41.8 MB	1.2 MB	25.1 MB	1.2 MB

Table 3.1 cont'd

www.paypal.com/home	28.5 MB	991.4 MB	12.5 MB	955.6 MB
www.pixnet.net	113.9 MB	3.1 MB	58.0 MB	3.3 MB
www.popads.net	25.0 MB	232.5 MB	12.1 MB	186.5 MB
www.reddit.com	32.2 MB	453.4 MB	21.7 MB	492.1 MB
www.so.com	4.3 MB	24.5 MB	3.3 MB	154.6 MB
www.sogou.com	39.7 MB	720.9 MB	42.7 MB	1.0 MB
www.tianya.cn	7.7 MB	158.1 MB	4.1 MB	129.8 MB
www.wellsfargo.com	40.7 MB	657.2 MB	20.2 MB	606.3 MB
www.wikipedia.org	7.4 MB	61.9 MB	3.3 MB	55.6 MB
www.yandex.ru	25.8 MB	276.7 MB	13.8 MB	357.0 MB
www.youku.com	53.7 MB	1.6 MB	53.9 MB	7.7 MB
www.zillow.com	17.8 MB	1.4 MB	8.2 MB	1.2 MB

3.6 Validation of page models

Now that we have obtained the page models, the next task is to validate the models: how accurate are they, i.e., how "close" are they to the real pages? This requires to come up with some form of distance measure between a real web page and its model. Since ultimately the bits that matter to us are the network traffic traces that is observable to the website traffic fingerprinting attack, which already capture notion of similarities among network traces. We use the state-of-the-art attack, k-fingerprinting of Hayes and Danezis [16], as our validator: train the attack algorithm on **real traces**—network traces captured when Chrome loads the real pages—and test on **generated traces**—those generated by our traffic generator.

Experiment setup and methodology : In the website traffic fingerprinting threat models, the user typically tunnels her traffic through an encrypting proxy channel, and the attacker can observe that encrypted channel. We do the same here: tunnel all traffic through a SOCKS5 proxy channel; one end of the proxy channel is on our client machine on the University of Illinois at Urbana-Champaign campus, and the other end is in Amazon EC2 Oregon.

For each web page, we obtain the real traces by using Chrome to load the page 100 times and using `tcpdump` to capture the traces. During this real

trace collection phase, we also log the round-trip times (RTT) between the server proxy in EC2 and the web servers. This is important because in the next phase, to collect the generated traces, we need emulate similar RTTs to our server simulator, to try to remove one source of difference between the real traces and the generated traces. Also, from all the page models, the maximum number hostnames used by a page model is 30.

To collect the generated traces, we set up the traffic generator as follows. On the server proxy machine in EC2, we launch 30 instances of our server simulator, placing each into a different Linux network namespace. This allows each server simulator to have its own unique virtual IP address and network link on the proxy machine. From the experiment harness script, before loading each page model, we can assign the hostnames used by the page model to individual server simulator by updating the `/etc/hosts` file accordingly. We also configure the RTT of the virtual links accordingly by using the median RTT values observed from the real trace collection phase.

In both phases, collecting real traces or generated traces, we employ the heuristics discussed in Section 3.2.6 to decide when the page has finished loading before stopping `tcpdump`. We collected the traces late October 2016.

Validation results : As a sanity check we perform a normal closed-world k-fingerprinting attack on the real traces. The attack accuracy is 98%. Another sanity check we perform is the same closed-world attack on the generated traces, and the attack accuracy is 99%. With these results we have confidence that we are using the attack code properly, as these high attack accuracies are to be expected.

Next, we modify the attack code[22] to be able to train on the real traces but test on the generated traces. The resulting attack accuracy is 13%, which is better than the 2% if randomly guessing but far from the 98+% achieved by the normal attacks. This means that our generated traces are far from resembling their respective real traces, which is disappointing yet at the same time unsurprising given the challenge that is the task of modeling the web page loading process.

[22] https://bitbucket.org/hatswitch/k-fingerprinting

CHAPTER 4

PROVABLE WEBSITE TRAFFIC FINGERPRINTING DEFENSES OVER TOR

We first describe general provable website traffic fingerprinting defenses, and then we explore a simple design to bring them to Tor.

4.1 Provable defenses

All existing provably secure defenses use a pair of proxies to essentially provide a *single communication channel* through which the web browsing traffic is tunneled, enabling researchers to enforce strict rules on the observable properties of the channel—message sizes, counts, timing, and ordering—in order to reason about its security against a theoretical optimal attacker.

4.1.1 BuFLO

The first provably secure defense in the literature, due to Dyer et al. [13], is specified with three parameters d, p, and τ. The BuFLO channel always sends packets of size d (bytes) at a fixed schedule defined by inter-packet interval p (milliseconds); if real application data is not available when a packet needs to be sent, then dummy bytes are used instead. The channel continues transmission until the minimum amount of time τ (seconds) has elapsed since the start of the page load, or until the page finishes loading, whichever is later.

BuFLO is effective at hiding packet sizes, timing, and ordering. However, for web pages that take longer than τ to load, BuFLO still leaks two related coarse features: page sizes and page load times. In fact, in a 128-site closed-world simulation study, with BuFLO's incurring a bandwidth overhead of 93.5% ($\tau = 0$, $\rho = 40$, $d = 1\,000$), Panchenko et al.'s attack still achieves a 27.3% accuracy rate. The most secure configuration ($\tau = 10$, $\rho = 20$,

$d = 1500$) reduces the attack accuracy to 5.1%—which is still significantly higher than random guessing—but requires a 418.8% bandwidth overhead.

4.1.2 Tamaraw

Cai et al. propose an extension on BuFLO called Tamaraw that yields significant improvement [8]. Exploiting the asymmetry in web traffic, Tamaraw makes two changes to BuFLO to save bandwidth overhead. First, it uses a smaller packet size d of 750 bytes, which Cai et al. observe to be sufficient to cover the majority of outgoing packets when loading Alexa's top 800 sites.[1] Second, Tamaraw uses different sending rates for different directions of traffic: typically web page loads require less frequent outgoing traffic than incoming traffic.

Also, Tamaraw does away with BuFLO's minimum transmission time τ; instead, it pads the number of sent packets to a multiple of a parameter L (each direction is padded separately). Considering only the download direction for a moment, parameter L effectively groups web pages into equivalent partitions based on the number of incoming packets. For example, all web pages that require a number of incoming packets in the range $(AL, (A+1)L]$ for some integer A will generate $(A+1)L$ incoming packets when loaded under Tamaraw and thus appear indistinguishable from one another in the download direction. Thus, a larger L generally implies higher security but also higher bandwidth overhead.

To compare their defense to BuFLO, Cai et al. formulates an ideal attacker, which can observe only the total numbers of incoming and outgoing packets under Tamaraw, and only the total transmission size under BuFLO. They also computed a lower-bound on bandwidth overhead for a given security level. They show that Tamaraw achieves significantly better security/overhead tradeoff compared to BuFLO. For example, in a closed-world scenario using top 800 Alexa sites, at 130% bandwidth overhead, Tamaraw is 30 times more secure than BuFLO against the ideal attacker.

[1] http://www.alexa.com/

4.1.3 CS-BuFLO

Cai et al. also propose a different extension to BuFLO called CS-BuFLO [7] (which they sketched earlier [6]). We highlight those extensions here:

- **Rate adaptation**: instead of using a fixed inter-packet interval ρ throughout the entire session, CS-BuFLO uses a variable interval ρ^*, which it tries to match to the application's rate, to reduce wasted bandwidth for slow senders and reduce latency for fast ones. However, to avoid revealing information about the sender, CS-BuFLO implements several rules: (1) it updates ρ^* only after having sent 2^k bytes for integer values of k (i.e., a CS-BuFLO session that sends total n bytes will make $\log_2 n$ updates); (2) on an update, it sets ρ^* to match the median instantaneous bandwidth value in the interval since the last update; and (3) it rounds ρ^* to powers of 2.

- **Randomized rate**: given a target interval ρ^*, CS-BuFLO randomizes the actual interval between two consecutive writes uniformly in $[0, 2\rho^*]$ to reduce leaking information about the application's real sending rate to Fu et al.'s attack [14].

- **Congestion sensitivity**: CS-BuFLO uses congestion signals from TCP to avoid sending dummy data unnecessarily, e.g., when the network is congested and the TCP socket is refusing Tamaraw packets that contain dummy bytes, Tamaraw does not need to buffer those dummy bytes to be sent later: it can just forget about them.

- **Stream padding**: in order to better hide the total size of transmitted data, CS-BuFLO continues to send dummy data after the application has stopped sending real data.

- **Early termination**: stream padding means the CS-BuFLO server (the proxy closer to the web servers) likely has to continue sending a large amount of dummy data after the page has finished loading. Cai et al. find that this only slightly improves security, so they introduce a command that the CS-BuFLO client (the browser's proxy) can send to the CS-BuFLO server to inform the server that it no longer needs to send stream-padding dummy data to the client.

4.2 Architecture for BuFLO-based defenses over Tor

BuFLO-based defenses CS-BuFLO and Tamaraw use a pair of proxies to provide a single communication channel through which the web browsing traffic is tunneled. Let us call the two proxies the *client-side proxy* and the *server-side proxy*: the browser accesses the BuFLO channel through the client-side proxy, and the web servers access the channel through the server-side proxy. While implementation details of the Tamaraw channel are not specified, the CS-BuFLO channel is implemented as a SOCKS tunnel [7].

We explore a simple architecture to bring BuFLO-based defenses to Tor: tunnel the BuFLO channel through Tor (Figure 4.1). The client-side proxy logically sits between the browser and the Tor client; it provides a SOCKS5 proxy service to the browser, and it uses the SOCKS5 proxy service provided by Tor client. For the server-side proxy, a natural location for it is at the Tor exit node. The advantage of this architecture is that it does not require implementing BuFLO inside Tor. At the minimum, an exit proxy host that is running a server-side proxy can advertise this fact out-of-band, and just needs to configure Tor to allow streams to exit to its server-side proxy; then, clients that want to use this specific server-side proxy can do so via a small number of manual configurations, i.e., the `MapAddress` option.[2] Alternatively, to reduce the amount of manual configuration, Tor can be updated to allow exit nodes to advertise their BuFLO server-side proxy in the directory, and to enable Tor clients to optionally select an exit that runs a BuFLO server-side proxy for certain or all circuits.

On the other hand, a disadvantage of this architecture is that CS-Tamaraw is not directly above the network layer, making it less efficient. For example, if CS-Tamaraw were integrated into Tor, operated to protect the TCP connection between the user's Tor proxy and the Tor entry guard, and supported the ability for the browser to flag a circuit as "sensitive"—the browser would direct sensitive page loads through this circuit—then CS-Tamaraw could potentially reduce the dummy cover data overhead by using for cover the traffic from other circuits, such as less sensitive page loads, Tor-generated communication (such as downloads of consensus data), etc. Additionally, being directly above the network would afford CS-Tamaraw access to more accurate congestion signals, making its congestion avoidance more effective.

[2] https://www.torproject.org/docs/tor-manual.html.en

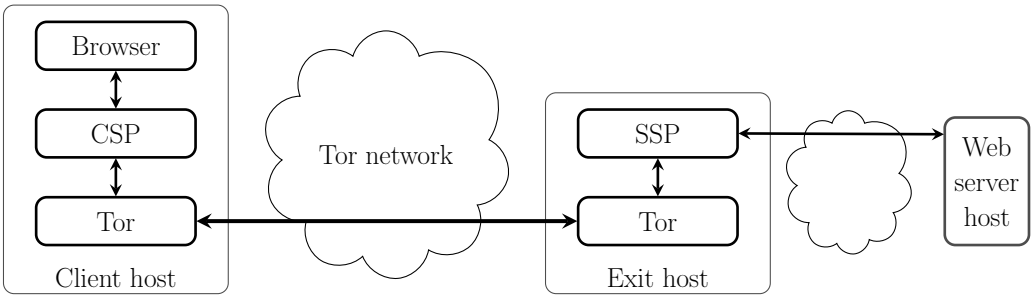


Figure 4.1: Server-side proxy (SSP) is co-located with the Tor exit relay. Logically sitting between the browser and the Tor client, the client-side proxy (CSP) provides a SOCKS5 proxy interface to the browser.

4.3 Proxy and protocol design and implementation

The first step is for us to decide which protocol to consider. Conceptually, Tamaraw is simple, which means its security guarantees can be easily analyzed. CS-BuFLO adds complexity and practicality at the expense of some loss in security guarantees. We decide to err on the side of security—to remain provable—so we mostly go with Tamaraw but do adopt the congestion-sensitivity idea of CS-BuFLO because that does not affect the security of the protocol. For ease of discussion, we will refer to this version as CS-Tamaraw.

4.3.1 Defense session

We imagine that, in order to support real world use and not just research lab experiments, the BuFLO client-side proxy should be able to maintain a persistent underlying communication channel with its server-side proxy, similar to, say, the persistent connections between pairs of Tor proxies on a circuit. In other words, the user should not be expected to restart or reset the client-side proxy before loading a web page if he wants the page load to be protected from website traffic fingerprinting. Essentially, during normal usage the BuFLO proxy should not be in the way of the user. Yet, only the traffic generated when loading a web page needs to be protected; outside of the page loads, other traffic such as that generated by the user's interaction with the web page does not need to be. Therefore, the communication channel provided by the BuFLO proxies is more of a BuFLO-capable channel, where the BuFLO defense can be activated when needed.

Thus, we introduce the notion of a **defense session**. In our design, an invariant that the BuFLO channel maintains at all times is the use of fixed-size cells (though this is not inherently required). The other components of a BuFLO defense, i.e., sending at fixed intervals, the use of dummy cells, and padding the number of sent cells to parameter L, are by default not enabled. A defense session is when the BuFLO channel is actively employing these tactics to protect the TCP proxy channel's traffic. This allows the browser, or possibly a separate controller process, to activate the defense only when it starts loading a web page. Once the page is loaded, the browser can notify the proxy, which will stop the defense session once the stopping condition is met. Or, for example, if the user starts a second web page load while a defense session is active, the browser may choose to stop the defense session only after both web pages have finished loading. We do not claim that this notion of a defense session is novel; we simply explicitly codify it here for ease of discussion.

When is a defense session really done, in the perspective of the client-side proxy? A defense session is done when both directions of communications have been protected, i.e., satisfied the stopping condition. However, the client-side proxy can only guarantee the defense of its sending direction, i.e., it ensures the number of `sent` cells—or more accurately the number of send **attempts** in the presence of congestion—is a multiple of L. For its receiving direction, the client-side proxy cannot infer based on the amount of data received whether the receiving direction has been correctly protected or not, because potentially some of the server-side proxy's send attempts have been denied or partially accepted due to congestion. Therefore, after notifying the server-side proxy to stop the defense, the client-side proxy needs to wait for an explicit signal from the server-side proxy that it has finished defending *its* send direction, i.e., the client-side proxy's receive direction. Then, the client-side proxy can consider a defense session done when both its sending direction and receiving direction are done.

4.3.2 Design

The pair of proxies maintain a single TCP connection with each other. After the initial "handshake" messages that exchange some meta-information to

set up the channel, the two peers effectively switch on the CS-Tamaraw-capable channel: all communication happen in fixed-size **cells**—we stick with Tamaraw's selection of 750 bytes. (In the rest of the document, we will use CS-Tamaraw-capable and CS-Tamaraw interchangeably unless explicitly differentiated.) Each peer maintains a `cell_outbuf` that contains cells that it wants to send to the peer, i.e., `write()` into the TCP socket.

Above the CS-Tamaraw channel, the peers have to multiplex multiple concurrent application TCP streams. Fortunately we do not have to implement the multiplexing layer; rather, we use the SPDY protocol, which has multiplexing support built-in. For the actual implementation we use the open-source `spdylay`[3] library, which is fairly mature and by design cleanly separates the SPDY protocol logic—which it handles—and the network IO—which it leaves to the application. This is important because once again it allows us to perform network IO using only the system calls that have stable implementations in Shadow. Each peer maintains a `spdy_outbuf` that contains all the bytes that the SPDY protocol wants to send to the other peer, including SPDY control frames (e.g., for setting up and tearing down streams) and data frames (containing the user—i.e., browser—traffic).

Figure 4.2 illustrates the logical components of the CS-Tamaraw proxy as well as the how output data from the browser data flows through the proxy when there is *no* active defense session. The proxy logical component deals with receiving data from the browser's multiple connections, pushing them through the SPDY engine, which outputs a single stream of data that can be stored in the `spdy_outbuf`. Whenever there is data in `spdy_outbuf`, the proxy component immediately forwards the data to `cell_outbuf` to be sent—using the `write()` system call—whenever the underlying TCP connection allows.

Figure 4.3 shows the system and the data flows when there is an active defense session. As before, the proxy component is responsible for filling the `spdy_outbuf`, but in this case it does not interact with `cell_outbuf`. The CS-Tamaraw logical component removes data from the `spdy_outbuf` and packs it into fixed-size cells and adds them to `cell_outbuf`, as well as adding dummy cells if needed. Specifically, CS-Tamaraw schedules a repeating timer with the specified frequency. Whenever the timer fires, then it is time to *attempt*

[3] http://tatsuhiro-t.github.io/spdylay/

to write exactly one cell's worth of data into the network socket connection, i.e., `write(fd, cell_outbuf, 750)`. Right before making this call, CS-Tamaraw logic ensures that `cell_outbuf` contains at least 750 bytes: if there are fewer than 750 bytes, then CS-Tamaraw adds to it either a (potentially padded) data cell from `spdy_outbuf` or a dummy cell; if there are already at least 750 bytes in `cell_outbuf`, then nothing needs to be added to it. This logic captures part one of the congestion sensitivity: CS-Tamaraw does not unnecessarily add dummy cells.

Part two of CS-Tamaraw's congestion sensitivity is dropping dummy cells already queued. Suppose CS-Tamaraw adds a dummy cell to `cell_outbuf`, and then because of network congestion, over the next several intervals, it has not been able to drain `cell_outbuf`, such that the dummy cell still remains. Then suppose at the next interval, before making the `write()` call above, CS-Tamaraw finds that there is now data in `spdy_outbuf`; then CS-Tamaraw will remove the dummy cell from the end of `cell_outbuf` and replace it with data from `spdy_outbuf`. It is possible to optimize even further: replace dummy data from a data cell that is still queued at the end of `cell_outbuf`, with newly available data from `spdy_outbuf`; however, we do not implement this optimization because it requires updating meta-data of a cell already in `cell_outbuf`, a more intricate operation in our implementation, and it is not clear how much we will gain.

4.3.3 Implementation

We have implemented the CS-Tamaraw from scratch, totaling 6 400 lines of code. The proxy is instructed to be a client-side proxy or a server-side proxy (but not both) at start-up time. Each client-side proxy can only peer with a single server-side proxy at a time, but each server-side proxy can serve any number of concurrent client-side proxy peers. The client-side proxy can be configured to reach a server-side proxy directly, or indirectly via a SOCKS proxy such as Tor.

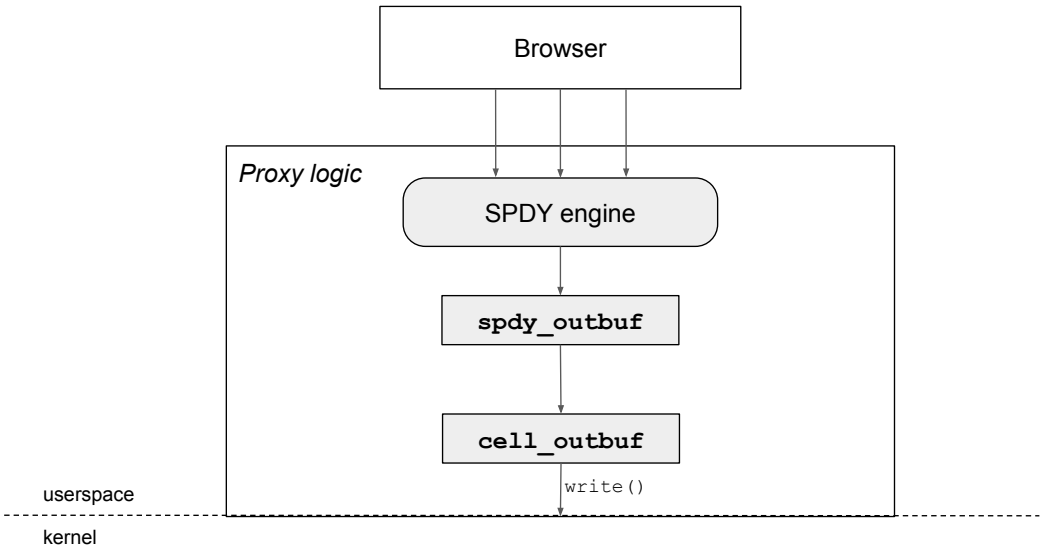


Figure 4.2: The high-level logical components of our CS-Tamaraw proxy and how output data flows from the browser's multiple connections through the proxy to the underlying "network" when there is no active defense session.

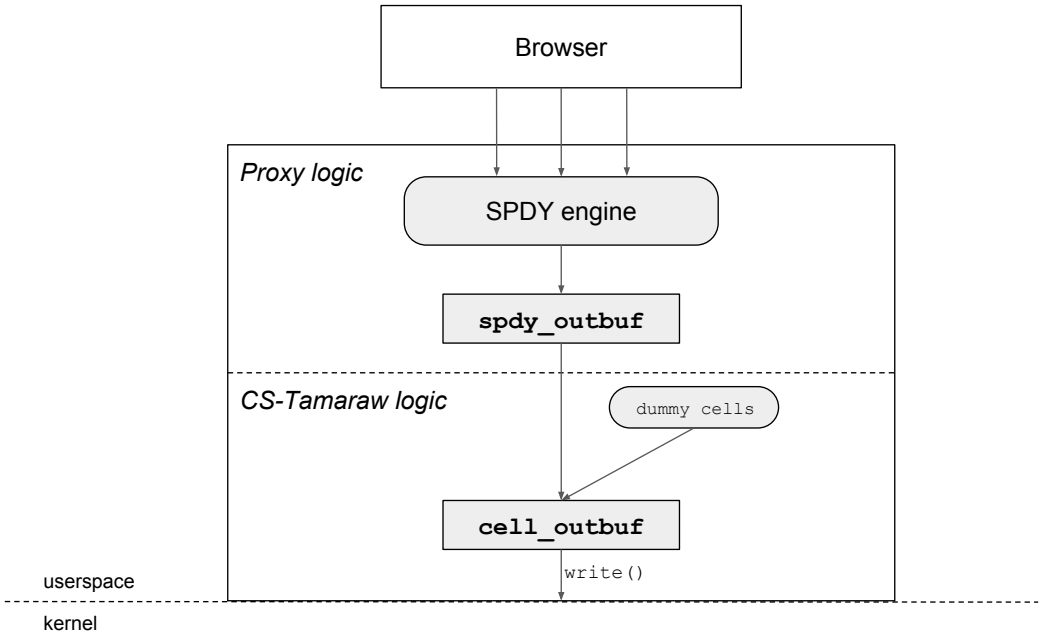


Figure 4.3: The high-level logical components of our CS-Tamaraw proxy and how output data flows from the browser's multiple connections through the proxy to the underlying "network" when a defense session is active.

L	**Attack accuracy (%)**	Performance (avg per page load)						
		Bytes recv (MiB)	Bytes sent (MiB)	Dummy cells recv	Dummy cells sent	Bytes recv overhead (%)	Bytes sent overhead (%)	**Overall overhead (%)**
50	65.1	0.43	1.68	557	754	1498	48.5	82.5
100	50.3	-	-	-	-	-	-	-
150	45.8	0.46	1.71	599	756	1613	51.4	88.0
200	39.4	0.48	1.74	628	799	1685	54.1	92.3
250	39.3	0.49	1.75	644	852	1727	54.6	93.8
300	34.3	0.51	1.79	668	904	1790	57.9	98.5

Table 4.1: Performance and security tradeoffs when when page models are loaded through the CS-Tamaraw peers that are directly connected, i.e., not through Tor. The packet rates are, for the client-side proxy, $p_{out} = 20$ and $p_{in} = 5$. The security metric is the k-fingerprinting accuracy. The performance metrics are as observed by the server-side proxy; overall overhead is the combined send and receive bandwidth overhead. The server-side proxy log for the $L = 100$ experiment is incomplete, so we do not show it here.

4.3.4 Sanity check experiments

Before running experiments on the CS-Tamaraw design through Tor, we run the performance-security tradeoffs experiment with the two CS-Tamaraw peers having a direct connection. The experiment setup is similar to the setup in Section 3.6, i.e., we use our traffic generator to load the 50 page models, 80 times each. The differences are as follows:

- The client machine, i.e., the browser simulator and client-side proxy, is in the same Amazon EC2 region as the server-side proxy, so that we can perform these experiments under consistent network conditions. We use Linux `tc` and `qdisc` tools to configure the link between the client-side proxy and server-side proxy to have 80 ms RTT, 19 Mb/s upload bandwidth and 55 Mb/s download bandwidth for the client. These speeds are average US broadband speeds as of second half of 2016 [1].

- CS-Tamaraw is enabled to defend the page loads. Since this is only for sanity checking, we need not explore a large space of parameters; thus, we keep p_{out} and p_{in} fixed at 20 ms and 5 ms, respectively, and vary L.

The key results are shown in Figure 4.4, which clearly shows an inverse relation between the security and bandwidth overhead, as expected. The security metric is the k-fingerprinting attack accuracy on the traces collected

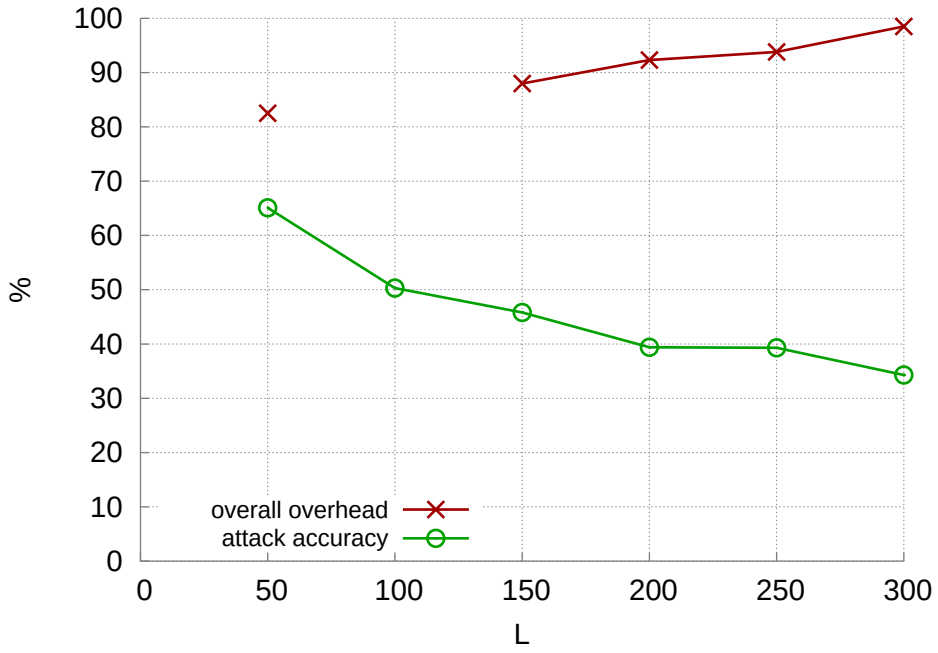


Figure 4.4: Overhead and security tradeoff graph when page models are loaded through the CS-Tamaraw peers that are directly connected, i.e., not through Tor. The log for the $L = 100$ experiments is incomplete, so we do not show overhead data.

at the client, and the overall overhead is the combined send and receive bandwidth overhead as observed by the server-side proxy at the application level. We also look closer at other performance metrics in Table 4.1. Since we are showing the perspective of the server-side proxy for these metrics, the bytes it receives are what the browser sends, and since a typical web page load requires the browser to send much smaller amount of traffic than it receives, the overhead of the bytes received by the server-side proxy is much larger than the bytes sent (over 1 000% versus less than 100%). However, the overall overhead, combining both send and receive direction, is still within the tolerable realm. $L = 300$ provides the strongest defense with almost 100% bandwidth overhead.

We also note that, in these experiments neither the client-side proxy nor the server-side proxy ever reports any avoided dummy cell. It is possible that the CS-Tamaraw channel never became congested enough to activate the congestion avoidance logic in these experiments. However, after subsequent investigation, we find that it is also possibly due to a bug we discover where the proxies did not count the situation where they were able to avoid adding

a dummy cell to `cell_outbuf`—the first part of the congestion sensitivity logic; they were counting only occurrences of the second part of that logic.

To confirm the fix, we run a few quick experiments using the minimal client and server described in Section 4.4.1, and make the server wait two seconds before starting to send the response. During these two seconds, the server-side proxy does not have any user data to send and thus will have to resort to using dummy cells. Thus, with the maximum TCP send and receive windows set to 21 KB,[4] we observe consistent counts of avoided dummy cells by the server-side proxy; however, the client-side proxy still does not avoid any dummy cells. When we set the maximum windows to 14 KB, then we also see the client-side proxy avoid dummy cells; this is because the client-side proxy sends at a slower rate (20 ms versus 5 ms), and thus smaller windows are required to cause congestion.

4.4 Evaluating CS-Tamaraw over live Tor network

To run experiments over the live Tor network, we set up in Amazon EC2 our own custom "exit" node (Figure B.1): it advertises to the world as being only a relay node, i.e., exit policy of "`reject *:*`", but secretly it allows connections to the co-located server-side proxy: exit connections to the special hostname `buflo-tproxy-ssp` are directed to the server-side proxy. (It is worth emphasizing that we do not inspect, store, or in any way process the relay traffic traversing our relay, other than redirecting our own special exit traffic.) We make sure to run the Tor node for a sufficient amount of time for it to get the `Fast` and `Stable` flags, and attract a consistent 920 KB/s of relay traffic (this seems to be the limit imposed by EC2 on `small` instances). The other server-side proxy's setup involving the server simulators is identical to the earlier experiments.

We run the client on the University of Illinois at Urbana-Champaign campus. The Tor client is configured with the `MapAddress` option, such that client connections to the special hostname `buflo-tproxy-ssp` will be sent down a circuit that exits at our own Tor node. The client uses the browser simulator to load the 50 page models, 80 times each; specifically, we collect all 80 traces for one page before moving on to the next page. Each of these

[4] i.e., by setting `net.ipv4.tcp_rmem` and `net.ipv4.tcp_wmem` on Linux.

L	**Attack accuracy (%)**	Performance (avg per page load)						
		Bytes recv (MiB)	Bytes sent (MiB)	Dummy cells recv	Dummy cells sent	Bytes recv overhead (%)	Bytes sent overhead (%)	**Overall overhead (%)**
50	72.7	0.60	2.39	798	1743	2150	111	160
100	73.9	0.66	2.53	881	1930	2359	122	174
150	72.3	0.63	2.42	837	1782	2238	114	164
200	71.8	0.65	2.41	858	1781	2284	114	165
250	68.1	0.83	2.94	1116	2504	3031	159	224
300	70.5	0.75	2.63	1002	2157	2844	144	207
500	66.0	0.91	3.04	1234	2634	3317	165	237
1000	63.7	0.90	2.73	1213	2318	3265	157	234

Table 4.2: Performance and security tradeoffs experiment for when the CS-Tamaraw peers are connected through Tor. The send rates are, for the client-side proxy, $p_{out} = 20$ and $p_{in} = 5$. The security metric is the k-fingerprinting accuracy. The performance metrics are as observed by the server-side proxy.

experiments takes a significant amount of time, by virtue of being a live Tor experiment, with lower performance and higher failure rates that would require multiple runs in order to obtain the 80 traces for each page.

For the first batch of experiments in early January 2017, we once again fixed p_{out} and p_{in} at 20 ms and 5 ms, respectively, and varied L. As the results come in, we began to be perplexed: increasing L did not noticeably decrease the attack accuracy. The pattern persisted through the entire set of values for L that we experimented with in the sanity check experiments. We were working with the belief that, since the CS-Tamaraw channel removes timing and size information from its output, then whatever the underlying transport, e.g., either a direct connection between the two peers, or a proxied connection through Tor, the CS-Tamaraw channel would not leak any additional information. Thus, we had expected similar gains in security as we increased L. Thinking that it was perhaps due to some transient issue, we reran the experiments through the remainder of January 2017 and also added two more runs with $L = 500$ and $L = 1\,000$.

The end results are summarized in Table 4.2. First of all, due to the more fragile nature of loading web pages over Tor, as L increases, the trends in the metrics are not as smooth as in the direct connection, but they are clear trends nonetheless. Clearly, the overall trends are in the expected direction: i.e., as L increases, overall bandwidth overhead increases, and attack accuracy decreases. However, the key metrics start off from worse positions compared to the direct connection—i.e., at $L = 50$, attack accuracy is 72.7% compared

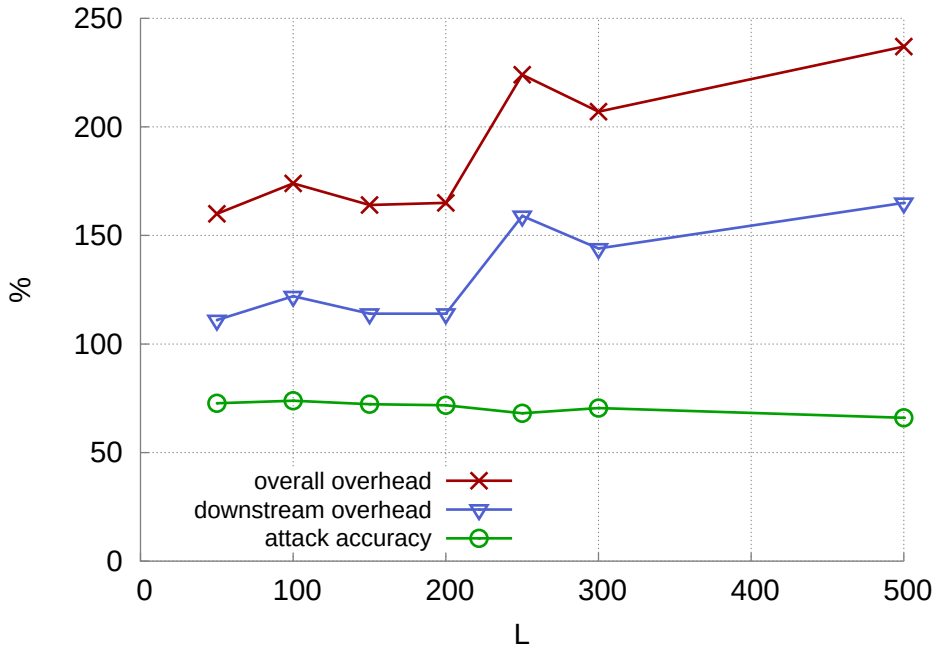


Figure 4.5: Overhead and security tradeoff graph when page models are loaded through the CS-Tamaraw peers that are connected through Tor

to 65.1%, the bandwidth overhead is 160% compared to 82.5%—and also improve more slowly—i.e., at $L = 300$, the overhead is more than twice in the direct connection, yet the attack accuracy is still above 70%. Finally, at $L = 1000$, the attack accuracy is ∼64%, but requires three times as much bandwidth overhead as in the direct connection.

We need to understand the cause behind CS-Tamaraw's lackluster performance when tunneled through Tor. With Tamaraw-based defense, we expect that the only information that is leaked is the total bytes of the page load. We want to confirm this is at least possibly the cause of the information leak. We examine the entropy of the packet trace size, i.e., the total sizes of the packets in a trace. We see that when the proxies are connected through Tor, there is no noticeable change in the entropy as we increase L, whereas it generally decreases—albeit not a smooth line—as we increase L when the proxies are directly connected. Nevertheless, the evidence here is insufficient to make any conclusion, e.g., it's possible that our implementation of CS-Tamaraw proxies are erroneous. Therefore, we conduct additional experiments to investigate this issue.

L	Entropy of total size of traces		
	Upstream+downstream, connected through Tor	Upstream+downstream, connected directly	Downstream only, connected directly
50	8.418	5.389	4.365
100	8.422	5.335	4.271
150	8.423	5.353	4.124
200	8.422	4.993	3.604
250	8.422	4.890	3.639
300	8.421	5.299	4.190
500	8.420	-	-
1 000	8.419	-	-

Table 4.3: Entropy of the total sizes of IP datagrams, including IP headers, in the traces. (We did not perform $L = 500$ and $L = 1\,000$ experiments where the proxies were directly connected.)

4.4.1 More rigorous sanity checks

To eliminate any potential issue with our web page models as well as our browser and server simulator, we decided to use the simplest page models possible: each page is a single file whose size is the total download size of the original page.[5] We first ran a set of experiments using Chrome as the browser, and a simple web server based on `web.py` module.[6] The web server is configured with the sizes of the responses to return to the browser, based on the request URL. We should stress that, we are not able to control the exact sizes of the request and the responses, since the browser includes request headers, and the `web.py` layer includes response headers, etc. The network configuration is as in the sanity check experiments 4.3.4, except in this case we also fix the link between the server-side proxy and web server to be 80 ms RTT instead of varying based on the page. The CS-Tamaraw defense is once again $p_{in} = 5$ and $p_{out} = 20$, and we vary L in increment of 50 between 0 (i.e., no defense) and 250, and then 500.

Looking at the results, as shown in Figure 4.6, we see again that as we increase L, the k-fingerprinting attack accuracy (**k-FP**) generally goes down: from from 75% with $L = 50$ down to 41% with $L = 500$. (Note that $L = 0$

[5] We are using single-file models of our page models, not of the original pages; nevertheless, this should not affect the validity of the experiments.

[6] http://webpy.org/

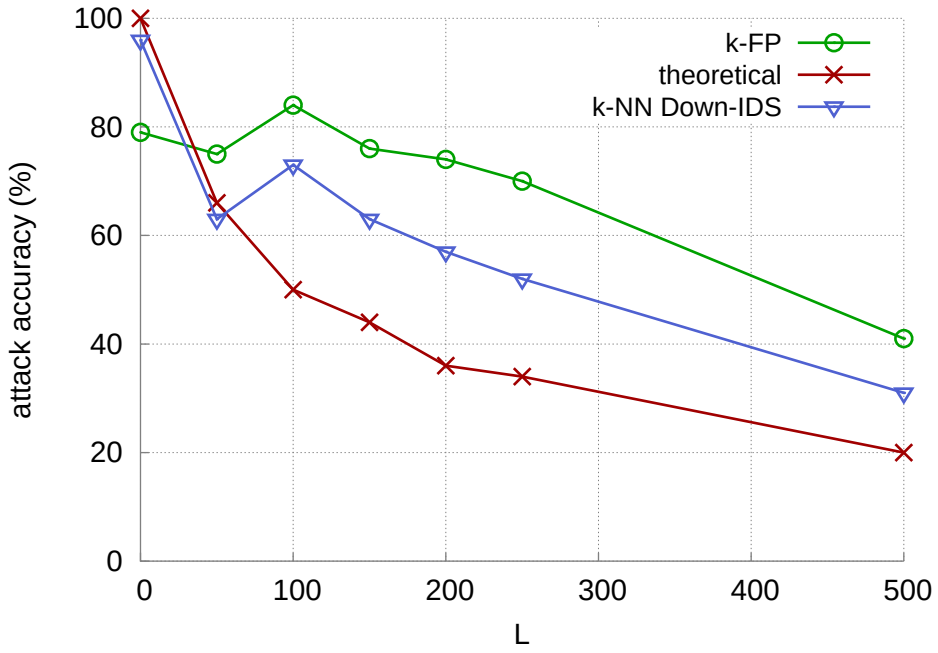


Figure 4.6: Attack accuracies when each page is a single main resource, fetched by Google Chrome over CS-Tamaraw defense tunnel, and web server application is built on top of `web.py` framework. It is not possible to precisely control the amount of *application-layer* data sent into the network by either the client or the server.

means no defense is enabled, and the 80% attack accuracy is curious.) Working with the assumption that the CS-Tamaraw defense leaks only the total download size, we also compare the k-fingerprinting result to the accuracy of a simplified theoretical experiment and attacker outlined by Cai et al. [8] that have only the total page size available, i.e., no network traces involved. The corresponding **theoretical** attack accuracies are 66% and 22% at $L = 50$ and $L = 500$, respectively, significantly lower than k-fingerprinting is able to achieve. We also quickly implement a simple k-NN classifier ($k = 40$) that classifies the total downstream IP datagram size (IDS) of the traces. This classifier **k-NN Down-IDS** also achieves higher accuracies than the **theoretical** experiment.

These results suggest that the total page sizes still leak significant information, prompting us to proceed to tighten up the experiment setup. After several attempts, including disabling the CS-Tamaraw defense and leaving the padding defense to the server application layer, fail to explain the discrepancies, we decide to simplify the client and server even further, to something

that affords us precise control over the amount of transferred *application* data.

Minimal protocol and applications We implement a simple custom non-HTTP protocol where the client sends a single message that instructs the server to send the desired amount of data back to the client, as fast as possible. For example, if the client wants the server to send back $N = 1\,234$ bytes, it simply sends the string `0000001234` to the server, which will send back exactly 1 234 bytes. (The message from the client is 0-padded so that all requests have the same size even though they request different amounts of data.) Furthermore, depending on the experiment, the client optionally takes care of defending the download amount: given a specific L parameter, it will round the request size to the nearest multiple of $750 * L$, where 750 is the cell-size that our CS-Tamaraw protocol uses. For example, if the page size is 40 000 bytes and $L = 50$, the client will request $750 * L * 2 = 75\,000$ bytes from the server. Also, since each "page" is now only one resource, we consider the "page load" done as soon as the resource completes the download, instead of waiting for 2 additional seconds of idleness as outlined in Section 3.2.6. (Refer to Appendix A for the client and server code.)

Direct, LAN connection In this experiment, we use the simplest network configuration possible: the client and server are connected directly without the proxies, eliminating CS-Tamaraw from the picture altogether—which means we enable the client to control the defensive-padding described above. The client and server machines are on the same local network in EC2, where the link is 1 Gb/s and 0 ms RTT. The client fetches each of the 50 "pages" 80 times. Before examining the results, we describe the classifiers and the features that we use:

- **k-FP Full**: This is the k-fingerprinting classification algorithm, run with the full default feature set implemented by the code release.
- **k-FP Down-IDS**: for consistency, we use k-fingerprinting in the rest of the experiments instead of our one-off **k-NN** classifier. The feature vector here is a single value: the total downstream IP datagram size (IDS) of each trace.
- **k-FP Up-IDS**: k-fingerprinting with the total upstream IP datagram size as the only feature.

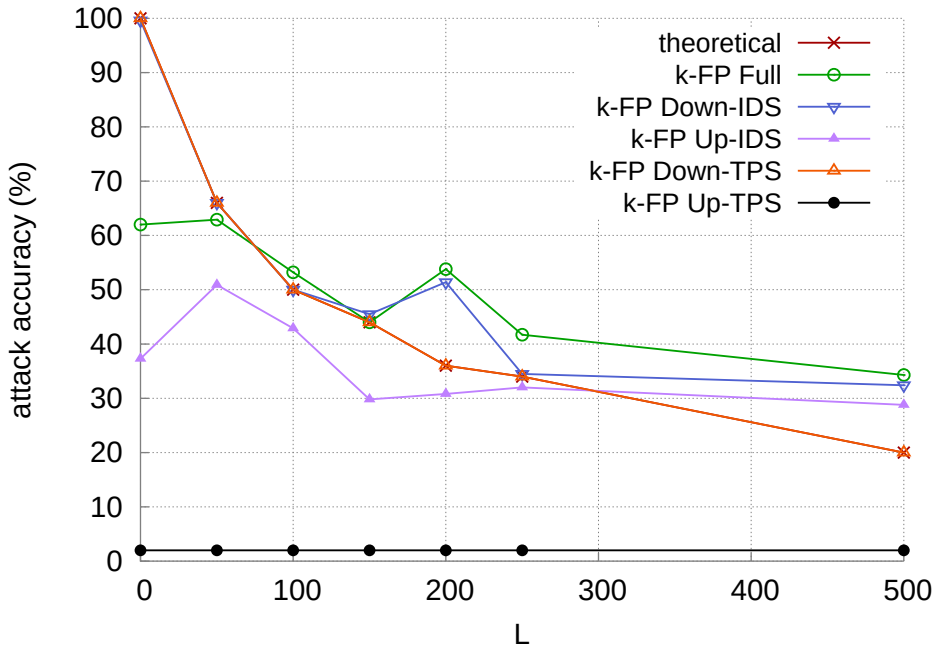


Figure 4.7: Attack accuracies by k-fingerprinting on no-CS-Tamaraw-direct-LAN page loads where the client and server speak a minimal stripped down non-HTTP protocol, which makes it possible to precisely control the amount of *application-layer* data either side transmits. Without CS-Tamaraw providing protection, the client controls the defensive padding.

- **k-FP Down-TPS**: k-fingerprinting with the total downstream TCP payload size (TPS) of each trace as the only feature, i.e., this means that packets with empty TCP payload, which are typically 68- or 74-byte IP datagrams, do not contribute towards the fingerprint of the traces.
- **k-FP Up-TPS**: k-fingerprinting with the total upstream TCP payload size of each trace as the only feature.
- **theoretical**: the simulated attack aforementioned that has only the original size of each page.

We now examine the results, as shown in Figure 4.7. First, we can see that the **k-FP Up-TPS** attack accuracy is consistently at 2%; this is as expected because all request sizes are identical, so k-fingerprinting is essentially making a random guess out of 50 possible choices. Furthermore, since the server application responds with precisely the amount of data requested

by the client, the **k-FP Down-TPS** accuracy matches that of the **theoretical** attacker exactly. After many unexpected results, this result gives us a foundation on which we can incrementally build to get back to our desired CS-Tamaraw-over-Tor experiments.

Next, the **k-FP Up-IDS** accuracy ranges between 30% and 50%, which is not entirely expected but makes sense in retrospect, given that the IDS feature includes the sizes of the upstream TCP ACK packets that the client's TCP stack sends to the server. As a result, larger pages generally cause more upstream ACK packets, and thus the total size of upstream IP datagrams contains some information about the size of the page. The **k-FP Down-IDS** accuracy is at least as good as and often better than the **k-FP Down-TPS/theoretical** baseline, once again with the only difference being the additional information provided by IP and TCP header sizes. While we cannot explain the unexpected jump in accuracy at $L = 200$, one plausible explanation is that transient changes in network condition, e.g., congestion—recall that we are on a shared cloud environment—can cause two pages that are in the same theoretical bucket to generate distinct network profiles, e.g., there are significantly more ACK packets in one profile than the other.

Finally, we look at the **k-FP Full** attack. On one hand, we see that this attack is generally better than all other attacks, which generally matches our expectation. On the other, as in the earlier results shown in Figure 4.6, when $L = 0$, i.e., there is no attempt to defend the traffic, at 60% accuracy, **k-FP Full** significantly underperforms the baseline. Since this is now no longer isolated behavior, we start to suspect it could be because k-fingerprinting is not tuned for attacking "defenseless" traces, thus certain low-information features negatively affect its overall performance.

Broadband-like connection over CS-Tamaraw In this experiment, we again use the same client and server application, but the defense will be provided entirely by CS-Tamaraw proxies, i.e., the client application now requests the original size of each page. Also, the client and the server-side proxy are about 80 ms RTT apart, and about 80 Mb/s download and 30 Mb/s upload bandwidths[7] are available to the client. The link between the server-side proxy and the server has 40 ms RTT but unrestricted bandwidths.

Overall, the results as shown in Figure 4.8 are surprising but in a good

[7] extrapolated based on the data in the broadband report [1]

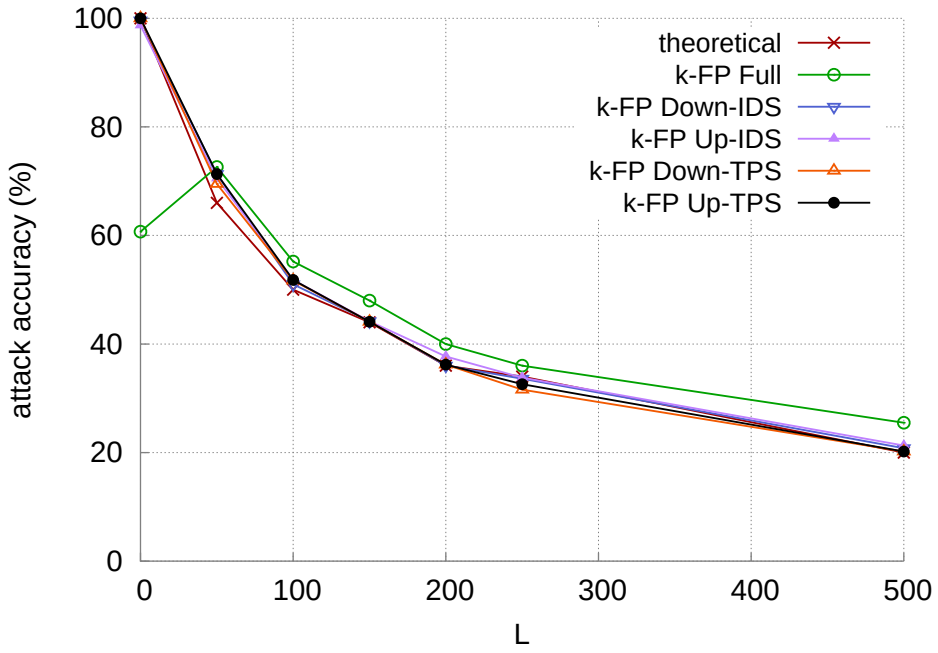


Figure 4.8: Attack accuracies on page loads where the client and the server speak a minimal stripped down non-HTTP protocol over a broadband-like connection that is protected by CS-Tamaraw.

way: all attack accuracies are close to theoretically predicted values (save for the sub-par performance of **k-FP Full** when no defense is employed, which we have seen in earlier experiments). **k-FP Full**, which is of most interest, is never more than 7% in absolute terms more successful than the **theoretical** attack. With this result, we have high confidence in the correctness of our protocol and applications, most importantly in the CS-Tamaraw implementation. Therefore, we can now proceed to extend the experiment to tunnel CS-Tamaraw through Tor.

4.4.2 Revisiting live Tor experiments

Since we have described the setup for these experiments earlier in this Section, we only highlight the differences here: we use the simple client and server to transfer single files instead of using our web page modeling applications; also we omit the 2-second idleness check when considering the page load "done"—but we stress that this does not affect the CS-Tamaraw client-side proxy's determination of when the defense sessions are done. We

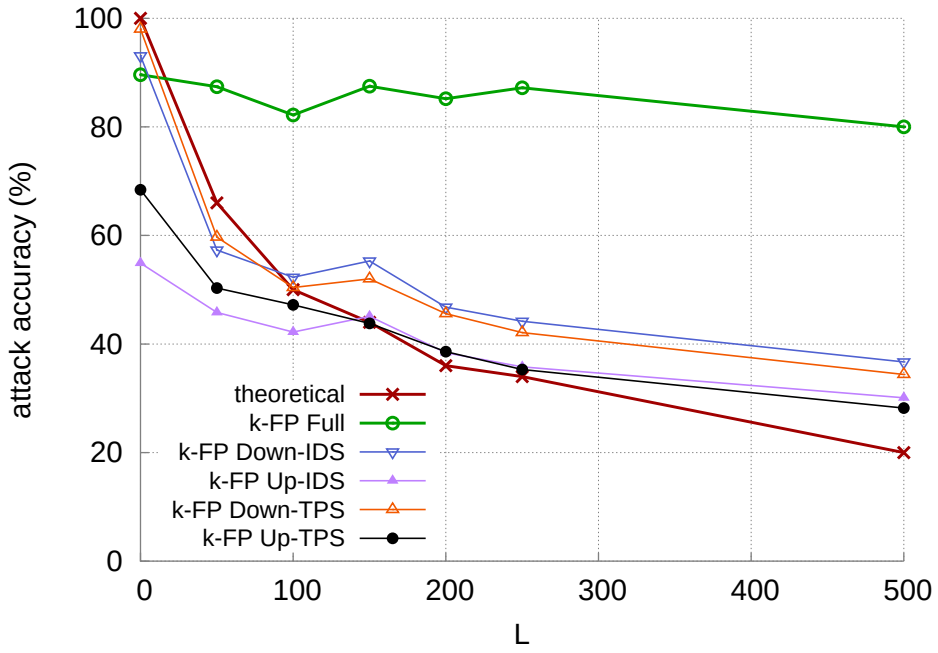


Figure 4.9: Attack accuracies on page loads where the client and the server speak a minimal stripped down non-HTTP protocol through a CS-Tamaraw channel that is tunneled through Tor.

performed these experiments in the second half of June 2017. Because of the large duration of each experiment (L is the only parameter here), we ran them in parallel, using three different exit relays with essentially identical configurations,[8] after ensuring that all three have attained 920 KB/s of relay traffic. The attack accuracy results are shown in Figure 4.9.

First, the k-fingerprinting results based on only sizing information from the traces are generally tracking the **theoretical** predictions. The differences between the corresponding IDS and TPS results (e.g., **k-FP Down-TPS** v.s. **k-FP Down-IDS**) are negligible; a possible explanation is that, since CS-Tamaraw proxies transmit fixed-size cells at constant rates—but importantly, *not* trying hard to fill up the pipe—there is little opportunity for the kernel to coalesce multiple cells into the same TCP segment, and thus there is almost a one-to-one mapping between a cell going one direction and a subsequent ACK segment going in the opposite direction. More importantly, even though these attacks are significantly more successful than **theoretical** predictions, which is a surprising result on its own, they are nowhere close to the wild

[8] also, the three relays were all EC2 instances, in the same EC2 `us-west-2a` "availability zone."

success of the **k-FP Full** attack, which hovers above 80% even as **theoretical** predictions and the second best attack, **k-FP Down-IDS**, quickly reduce to 50% and lower starting at $L = 100$. Note that, as big a surprise this result is, we see it earlier in Table 4.2.

Since the total upstream and downstream size information do not explain the unexpected success of **k-FP Full**, we re-run k-fingerprinting with a few isolated features—due to time constraints, we are not able to comprehensively explore the feature space used in **k-FP Full**. First we briefly describe the features, all of which are used in **k-FP Full**:

- **k-FP Timing**: uses only timing-based features, which we isolate further into two sub-feature sets.
- **k-FP Timing-Inter**: uses only 12 features from the inter-packet timings, i.e., mean, maximum, standard deviation, and 75th percentile, for each direction of traffic and for both directions combined. The 13th feature is the sum of the 12.
- **k-FP Timing-Quartile**: uses only 12 features from the raw packet timestamp values (specifically, offsets from the first packet in the trace), i.e., the 25th, 50th, 75th, and 100th percentiles, for each direction of traffic and for both directions combined. The 13th feature is the sum of the 12.
- **k-FP Ordering**: uses only four features: the means and standard deviations of downstream and upstream packet positions within the full packet trace.
- **k-FP Concentration**: uses various statistics on packet concentrations, e.g., the number of upstream packets in each consecutive non-overlapping group of 20 packets.

We show the results of using these isolated features in Figure 4.10, along with **theoretical**, **k-FP Full**, and **k-FP Down-IDS** for reference. We make several observations here. First, when there is no defense, the poor performances of **k-FP Timing-Inter**, **k-FP Concentration**, and particularly **k-FP Timing-Quartile** and **k-FP Ordering**, likely contribute to the unexpected inferior accuracy achieved by **k-FP Full** as compared to **theoretical** and even **k-FP Down-IDS**. Second, starting at $L = 100$, all

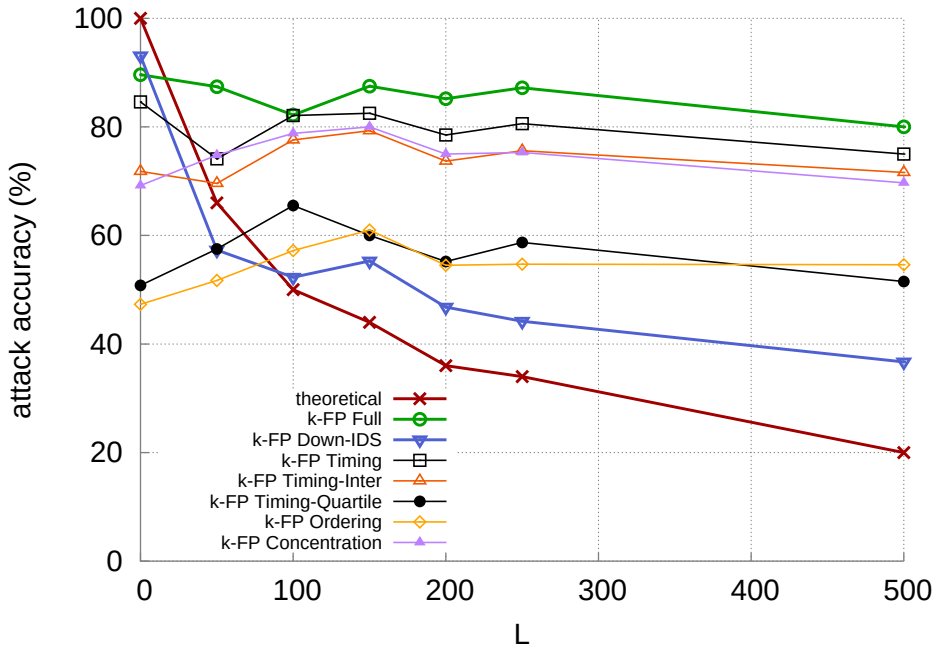


Figure 4.10: Attack accuracies by k-fingerprinting using additional isolated feature sets on page loads where the client and the server speak a minimal stripped down non-HTTP protocol through a CS-Tamaraw channel that is tunneled through Tor.

the k-fingerprinting features considered result in a stronger attack than those based only on trace sizes. Finally, **k-FP Timing-Inter** is always more accurate than **theoretical** and appears potentially largely responsible for the strong performance of **k-FP Timing**, which, along with **k-FP Concentration**, contribute to the success of **k-FP Full**.

The above results are surprising, and we have yet an explanation; however, there is another angle we can investigate: how would k-fingerprinting perform in a slightly more realistic scenario, where there is a larger temporal separation between the collections of the training data and the test data. Thus, we start another set of CS-Tamaraw-over-Tor experiments on July 1st, 2017, approximately 10 days after the experiments above, which we started on June 21st, 2017.

First we run **k-FP Full** on this second set of traces, and the attack accuracies are consistent with those on the June 21st traces. Then we run two other attack scenarios: train on June 21st traces and test on July 1st traces (**Tr6-Te7**), and vice versa (**Tr7-Te6**). As the results in Figure 4.11 show, k-fingerprinting's accuracy significantly degrades with temporal separation

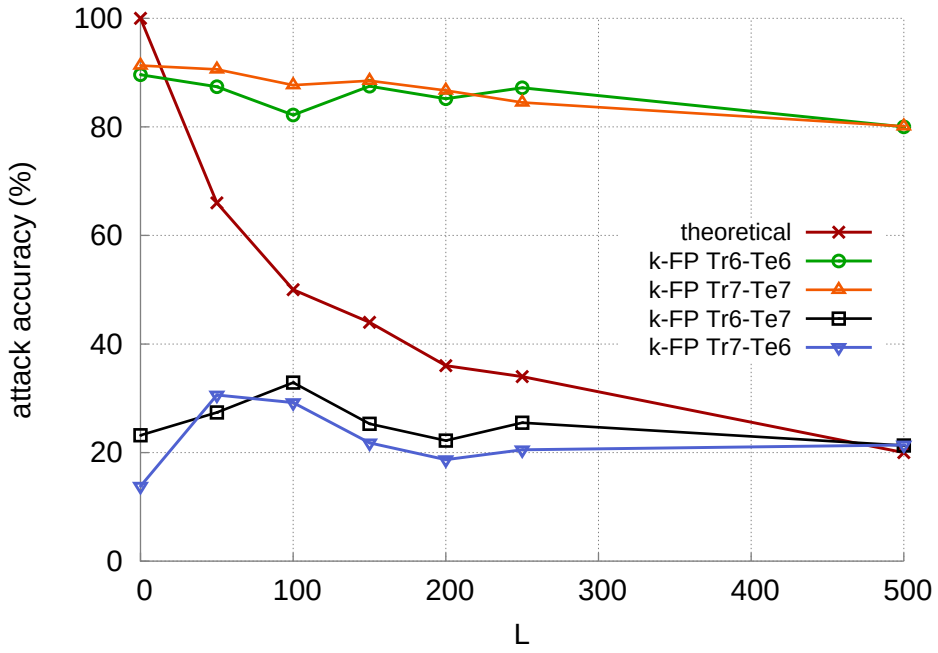


Figure 4.11: Attack accuracies from two sets of traces on page loads where the client and the server speak a minimal stripped down non-HTTP protocol through a CS-Tamaraw channel that is tunneled through Tor. "TrX-TeY" means the attack is trained with traceset X and tested on traceset Y, where "6" and "7" refer to June and July, respectively, the months in which the tracesets were collected.

between training and testing data, which is not a surprising result. Also, consistent with the results we have seen thus far, CS-Tamaraw is not able to provide significant improvement in privacy.

However, as we confirmed in the sanity check experiments, CS-Tamaraw performs as expected when the proxies are directly connected. Therefore, the surprising results for CS-Tamaraw-over-Tor here could potentially be due to the interaction between CS-Tamaraw and Tor, or due to Tor alone (and possibly other factors). So, we perform another set of experiments to isolate the unexpected behavior even further: once again we remove CS-Tamaraw from the picture, and let the client and server defend their downloads over Tor. We show the results in Figure 4.12. Once again we see that k-fingerprinting is able to achieve significantly higher attack accuracies against the defense, compared to theoretical predictions as well as when the client and server are directly connected without Tor. With this result, we can largely rule out the interaction between CS-Tamaraw and Tor as the root cause of CS-Tamaraw's

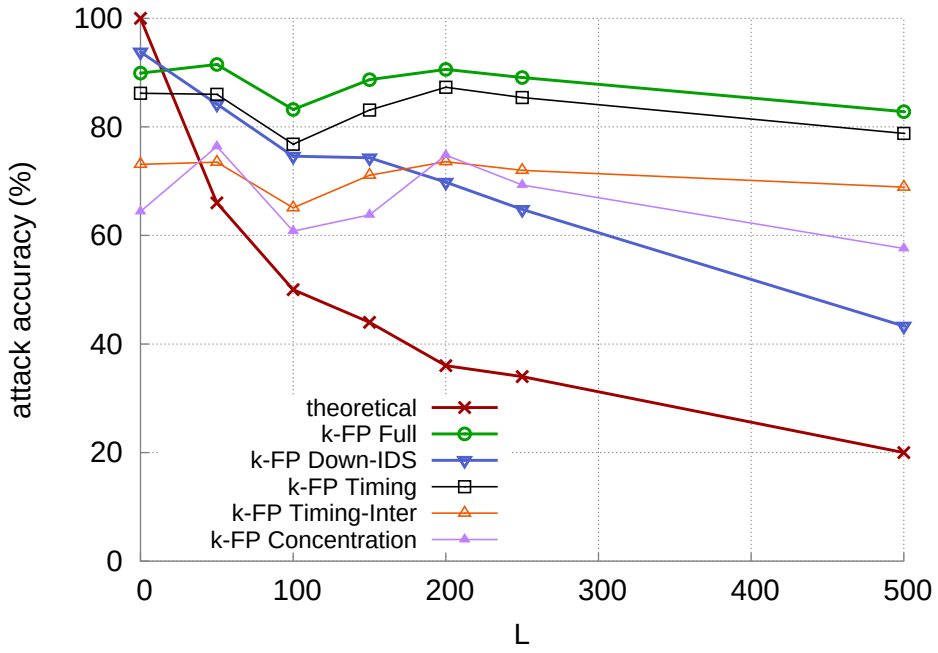


Figure 4.12: Attack accuracies by k-fingerprinting on page loads where the client and server are connected through Tor and speak a minimal stripped down non-HTTP protocol. The client controls the defensive padding.

lackluster security when tunneled over Tor.

At this point, we take a moment to summarize the macro observation thus far of the effectiveness of the defenses in relation to the underlying transport in Table 4.4. Whether the client and server defend themselves, or the defense is provided by CS-Tamaraw, the strength of the defense *mostly* matches up with expectations. Recall the **Direct, LAN connection** experiment, which is the simplest possible scenario: even there, we can see in Figure 4.7 that the defense is not perfect, i.e., at $L = 200$ and higher. Yes, it defends the TCP payload size perfectly by application-layer padding, but **k-FP Full** exceeds theoretical bounds. We speculated there that a possible cause is transient changes in network conditions. The fact that k-fingerprinting exceeds the theoretical bounds by even larger margins when either defense is tunneled through Tor gives more weight to our speculation regarding the role of time-variant network conditions.

This prompts us to re-examine our trace collection methodology, and we realize that we are giving the attack an unlikely advantage. As described earlier, we collect all 80 traces for one page before moving on to the next

Underlying transport	Defense provider	
	Client and server	CS-Tamaraw proxies
Direct	✓/?	✓
Tor	!?	!?

Table 4.4: Summary of the expected efficacy of defensive schemes relative to theoretical bounds. When the two end points of the defensive channel are directly connected, the defense is *typically* as effective as expected; when the same defensive channel is tunneled through Tor, it significantly loses its efficacy, i.e., k-fingerprinting attack exceeds theoretical bounds by wide margins.

page. This means that, all 80 traces for page A are collected, for example, around 5 AM, while all 80 traces for page B might be collected around, say, 1 PM. The implication is that the training traces and the test traces for a given page are likely collected under the same network conditions, e.g., the amount of network congestion, and thus are likely to yield similar fingerprints. Furthermore, even if page A and page B are in fact identical pages, which in theory should make all attacks unable to distinguish between them, the traces for page A can look completely different than those for page B—perhaps because the network is heavily congested when page B's traces are collected—making it trivial for any attack to differentiate the two pages.

As a result of this examination, we rerun the Tor experiments with a modified trace collection strategy that is not optimal for the attacker: the traces are collected in 80 rounds; in each round, only one trace is collected per page. Due to time constraints, we experiment with only a few select values of L. Table 4.5 summarizes the results.

First, consider the case where there is no CS-Tamaraw, i.e., the client and server applications defend themselves. When $L = 0$, i.e., all defensive mechanisms are disabled, we see that **k-FP Down-IDS** accuracy is 83%, which generally is considered a successful attack. We also see that **k-FP Full** accuracy is only 45%, but we have grown to expect this from earlier experiments—likely because **k-FP Timing** is only at 25% accuracy. When defensive padding is enabled, for all of $L = 150$, $L = 250$, and $L = 500$, we can see clearly that **k-FP Down-IDS** accuracies closely track the theoretical bounds! And as before, **k-FP Full** performance is lower than **k-FP Down-IDS** likely because of weakness in **k-FP Timing**.

L	theoretical	client & server defend			CS-Tamaraw defends		
		k-FP Full	**k-FP Down-IDS**	**k-FP Timing**	**k-FP Full**	**k-FP Down-IDS**	**k-FP Timing**
0	100	45.3	83.0	25.0	-	-	-
150	44	23.7	41.5	18.0	-	-	-
250	34	16.9	32.7	13.8	26.6	24.7	27.9
500	20	12.8	19.7	12.3	-	-	-

Table 4.5: k-fingerprinting attack accuracies for select values of L when the client and server applications defend themselves, and when CS-Tamaraw provides the defense instead (we have only one traceset in this case due to time constraints). In either case, the defended channel is tunneled through Tor. The traces are collected in rounds, where each round collects **only one trace** each for the 50 pages. $L = 0$ means all defensive mechanisms are disabled.

Now consider the case where CS-Tamaraw provides the defense. At $L = 250$ we see that **k-FP Down-IDS** accuracy is significantly less than when CS-Tamaraw is not in place; however, **k-FP Timing** seems to have made up for some of the slack, enabling **k-FP Full** to attain higher accuracy than when CS-Tamaraw is not used, yet at 26.6% is still less than the theoretical bound, and compared to more than 85% as in Figure 4.10.

With only one data point ($L = 250$), we cannot generalize the behavior of CS-Tamaraw under the modified trace collection strategy. At the same time, when considered along with all the other evidence thus far, this result suggests that the unexpected high accuracies that k-fingerprinting has been able to achieve against both types of defenses when tunneled over Tor have been partly due to artifacts of the experiment methodology, and not due to flaws in the design or implementation of CS-Tamaraw.

4.5 Large-scale simulations with Shadow

In this section we discuss simulations of CS-Tamaraw-over-Tor using Shadow, a simulation toolkit tailored for large-scale Tor network simulations, due to Jansen and Hopper [19]. All our Shadow simulations are run on machines in Emulab [36] with large amounts of RAM.

Before proceeding, we want to mention a key flaw in our simulations: they contain a bug where the browser, as soon as the page load has finished,

immediately instructs its client-side proxy to tear down the CS-Tamaraw channel, instead of allowing the channel to continue operating to completion, i.e., satisfy L. This means that our performance metrics are most likely exaggerated. Unfortunately, we discover this flaw after the experiments but are not able to address it due to time constraints. This is because fixing the bug requires a non-trivial amount of design and engineering work to enable the proxies to support potentially overlapping defense sessions that can happen on a short think-time pause.[9]

4.5.1 Shadow

Shadow is a discrete-event simulator that can run real applications, which are referred to as "plugins." Multiple simulated nodes can use the same Shadow plugin, sharing the plugin's code, while Shadow maintains separate copies of each node's application/plugin state. By the function interposition technique, the simulator redirects system calls such as `socket()`, `connect()`, `send()`, and `recv()`, etc., made by the nodes to its own implementations, allowing it to simulate the network environment, e.g., TCP stack, link bandwidth, delay, etc. Being targeted at Tor studies, Shadow provides the `shadow-plugin-tor`,[10] which includes the necessary modifications to the Tor code in order to work in Shadow simulation environment.

However, as we have touched on previously, running real, moderately complex applications in Shadow requires great care, if not impossible without major modifications. This is because Shadow supports only a small of set of system calls, and even those that are supported can have buggy implementations. Therefore, Shadow is only really recommended for applications one writes from scratch or those that use the set of basic system calls that has been well tested. An alternative to Shadow is the Direct Code Execution framework in ns-3 [32], which can use the more mature implementations of network stacks and protocols that are in ns-3; however, we do not have any experience using that framework.

[9] in the live experiments, each page load sample is isolated, so we can simply launch a new client-side proxy for each load.

[10] https://github.com/shadow/shadow-plugin-tor

4.5.2 Setup and methodology

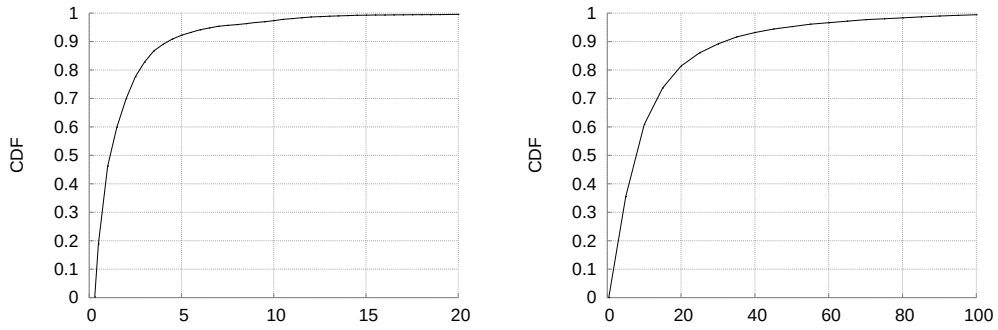
Page models We attempt to keep page models on servers in countries that reflect the real life locations of the original web sites. For example, for `ameblo.jp`, two of the hostnames used in the original page model are `act.ameba.jp` and `sy.amebame.com`. In the page model for Shadow, we map `act.ameba.jp` and `sy.amebame.com` randomly to, for example, virtual hostnames `serverX` and `serverY` that are located in Japan. If there are fewer servers available in Japan, then we sample with replacement, so the two original hostnames might be mapped to the same virtual hostname.

Establish the baseline network model A Shadow simulation consists of a scaled down version of the Tor network, including directory authorities, guards, relays, exits, and even optionally bridge nodes, etc. It also consists of clients and servers. There are two main types of clients: web clients load web pages, with some distribution of pauses—"think times"—between consecutive page loads, and bulk clients simply transfer large files repeatedly without pauses.

Before embarking on experiments with new protocols, we need to establish a baseline network model, i.e., arrive at a "reasonable" model of the network, including the number of Tor nodes, the number of web clients and bulk clients, servers, traffic generation pattern, etc. Since accurately modeling the Tor network is itself an open research topic, a proxy measure of reasonableness commonly used in the literature is the proportion of web traffic and bulk traffic (in total byte counts). In other words, through trial-and-error, one finds the set of parameters that results in the desired ratio of web traffic to bulk traffic. For this purpose, we target the 36% web traffic as Chaabane et al. reported in September 2010 [9], which was the most recent data available[11] in Fall 2016 when we began these experiments. Note that we cannot simply use the network models that have been used in published studies because they are based on coarse-grained page models that are single files, typically smaller than 500 KB [19, 21, 22].

After several tries, we decide to use a network of 100 Tor nodes (12 of which are exit nodes), 350 web clients, 150 bulk clients, and 200 servers. Each server node runs both our server simulator plugin to serve web traffic,

[11] Jansen and Johnson reported newer data at CCS 2016 [20].



(a) Time to first byte

(b) Page load time

Figure 4.13: CDF for time to first byte and page load time of web page loads in the baseline Shadow experiment.

as well as the `tgen` plugin[12] to serve bulk downloads. The bulk clients use `tgen` to repeatedly download a 5 MB file, selecting a random server for each download. The web clients use our browser simulator plugin to load the 50 page models, with uniform think times in $[20, 40]$ seconds between page loads, and 120 seconds timeout for each page load. The hosts are scattered across the (virtual) globe, in different countries and US states; this is taken care of by the topology generator script provided by Shadow, which also scales virtual network capacity accordingly. Each experiment runs for one virtual hour, with all clients actively generating traffic by virtual minute 30.

In the end, web traffic accounts for 28.7% of the total bytes received by all clients (10.8 GiB out of 37.5 GiB), and bulk the remaining 71.3%. While this is not the 36% that we target, in the interest of time, we decide this was sufficient.[13] There were 8 102 successful page loads, and 138 timed out. We show the CDF of the time to first byte and page load time in Figure 4.13. The average time to first byte is 2.1 seconds, with median and 90th percentile at 1.2 second and 4.3 seconds, respectively. For the page load time, the average is 13.6 seconds, the median is 7.4 seconds, and the 90th percentile is 31.5 seconds.

Parameter space We explore the space of the main CS-Tamaraw parame-

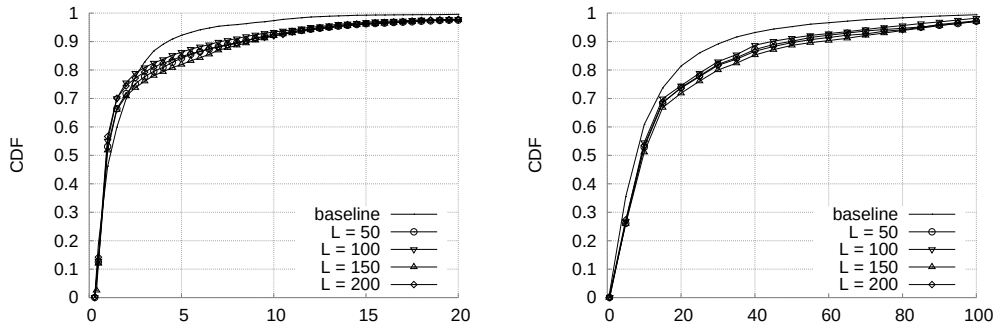
[12] https://github.com/shadow/shadow/tree/master/src/plugin/shadow-plugin-tgen

[13] As with all experiments, the larger the scale, the better; thus, we also attempted to use a network with 300 Tor nodes and 1 000 clients, but the simulation would stall for unknown reasons (i.e., the machine still had majority of its RAM available and was otherwise behaving normally) during the early phase where the Tor network is coming online, so we had to abandon this approach.

ters, namely p_{in}, p_{out}, and L. For p_{in}, we use 5, 20, and 50; larger values are of less interest because they would likely inflate page load times beyond usable realm. For p_{out}, we pick among 20, 50, 75, and 100. We do not experiment with configurations where $p_{in} > p_{out}$: because web traffic is download-heavy, it is generally not useful to throttle download traffic more than upload traffic. For L, we consider 50, 100, 150, and 200.

Client adoption rates Because CS-Tamaraw requires non-trivial bandwidth overhead, we also study the effects on the network by different CS-Tamaraw client adoption rates. In particular, we consider five scenarios: 100-0, 95-5, 75-25, 50-50, and 0-100, where a scenario "X-Y" means X% of the clients are "vanilla" clients who do not use CS-Tamaraw, and Y% of the clients use CS-Tamaraw. (In particular, 100-0 is the baseline experiment.) During an experiment run, each client's mode is fixed, either vanilla or CS-Tamaraw—it does not switch between the two modes. For each adoption scenario, we randomly pick a sample of the corresponding size from the 350 web clients and make them vanilla clients, and the remaining clients will use CS-Tamaraw. We perform this vanilla-vs-CS-Tamaraw client assignment once for each adoption scenario and use it in all parameter-space experiments for that scenario.

Metrics We are interested in the usual metrics such as time to first byte and page load time for web page loads, and bandwidth overheads. In addition, other metrics we will be looking into include the total number of page loads (specifically, the number of attempts), the number/ratio of successful page loads, and the unsuccessful ones. For unsuccessful page loads, we distinguish between hard failures where the main resources fail to download, and time-outs. To assess the value of congestion-sensitivity, we are interested in the statistics for the dummy cells that can be avoided; alas, the experiments are run with the flaw described in Section 4.3.4 that undercounts the number of avoided dummy cells. All reported metrics are only for the second half (30 minutes) of each experiment, after all clients have started to generate traffic.



(a) Time to first byte (seconds)

(b) Page load time (seconds)

Figure 4.14: CDF of the time to first byte and page load time for different values of parameter L when CS-Tamaraw adoption is at 100% and $p_{in} = 5, p_{out} = 20$, compared to the baseline (i.e., 0% CS-Tamaraw adoption).

4.5.3 Results

Full adoption We first look at the results for the scenarios that improve overall privacy the most: when all clients adopt CS-Tamaraw; and at the same time, all clients want to maximize their web browsing performance, so all use the fastest sending rates, i.e., $p_{in} = 5, p_{out} = 20$. Figure 4.14 shows the time to first byte and page load time for all successfully web page loads during the experiments with L from 50 to 200. The two key observations here are: first, our expectation is that time to first byte and page load time when using CS-Tamaraw will be higher than in the baseline, and this is mostly true (except for the time to first byte, where CS-Tamaraw scenarios yield slightly lower values up until approximately the 75th percentile).

Second, the four CS-Tamaraw experiments with different L values do not differ significantly in performance. This is likely due to the aforementioned bug in our experiments where the browser instructs its client-side proxy to tear down the CS-Tamaraw channel prematurely as soon as the page load is done. Even if we were not affected by this bug, consider $L = 50$ and $L = 200$: the additional bandwidth required by $L = 200$ would likely be on the order of several hundred KiB because $2 * ((200 - 50) * 750) = 220\,\text{KiB}$ (the 2 multiplier accounts for both directions of traffic), which is relatively small compared to all the other traffic generated during the page load as well as other traffic such as bulk downloads.

In Table 4.6, we show other useful statistics regarding page load counts and

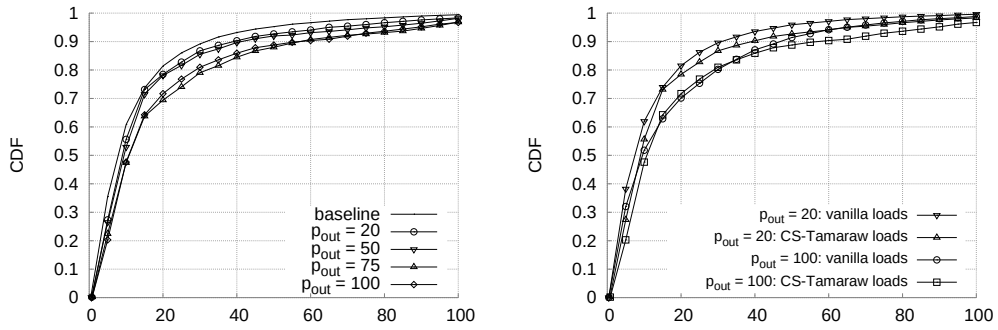
L	Page loads per client				SSP byte overhead (%)		
	Total	Failed	Timed out	Success ratio (%)	Send	Recv	Overall
baseline	23.5	0	0.39	98	-	-	-
50	14.4	0.55	4.13	67	167	3 340	242
100	15.2	1.29	3.89	66	146	3 067	211
150	14.4	0.69	3.98	68	178	3 619	259
200	14.4	0.51	4.14	68	147	3 147	215

Table 4.6: Page load counts and overall server-side proxy byte overhead (at the application layer, i.e., not including Tor overhead) for different values of parameter L when CS-Tamaraw adoption is at 100% and $p_{in} = 5, p_{out} = 20$, compared to the baseline (i.e., 0% CS-Tamaraw adoption).

bandwidth overhead. Once again, there is no clear pattern of differentiation among the four CS-Tamaraw scenarios. However, one thing is clear: overall, they all result in a fairly poor page load success rate: a full one third of page loads are unsuccessful—mostly due to reaching the two minute timeout—compared to just 2% unsuccessful page loads in the baseline. The overall byte overhead seen by the server-side proxies is larger than what we see in the live Tor experiments (Table 4.2).

25% adoption We now consider the more plausible adoption rate of 25%. We will fix $L = 50$ since L has little effect on our results (due to aforementioned flaw). We also fix $p_{in} = 5$, our fastest downstream rate. Now, we look at the network performance with four p_{out} values. First, Figure 4.15(a) shows the page load times of CS-Tamaraw page loads in the four experiments. We see that, except for the case of $p_{out} = 75$ experiment, as p_{out} increases, the page load time increases, as expected. For example, the median page load times are 8.5, 9.2, 10.7, and 10.7 seconds, respectively, as we increase p_{out}, and the 90th percentiles are 39.7, 41.2, 57.3, and 58.4 seconds, respectively. Recall that the corresponding values in the baseline scenario are 7.4 and 31.5 seconds.

Next, we look at Figure 4.15(b), which compares the page load times of CS-Tamaraw page loads to vanilla loads in the same experiment, for two experiments with $p_{out} = 20$ and $p_{out} = 100$. As expected, vanilla page loads



(a) Page load time (seconds)

(b) Page load time (seconds)

Figure 4.15: CDF of page load time for different values of parameter p_{out} when CS-Tamaraw adoption is at 25% and $p_{in} = 5, L = 50$. Figure (a) shows page load times of only CS-Tamaraw page loads from four experiments in addition to the vanilla page loads from the baseline experiment. Figure (b) shows page load times of CS-Tamaraw page loads compared to vanilla loads in two experiments, for two values of $p_{out} = 20$ and $p_{out} = 100$.

are generally faster than CS-Tamaraw ones; for example, with $p_{out} = 20$, the median and 90th percentile page load time for vanilla page loads are 7 and 31.1 seconds, respectively, and the corresponding values for CS-Tamaraw page loads are 8.5 and 39.7 seconds.

In Table 4.7, we show other useful statistics regarding page load counts and bandwidth overhead. As we increase p_{out}, web pages take longer to load, which explains the decreasing number of page loads per CS-Tamaraw client; for the same reason, more page loads time out, and the page load success ratios decline with larger p_{out}. (The page load failure rate of 0.53 when $p_{out} = 20$ appears to be an anomaly—compared to less than 0.06 for other experiments—but we do not have an insight into the cause.) Also, we see that as p_{out} increases, client-side proxies send less dummy traffic, which is reflected in the decreasing receiving overhead at the server-side proxies; however, since the page loads overall take longer, and p_{in} is fixed, the server-side proxies sending overhead grows, and the overall overhead grows as well, since $p_{in} = 5$ means that server-side proxies send data much faster than client-side proxies.

Next, we see that the additional bandwidth utilization by CS-Tamaraw negatively affects vanilla clients, reducing the number of page loads they can complete. However, the reduction is small and acceptable if it is traded for

p_{out}	Page loads per CS-Tamaraw client				Page loads per vanilla client		SSP byte overhead (%)		
	Total	Failed	Timed out	Success ratio (%)	Total	Success ratio (%)	Send	Recv	Overall
baseline	-	-	-	-	23.5	98.3	-	-	-
20	17.5	0.53	2.78	81.1	22.2	95.4	139	2720	198
50	16.8	0.05	3.05	81.6	22.3	97.4	188	1688	221
75	14.6	0	3.80	74.0	21.4	98.3	318	1912	354
100	15.2	0.06	3.40	77.2	20.6	96.9	290	1339	313

Table 4.7: Per-client page load counts and overall server-side proxy byte overhead (at the application layer, i.e., not including Tor overhead) for different values of parameter p_{out} when CS-Tamaraw adoption is at 25% and $p_{in} = 5, L = 50$.

significantly increased security of CS-Tamaraw page loads. Finally, we also mention that across these four experiments, the server-side proxies are able to avoid sending between 120 and 350 dummy cells, which is a small fraction of the more than 1 200 page loads attempted by CS-Tamaraw clients, but once again, these are underestimates due to the aforementioned bug.

CHAPTER 5

CONCLUSION

Over the last 15 years, the Internet has grown to become an integral part of modern life. It has democratized access to information, knowledge, and communication. It has made possible the streamlining of many interactions between citizens and governments, between consumers and businesses, etc. However, as we migrate more and more of our daily lives into the digital world, there are serious privacy implications. Whereas it is impossible to conduct large-scale monitoring and surveillance in the physical world, it is easy to do so in the digital world: monitor our Internet connections, where a lot of our daily activities are happening. For example, merely the metadata such as the identities of websites we visit can reveal sensitive personal information. Browsing the Internet over encrypting proxies such as Tor can help mask the identities of the websites, but only to a certain extent: each website can have a distinctive network traffic fingerprint that gives it away even visited over an encrypted proxy connection.

After earlier ad hoc attempts to mask websites traffic fingerprints are found ineffective against more advanced attacks, researchers have proposed a family of provable defenses called BuFLO, or Buffered Fixed-Length Obfuscator, where the encrypted proxy channel transmits fixed-size packets at predetermined schedules, using dummy padding data if needed. Packet-simulation studies have shown the BuFLO-style defense Tamaraw to have the strongest security properties, albeit at the expense of high overhead. In this dissertation, we propose to use CS-Tamaraw, our version of Tamaraw with congestion-sensitivity—which itself is borrowed from CS-BuFLO—to protect web browsing traffic over Tor.

In order to achieve a high level of reproducibility for real life experiments as well as conduct large-scale Tor simulation studies, we want to construct precise models of the web page loading process. We need an unprecedented level of modeling precision because that is inherent in this research area:

state-of-the-art website traffic fingerprinting attacks are adept at picking out minute differences in network traces. (Another context where precise web page traffic models are useful is censorship circumvention; they would enable certain circumvention systems to generate traffic patterns according to the model of an allowed website while in reality are loading another arbitrary website.) Essentially, details such as computing delays, network request patterns, inter-dependencies between resources, browser design and implementation idiosyncrasies, as well as opaque web page dynamism and application logic, etc. are all crucial and have to be reflected in the models. To do this, we perform an extensive study of the Google Chrome browser to understand as much as we can about how it loads a web page. We start with only a basic high-level understanding of HTML, and learn more as we dig deeper and deeper into the Chrome codebase. After a while, we realize that our goal is unattainable: we keep discovering more and more complex behavior as we test with more live web pages. (In retrospect, it should have been obvious from the beginning.)

At this point, we proceed to use what we have and try to model the top 100 Alexa-ranked global web pages. We instrument Google Chrome to log information about the page loading process, and build a tool to parse the log to produce the model. Alas, many of these web pages trigger assertions we introduce into the Chrome code and/or the log parser to catch code paths or features that we have not covered. As a result, the process requires extensive manual inspection and working around such assertions if reasonable. In the end we obtain the models for 50 of the 100 pages. We then implement a traffic generator framework that includes: 1) a browser emulator modeled after Chrome that generates network requests and emulates computing delays as specified in the input model; 2) a simple server emulator that simply sends back to the browser the amount of the data specified in the browser's request.

We take a short detour here to emphasize that it is a major engineering effort to build the traffic generator framework. This is because we want to use the same codebase as native applications as well as for simulation using the Shadow simulator. Shadow uses function interposition to redirect system calls made by applications to its own implementations, enabling it to provide a simulated network environment. In theory, this would allow real, unmodified applications to run in Shadow; however, we find that Shadow does not support many system calls, as well as in some cases has buggy

implementations. Essentially this prevents us from using any of the rich software libraries that provide high level abstractions for networking, multi-threading, inter-process communication, etc. because we lack visibility into their usage of system calls. (In fact, we discover these issues after starting to use one such library and running into incorrect behavior.) Consequently, we have to reinvent the wheel and build on top of only a small set of system calls that have been used extensively in other Shadow applications.

With the models and the traffic generator, we proceed to validate the models. As part of the initial phase of loading the original web pages in order to extract the browser logs, we already collect network traces generated by the real browser loading the real pages; we call these the real traces. Now, we use our traffic generator to load the page models, and collect the network traces; we call these the generated traces. To validate the models, we use the state-of-the-art fingerprinting attack k-fingerprinting as follows. First, we use k-fingerprinting to perform an attack on the real traces, and then separately an attack on the generated traces. In both cases, k-fingerprinting attack accuracies are above 98%, as expected. Then, to validate the page models, we train k-fingerprinting on the real traces, and test on the generated traces; the attack accuracy is only 13%, which means that our generated traces are far from resembling their respective real traces. This is unsurprising given the challenging task of modeling the web page loading process.

Nevertheless, we proceed to propose the CS-Tamaraw design to protect web browsing traffic over Tor: simply tunnel the CS-Tamaraw channel through the Tor circuit. The CS-Tamaraw client-side proxy is located at the client as usual, and the server-side proxy is co-located with the Tor exit. Because the CS-Tamaraw channel removes all timing information from the web page load, we expect that it holds true regardless of the underlying transport, whether it is a direct TCP connection, or tunneled through Tor. We design and implement from scratch the CS-Tamaraw protocol and proxies, and then proceed to conduct live experiments over Tor fetching the 50 page models to understand the performance-vs-security tradeoffs. However, we discover that as we increase the security parameter L significantly, it results in negligible gains security, i.e., k-fingerprinting still achieves high attack success rates with large values of L.

Subsequently, we perform a series of experiments including starting from the absolute minimal client-server setup where each page load simply involves

the client's sending a single request for a given number of bytes, the server sends back that exact number of bytes, and the connection is torn down. The client controls the defense, by padding the requested amount based on parameter L. We see that this setup results in largely expected behavior: as L increases, k-fingerprinting attack accuracy significantly reduces. Another set of experiment involves the same client and server, but the client does not perform padding; instead, the client-server connection is tunneled through CS-Tamaraw proxies, which provide the protection. The result here is also as expected: as L increases, CS-Tamaraw is able to significantly reduce k-fingerprinting attack accuracy.

However, when these two experiments are then extended by tunneling the client-server connection and the CS-Tamaraw channel, respectively, through Tor, we consistently see that their security is significantly diminished, i.e., k-fingerprinting is able to maintain high attack accuracies as L increases. In particular, when $L = 500$, a theoretical attacker that has only information on the total size of the server's response, should achieve average accuracy of 20%; however, in either experiment, k-fingerprinting can still attain above 80% attack accuracy. As we isolate the features that k-fingerprinting uses, we discover that packet timing information alone can achieve close to the accuracy of the full k-fingerprinting. We speculate that a possible cause is transient changes in network conditions; for example, two pages that are identical in size can appear distinctive if the network is more congested when one page is loaded compared to when the other is.

This prompts us to re-examine our trace collection methodology, and we realize that we are giving the attack an unlikely advantage. We rerun the Tor experiments with a modified trace collection strategy that does not give the attacker such a big advantage. While not at the scale that would allow us to make a stronger claim, the results of these experiments suggest that the unexpected high accuracies that k-fingerprinting has been able to achieve against both types of defenses when tunneled over Tor have been partly due to artifacts of the earlier trace collection methodology, and not due to flaws in the design or implementation of CS-Tamaraw.

We briefly discuss our whole-network simulations using Shadow. in our experiments with 100 Tor nodes, 350 web clients, and 150 bulk clients, full adoption of CS-Tamaraw by web clients results in poor and likely unacceptable web browsing performance. At 25% adoption, the CS-Tamaraw clients

can achieve much more reasonable web browsing performance, while at the same time not significantly impacting the rest of the web clients. Since our simulations contain a key bug, the performance results are likely exaggerated. It will be interesting future work to address the issue and perform a more comprehensive analysis.

REFERENCES

[1] United states speedtest market report, . http://www.speedtest.net/reports/united-states/. Accessed 01/03/2017.

[2] Top 9 browsers on dec 2016, . http://gs.statcounter.com/#all-browser-ww-monthly-201612-201612-bar. Accessed 03/21/2017.

[3] Experimental defense for website traffic fingerprinting. https://blog.torproject.org/blog/experimental-defense-website-traffic-fingerprinting. Accessed 09/18/2015.

[4] M. AlSabah and I. Goldberg. Performance and security improvements for tor: A survey. Cryptology ePrint Archive, Report 2015/235, 2015. http://eprint.iacr.org/.

[5] G. D. Bissias, M. Liberatore, D. Jensen, and B. N. Levine. Privacy vulnerabilities in encrypted http streams. In *Proceedings of the 5th International Conference on Privacy Enhancing Technologies*, PET'05, pages 1–11, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-34745-3, 978-3-540-34745-3. doi: 10.1007/11767831_1. URL http://dx.doi.org/10.1007/11767831_1.

[6] 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*, CCS '12, pages 605–616, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382260. URL http://doi.acm.org/10.1145/2382196.2382260.

[7] 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*, WPES '14, pages 121–130, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3148-7. doi: 10.1145/2665943.2665949. URL http://doi.acm.org/10.1145/2665943.2665949.

[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*, CCS '14, pages 227–238, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660362. URL http://doi.acm.org/10.1145/2660267.2660362.

[9] A. Chaabane, P. Manils, and M. A. Kaafar. Digging into anonymous traffic: A deep analysis of the tor anonymizing network. In *Proceedings of the 2010 Fourth International Conference on Network and System Security*, NSS '10, pages 167–174, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4159-4. doi: 10.1109/NSS.2010.47. URL http://dx.doi.org/10.1109/NSS.2010.47.

[10] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: a reality today, a challenge tomorrow. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*. IEEE Computer Society, May 2010. URL http://research.microsoft.com/apps/pubs/default.aspx?id=119060.

[11] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type iii anonymous remailer protocol. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 2–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1940-7. URL http://dl.acm.org/citation.cfm?id=829515.830555.

[12] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251375.1251396.

[13] K. Dyer, S. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 332–346, May 2012. doi: 10.1109/SP.2012.28.

[14] X. Fu, B. Graham, R. Bettati, and W. Zhao. On countermeasures to traffic analysis attacks. In *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society*, pages 188–195, June 2003. doi: 10.1109/SMCSIA.2003.1232420.

[15] C. Gulcu and G. Tsudik. Mixing email with babel. In *Proceedings of the 1996 Symposium on Network and Distributed System Security (SNDSS '96)*, SNDSS '96, pages 2–, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7222-6. URL http://dl.acm.org/citation.cfm?id=525423.830456.

[16] J. Hayes and G. Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1187–1203, Austin, TX, 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/hayes.

[17] D. Herrmann, R. Wendolsky, and H. Federrath. Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *CCSW '09: ACM Workshop on Cloud Computing Security*, pages 31–42, November 2009. URL http://epub.uni-regensburg.de/11919/.

[18] A. Hintz. Fingerprinting websites using traffic analysis. In R. Dingledine and P. Syverson, editors, *Proceedings of Privacy Enhancing Technologies workshop (PET 2002)*. Springer-Verlag, LNCS 2482, April 2002.

[19] R. Jansen and N. Hopper. Shadow: Running tor in a box for accurate and efficient experimentation. In *NDSS*. The Internet Society, 2012. URL http://dblp.uni-trier.de/db/conf/ndss/ndss2012.html#JansenH12.

[20] R. Jansen and A. Johnson. Safely measuring tor. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1553–1567, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978310. URL http://doi.acm.org/10.1145/2976749.2978310.

[21] R. Jansen, P. Syverson, and N. Hopper. Throttling tor bandwidth parasites. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2362793.2362811.

[22] R. Jansen, J. Geddes, C. Wacek, M. Sherr, and P. Syverson. Never been kist: Tor's congestion management blossoms with kernel-informed socket transport. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 127–142, San Diego, CA, 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/jansen.

[23] 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*, CCS '14, pages 263–274, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660368. URL http://doi.acm.org/10.1145/2660267.2660368.

[24] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. Webprophet: Automating performance prediction for web services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855711.1855721.

[25] 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*, CCS '06, pages 255–263, New York, NY, USA, 2006. ACM. ISBN 1-59593-518-5. doi: 10.1145/1180405.1180437. URL http://doi.acm.org/10.1145/1180405.1180437.

[26] X. Luo, P. Zhou, E. W. W. Chan, W. Lee, R. K. C. Chang, and R. Perdisci. Httpos: Sealing information leaks with browser-side obfuscation of encrypted flows. In *In Proc. Network and Distributed Systems Symposium (NDSS). The Internet Society*, 2011.

[27] D. Mccoy, K. Bauer, D. Grunwald, T. Kohno, and D. Sicker. Shining light in dark places: Understanding the tor network. In *Proceedings of the 8th international symposium on Privacy Enhancing Technologies*, PETS '08, pages 63–76, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70629-8. doi: 10.1007/978-3-540-70630-4_5. URL http://dx.doi.org/10.1007/978-3-540-70630-4_5.

[28] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 123–136, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-29-4. URL http://dl.acm.org/citation.cfm?id=2930611.2930620.

[29] R. Nithyanand, X. Cai, and R. Johnson. Glove: A bespoke website fingerprinting defense. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, WPES '14, pages 131–134, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3148-7. doi: 10.1145/2665943.2665950. URL http://doi.acm.org/10.1145/2665943.2665950.

[30] 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*, WPES '11, pages 103–114. ACM, 2011. ISBN 978-1-4503-1002-4. doi: 10.1145/2046556.2046570. URL http://doi.acm.org/10.1145/2046556.2046570.

[31] 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 of the 2002 IEEE Symposium on Security and Privacy*, SP '02, pages 19–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1543-6. URL http://dl.acm.org/citation.cfm?id=829514.830535.

[32] H. Tazaki, F. Uarbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous. Direct code execution: Revisiting library os architecture for reproducible network experiments. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 217–228, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2101-3. doi: 10.1145/2535372.2535374. URL http://doi.acm.org/10.1145/2535372.2535374.

[33] 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*, WPES '13, pages 201–212, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2485-4. doi: 10.1145/2517840.2517851. URL http://doi.acm.org/10.1145/2517840.2517851.

[34] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective attacks and provable defenses for website fingerprinting. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 143–157, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-15-7. URL http://dl.acm.org/citation.cfm?id=2671225.2671235.

[35] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystify page load performance with wprof. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[36] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[37] C. V. Wright, S. E. Coull, and F. Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *In Proceedings of the 16th Network and Distributed Security Symposium*, pages 237–250. IEEE, 2009.

APPENDIX A

MINIMAL CLIENT AND SERVER CODE

Listing 9 Simple server code that sends back precisely the amount of data that the client requests.

```
PORT = 3333
BUF_SIZE = 4096
BUF = 'B' * BUF_SIZE

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(('', PORT))
s.listen(1)

while True:
    try:
        conn, _ = s.accept()

        # read cmd from client
        request_str = conn.recv(10)
        size = int(request_str)
        assert size > 0, "bad size {}".format(size)

        remaining = size
        while remaining > 0:
            to_send = min(remaining, BUF_SIZE)
            sent = conn.send(BUF[:to_send])
            remaining -= sent

        # done sending... wait for client to close
        conn.recv(1)
    except Exception as exc:
        print('ERROR: {}'.format(exc))
```

Listing 10 Simple client code that rounds the requested amount of data and also pads the request message.

```
PORT = 3333
BUF_SIZE = 4096
VALID_BUCKET_WIDTHS = {750*(i*50) for i in range(0, 11)}

def compute_new_size(bucket_width, size):
    return int(math.ceil(float(size) / bucket_width)) * bucket_width

server_host = sys.argv[1]
bucket_width = int(sys.argv[2])
size = int(sys.argv[3])

if bucket_width:
    size = compute_new_size(bucket_width, size)

request_str = '{:010d}'.format(size)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((server_host, PORT))
s.send(request_str.encode())

recv_size = 0
while True:
    data = s.recv(BUF_SIZE)
    recv_size += len(data)
    if recv_size > size:
        # break here and we will error
        break
    elif recv_size == size:
        break

if recv_size == size:
    print('OK')
    sys.exit(0)
else:
    print('ERROR: requested= {} but got {}'.format(size, recv_size))
    sys.exit(1)
```

APPENDIX B

OUR TOR EXIT NODE ON AMAZON EC2

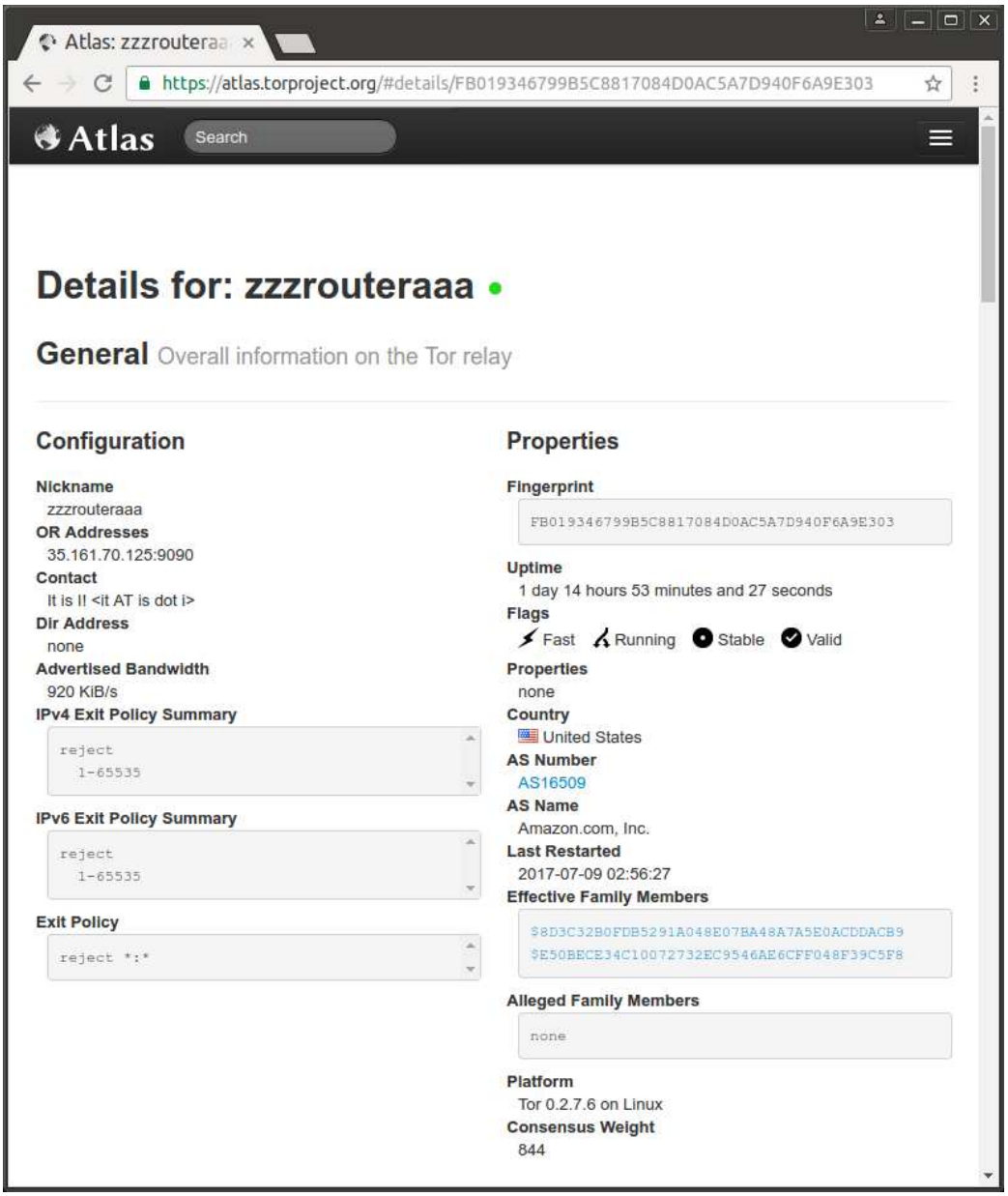


Figure B.1: Description of our primary Tor "exit" relay `zzzrouteraaa`.