Practical Countermeasures against Network Censorship

by

Sergey Frolov

B.S.I.T., Lobachevsky State University, 2015

M.S.C.S., University of Colorado, 2017

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2020

Frolov, Sergey (Ph.D., Computer Science)

Practical Countermeasures against Network Censorship

Thesis directed by Prof. Eric Wustrow

Governments around the world threaten free communication on the Internet by building increasingly complex systems to carry out Network Censorship. Network Censorship undermines citizens' ability to access websites and services of their preference, damages freedom of the press and self-expression, and threatens public safety, motivating the development of censorship circumvention tools.

Inevitably, censors respond by detecting and blocking those tools, using a wide range of techniques including Enumeration Attacks, Deep Packet Inspection, Traffic Fingerprinting, and Active Probing. In this dissertation, I study some of the most common attacks, actually adopted by censors in practice, and propose novel attacks to assist in the development of defenses against them. I describe practical countermeasures against those attacks, which often rely on empiric measurements of real-world data to maximize their efficiency. This dissertation also reports how this work has been successfully deployed to several popular censorship circumvention tools to help censored Internet users break free of the repressive information control.

Acknowledgements

I am thankful to many engineers and researchers from various organizations I had a pleasure to work with, including Google, Tor Project, Psiphon, Lantern, and several universities. This work would be impossible without the help of Nikita Borisov, Ox Cart, Katharine Daly, Fred Douglas, Roya Ensafi, David Fifield, Adam Fisk, Vinicius Fortuna, J Alex Halderman, Rod Hynes, Michalis Kallitsis, Allison McDonald, Will Scott, Steve Schultze, Ben Schwartz, Sze Chuen Tan, Benjamin VanderSloot, Jack Wampler, and last but certainly not least, my advisor Eric Wustrow. I want to thank the members of my committee for their guidance. I am also deeply grateful to Matthew Monaco, Andy Sayler, and Dirk Grunwald, who helped me get on my feet during the first months of my PhD program. Finally, I'd like to thank my parents and sisters for their support.

Contents

Tables

Figures

Chapter 1

Introduction

Network Censorship occurs when a *censor* prevents *clients* located within a network under the censor's control from communicating with *destination servers* of their choice. Censors install specialized hardware and software, either at the border of their network [7] or at the local ISP level [121], to monitor communications, detect usage of prohibited Internet resources and proxies, and *directly block* the desired endpoints. Prior to the development of censorship circumvention tools, blocked endpoints only included websites and servers, to which censors intended to deny access. Censorship circumvention tools have emerged to allow censored users to freely access the Internet by proxying clients' traffic to blocked destinations. Censors responded by developing *Censorship Attacks* that they can execute to increase the certainty or definitively confirm that a given endpoint is a proxy, and, thus, must be directly blocked.

An example of Internet Censorship capability that is not Network Censorship, and is out of the scope of this thesis, is Publisher-Side Censorship [83], which includes removal of politically charged content from on the publisher's platform as a result of censors' pressure [168] or preventing access by engaging in DDoS attacks [100].

1.1 Direct Blocking

IP blocking is an effective and straightforward censorship technique. Since the IP addresses of client and server are present in every IP packet unencrypted, censors will be able to block them reliably and efficiently. Censors can execute IP blocking either by simply dropping packets for any connection that involves a prohibited IP address, or, in case of TCP, by sending forged RST packets to the client and the server. Alternatively, censors may severely throttle the traffic, rendering access to blocked websites and services practically impossible, while potentially confusing the users and censorship measurement tools about the nature of the blocking.

The Domain Name System is used by networked devices to translate domain names into IP addresses. Despite the ubiquity of the protocol, DNS queries remain largely unencrypted and unauthenticated, which makes DNS poisoning one of the simplest methods of censorship. Censors can easily inject fabricated responses to DNS queries for blocked domain names, effectively redirecting visitors of censored websites to incorrect IP addresses, without affecting DNS queries for any other domains. Since publicly readable DNS queries and unauthenticated DNS responses also present a privacy and security issue, there is an ongoing effort to secure the DNS protocol, including DNS-over-HTTPS [68] and DNS-over-TLS [28], both of which naturally provide confidentiality and integrity. Google has announced that Android P [84] would include built-in support for DNS over TLS, while applications in the Google Play Store, such as Intra [79] may provide DNS-over-TLS functionality to earlier Android versions. Unfortunately, censors pay little cost for blocking services that provide DNS resolution over those secure channels, since regular users may always fall back to insecure DNS. Huang et al. demonstrated [77] that DNS-over-HTTPS, as currently implemented, could be easily downgraded to insecure DNS by censors. The high prevalence of DNS interference around the world is shown in Figure 1.1.

1.2 Censorship Attacks

Censorship circumvention tools use obfuscation techniques to evade blocking, while censors develop new attacks against those techniques. Anti-censorship researchers try to prevent blocking by proposing new defenses against known censorship attacks. Researchers also look for novel attacks to ensure that defenses against them could be built, hopefully, before censors get a chance to take advantage of those attacks. In the remainder of the Introduction, I will describe three attacks that censors are known to use: Enumeration, TLS Fingerprinting, and Active Probing. I will then summarize the novel attacks and defenses against them that I propose in my dissertation.

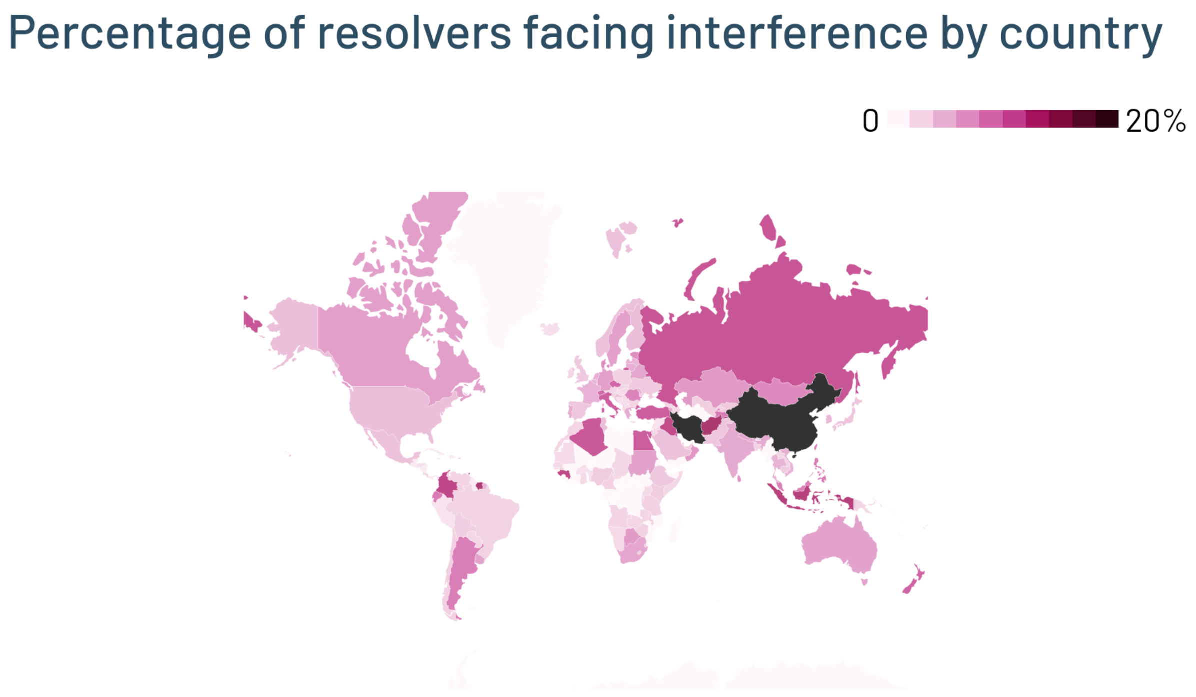


Figure 1.1: **Satellite Visualisation** — Satellite [135] detects DNS interference by comparing responses to the DNS queries made in a given country with responses from designated control resolvers. The list of tested domains includes the latest Alexa top 1k, Citizen Lab global test list, and a list of domains curated by Satellite maintainers.
Source: `https://censoredplanet.org/data/visualizations`. August 10th, 2020.

1.2.1 Enumeration

Enumeration is one of the most universally applicable attacks against censorship circumvention systems with public and free access, such as Tor. Since there is no known reliable way to tell legitimate users from censors, censors can simply reverse engineer [55] or launch circumvention tools, pretend to be a user, and connect to the proxy servers, thus revealing their IP address. Enumeration Attacks allow censors to reliably and conveniently attack public censorship circumvention systems without relying on any other sophisticated attacks, such as traffic fingerprinting or active probing.

I describe multiple known approaches that allow us to defend against Enumeration entirely or reduce its efficiency in Section 2.1. In this thesis, I focus on one of those approaches: Refraction Networking. Refraction Networking brings proxy functionality to the core of the network, through a partnership with ISPs and other network operators. As a result, anti-censorship tools get to use innocuous, unblocked websites as an ostensible destination, while users' traffic is rerouted to the desired destination.

In section 2.2, I report initial results from the world's first ISP-scale field trial of a refraction networking system. The high-performance implementation of the TapDance was deployed on four ISP uplinks with an aggregate bandwidth of 100 Gbps and served more than 50,000 real users during one week of operation. This experience demonstrated the possibility of the TapDance deployment at the ISP scale in practice, achieving reasonable performance at a reasonable cost, potentially paving the further deployments of refraction networking schemes in the future.

However, TapDance suffers from several problems: a limited number of "decoy" sites in realistic deployments, high technical complexity, and undesirable tradeoffs between performance and observability by the censor. These challenges may impede broader deployment and ultimately allow censors to block such techniques. In section 2.3, I present Conjure, an improved Refraction Networking approach that overcomes these limitations by leveraging unused address space at deploying ISPs. Instead of using real websites as the decoy destinations for proxy connections, Conjure client connects to IP addresses where no web server exists. In the meantime, the Conjure station conjures a "phantom" host at that address to be used as a proxy. I describe the Conjure protocol, analyze its security, and evaluate a prototype using an ISP testbed. Our results suggest that Conjure is more difficult to block than TapDance and is simpler to maintain and deploy. Moreover, Conjure offers substantially better network performance.

1.2.2 TLS Fingerprinting

TLS, the Transport Layer Security protocol, has quickly become the most popular protocol on the Internet. As of July 2020, this protocol is used to load over 83% [142] of web pages in Mozilla Firefox.

Due to its ubiquity, TLS is also a popular protocol for censorship circumvention tools. However, the wide range of features supported in TLS makes it possible to distinguish implementations from one another by what set of cipher suites, elliptic curves, signature algorithms, and other extensions they support. Already, censors have used deep packet inspection (DPI) to identify and block popular circumvention tools based on the fingerprint of their TLS implementation. In response, many circumvention tools have attempted to mimic popular TLS implementations such as browsers, but this technique has several challenges. First, it is burdensome to keep up with the rapidly-changing browser TLS implementations and know what fingerprints would be good candidates to mimic. Second, it can be challenging to mimic TLS implementations correctly, as they offer many features that may not be supported by the relatively lightweight libraries used in typical circumvention tools. Finally, dependency changes and updates to the underlying libraries can silently impact what an application's TLS fingerprint looks like, making it difficult for tool maintainers to keep up. In section 3.2, I analyze real-world TLS traffic from over 11.8 billion TLS connections over nine months to identify a wide range of TLS client implementations used on the Internet.

To enable this study, I collect TLS packets from a 10 Gbps campus network and process them in real time to extract a TLS fingerprint for each connection, allowing us to group messages generated by the same implementation together. So far, I have recovered fingerprints from over 9.1 billion TLS connections over five months from real-world users.

I use the collected data to analyze TLS implementations of several popular censorship circumvention tools, including Lantern, Psiphon, Signal, Outline, TapDance, and Tor (Snowflake and meek pluggable transports). I find that many of these tools use TLS configurations that are easily distinguishable from the real-world traffic they attempt to mimic, even when these tools have put effort into parroting popular

TLS implementations. To address this problem, I have developed a library, uTLS, that enables censorship circumvention tool developers to automatically mimic other popular TLS implementations. Using our real-world traffic dataset, I observe many popular TLS implementations I can correctly mimic with uTLS. I describe ways our tool can more flexibly adapt to the dynamic TLS ecosystem with minimal manual effort. In addition to mimicry, the uTLS library can generate randomized fingerprints, defeating any censor unwilling to embark on a challenging task of composing and enforcing a list of allowed fingerprints. In addition to the results presented herein, I release a tool that allows others to explore our live data via a website: `https://tlsfingerprint.io`.

1.2.3 Active Probing

Active probing of suspected proxies allows censors to either definitively confirm or increase the certainty that a particular address is a circumvention proxy, decreasing the false-positive rate of network censorship.

There are multiple variations of Active Probing attacks. In one previously known variation, the censor identifies a connection that might be using a particular prohibited protocol, such as Tor, and sends a probe to the server trying to "speak" the prohibited protocol. If the server "speaks" the prohibited protocol back, then it is safe to block.

In response, anti-censorship developers have created new "probe-resistant" proxy protocols, including obfs4, Shadowsocks, and Lampshade, that attempt to prevent censors from discovering them. Client applications of those tools have to prove knowledge of a secret, shared with clients out of band, to receive responses from the server; otherwise, the servers will remain silent. As a result, probes by a censor that does not know the shared secret will be unable to confirm the protocol definitively.

In section 4.2, I evaluate another variation of active probing attack, in which censors may attempt to speak popular protocols or send randomized probes to check if the circumvention server responds in a way that is different from the majority of endpoints on the Internet. Suppose a circumvention proxy did not respond to these probes while keeping the connection open, and most servers on the Internet respond and close the connection. In that case, the censor gains additional confidence that the suspect endpoint is indeed a circumvention proxy. I also discover unique TCP behaviors of five probe-resistant protocols used in popular circumvention software that could allow censors to confirm suspected proxies with minimal false positives. I evaluate and analyze this attack on hundreds of thousands of servers collected from a 10 Gbps university ISP vantage point over several days and from active scanning using ZMap. I find that this attack can efficiently identify proxy servers with negligible false positives.

Using our datasets, I also suggest defenses to these attacks that make it harder for censors to distinguish proxies from other common servers. I work with proxy developers to implement these changes in several popular circumvention tools.

As a long-term defense, I recommend hiding circumvention proxies behind common server applications. This design will hide any potentially distinctive TCP behaviors of the circumvention proxy and provide the endpoint a plausible purpose, demonstrating potential collateral damage. In section 4.3, I implement and evaluate HTTPT – a prototype using this defense. Our prototype is designed to be hidden behind HTTPS servers to resist these active probing attacks. HTTPT leverages the HTTPS protocol's ubiquity to effectively blend in with Internet traffic, making it more difficult for censors to block. I describe the challenges that HTTPT must overcome, and the benefits it has over previous probe-resistant designs.

Chapter 2

Enumeration Attacks

2.1 Background

Enumeration Attacks allow censors to find the public endpoints belonging to a given censorship circumvention system and block them. As early as September 2009, China grabbed [32] the list of public Tor relays, as well as one of their sets of bridges, available at `https://bridges.torproject.org/`. Tor bridges are publicly distributed, and any requesting user will receive a limited amount of them; however, the entire list was supposed to remain secret. In March 2010, China enumerated the last publicly accessible at the time set of bridges, that was distributed over email. In 2016, David Fifield and Lynn Tsai [55] measured the time it took for the GFW to block the public Tor bridge addresses across the five batches of bridges and observed varying blocking delays of between 2 and 36 days after the first public release.

In the remainder of the section, I will describe known countermeasures against Enumeration Attacks.

Private Proxies

A simple way to defend against enumeration attacks on public proxies is to make them private. After launching a private proxy, users may distribute their proxy's address within their trusted social network or reserve it for personal use. A great way to make this approach more practical is to automate [80] deployment of personal private proxy servers in the cloud. As an added advantage, user-friendly personal proxy deployment allows clients to simply migrate to another IP address in the cloud when their proxy gets blocked. Unfortunately, social network distribution may be slow and limited, and personal VMs may be unaffordable for many censored users.

User Score–based Distribution

One way to curtail censors' ability to enumerate proxy endpoints is to assign the potential clients an individual trust score. Users would receive a certain amount of credit by performing trustworthy actions; for example, they may receive credits if servers given to them, remain unblocked. While those approaches are helpful, there are substantial fundamental limitations to them. Some of those systems, such as Proximax [103], rBridge [155], and Hyphae [96] are invite-only and do not provide an easy way for new untrusted users to join. Other approaches, such as Salmon [38], allow previously untrusted users to receive proxy addresses, which opens them to Sybil attacks and allows resourceful censors to curb the invitation of new users.

Ephemeral Proxies

Because there is an inevitable delay between the enumeration of a proxy server and its blocking, making use of short-lived ephemeral proxies is an exciting direction of research. While there are no fundamental reasons why censors would not eventually be able to propagate IP addresses that need to blocked across their censorship apparatus relatively quickly, this research is still beneficial. Sophisticated censors will have to redirect their attention from other tasks and spend resources on implementing quick propagation, while censors with lower levels of sophistication may be unable to do so at all in the near future.

David Fifield and Lynn Tsai found [55] that China had a diurnal pattern in the rates of accessibility of destinations that are supposed to be permanently blocked, meaning that there sometimes was up to 24 hours of free usage after detection before the endpoint finally gets blocked.

Flash proxy [53] is one of the first designs of a censorship circumvention system based on short-lived proxies. It works as follows: first, volunteers, located in uncensored regions, connect to the centralized *facilitator* and start continuously requesting IP addresses of clients who need free Internet access. Then, when clients contact the facilitator and request anti-censorship service, their IP addresses will be forwarded to volunteers, who can then connect to the clients directly and proxy their traffic. A crucial component of the system is easy registration of volunteers: to start proxying connections of censored users, one only has to visit a web page that includes the flash proxy script in a browser. This simple process enables owners of websites to easily help circumvent Internet censorship by adding flash proxy script to their website and bring a large number of volunteers into the system. Authors show that flash proxy remains functional and well-performing, even with unreliable volunteer proxies. After measuring the performance of proxies with a lifespan of only 10 seconds, the authors found that those proxies only decreased performance by 20 to 40 percent compared to direct Tor connections. Reliance on volunteers is simultaneously a challenge and an opportunity to dramatically decrease the price of operating a censorship circumvention system and amass a large number of volunteer proxies in diverse geographical locations. Flash proxy easily scales, automatically increasing the overall bandwidth of the system depending on the number of volunteers, only requiring a relatively inexpensive facilitator to arrange the connections.

However, flash proxy's design is not without shortcomings. First of all, it is possible to block facilitators so that the design will require an additional covert channel. Fortunately, since we only need to share a small amount of information over this covert channel, it is allowed to be expensive or have low bandwidth and reasonably high latency. Secondly, clients are required to have a public IP and a publicly reachable port, which is a massive limitation, since residential networks are frequently behind NAT.

Snowflake [36] is a direct successor of Flash proxy that aims to modernize and improve the design. Snowflake uses WebRTC with a built-in NAT traversal algorithm to no longer require clients to have public IP addresses. Usage of popular WebRTC protocol may help prevent blocking based on traffic shape, provided Snowflake's implementation of WebRTC has no observable differences from the common implementation in use. To prevent trivial blocking of the facilitator's address, Snowflake started tunneling connections between client and facilitator using Domain Fronting, which will be described in the next paragraph.

Domain Fronting

Domain Fronting [54] is an anti-censorship technique that relies on a tendency of Cloud Providers and Content Delivery Networks (CDNs) to route traffic based on an encrypted HTTP Host header, rather than TLS Subject Domain Name (SNI) extension, which is visible to the censor. This behavior allows anti-censorship tools to specify an arbitrary uncensored SNI and then set a different Host header to route requests to a proxy within the CDN without putting the address of a proxy into unencrypted SNI. As a result, the censor will have no direct way of blocking such connections without blocking an entire CDN, which may be deemed to be too much collateral damage.

Domain Fronting proved to be an effective way of circumventing censorship, albeit expensive. Figure 2.1 shows that meek, which is a domain fronting-based Pluggable Transport for Tor, accounts for only a fraction of overall Tor traffic that uses pluggable transports. However, figure 2.2 demonstrates that meek worked well against sophisticated censorship infrastructure, such as the Great Firewall of China, while other deployed transports had failed.

During Spring 2018, Domain Fronting suffered a setback, caused by an attack that was political rather than technical. Both Google and Amazon [98] made changes to their infrastructure and started requiring SNI and Host header to match, thereby preventing the original Domain Fronting design from working. Amazon-owned website Souq did not disable Domain Fronting fronting at the same time as Amazon-owned EC2; however, when the Signal messaging app tried to use Souq website for domain fronting, Amazon threatened to suspend Signal's AWS account [101]. Microsoft Azure and Akamai appear to be the only major cloud providers that still support domain fronting as of August 2020.

Some CDNs allowed Domain Fronting to work if anti-censorship tools omit SNI extension. This technique is called SNI-less Domain Fronting. However, our measurements of TLS ClientHellos coming in and out of Colorado University, reported in section 3.2.4, show that SNI-less connections are relatively rare and may not present sufficient collateral damage for censors to refrain from blocking them.

The future of Domain Fronting remains unclear with substantial dependence on the actions of CDNs and Cloud Providers, and their ability and desire to resist censors' pressure.

Refraction Networking

Refraction Networking is an umbrella term for a family of technologies that were first introduced in 2011 when Telex [165], Decoy Routing [82] and Cirripede [72] independently proposed an entirely new approach to anti-censorship. This approach aims to provide proxying services by placing censorship circumvention proxies in the middle of the network, such as at ISP/CDN/IXP premises, instead of operating proxies at network endpoints. Clients would establish a connection to one of the unblocked websites and attach a hidden tag (for example, in the TLS ClientHello) to signal their desire to be intercepted by a Refraction Networking station and disclose the TLS encryption key to the station. The station will be able to use that key to decrypt the connection between the client and reachable site, inject packets that look like they come from a reachable site, and start serving clients' requests as a proxy.

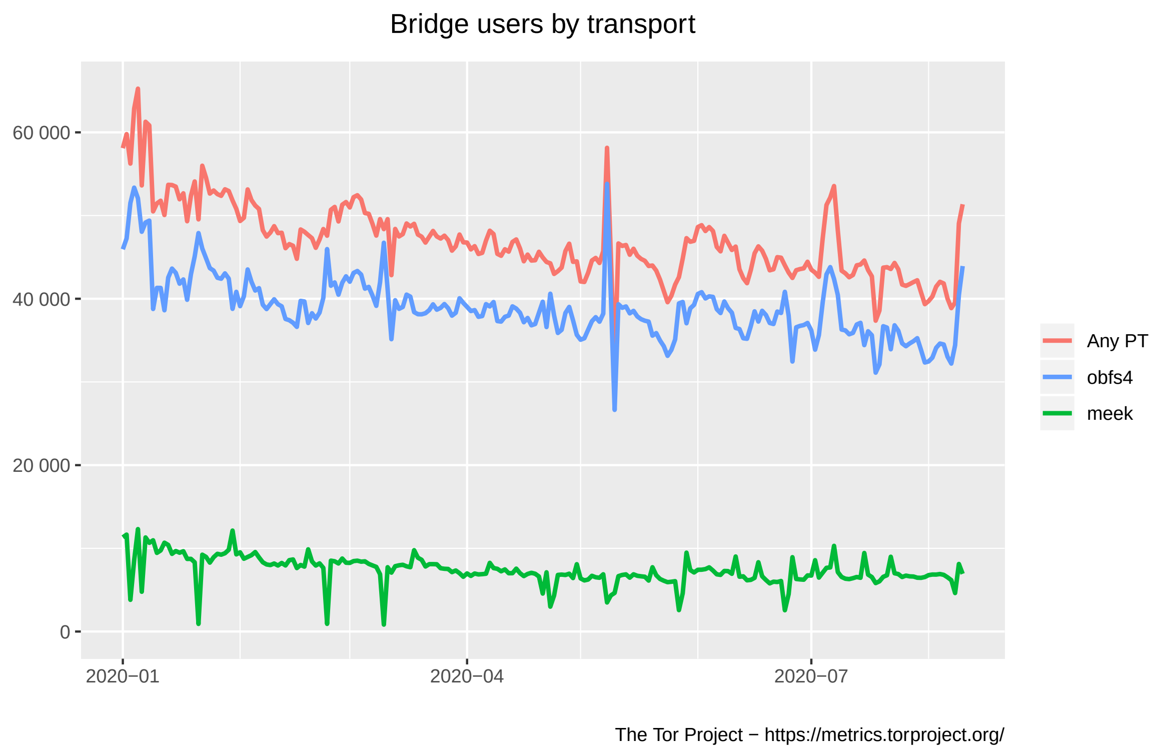


Figure 2.1: **Estimated Number of Tor Clients Connecting via Bridges Around the World** — As of 2020, obfs4 remains the most popular pluggable transport around the world.

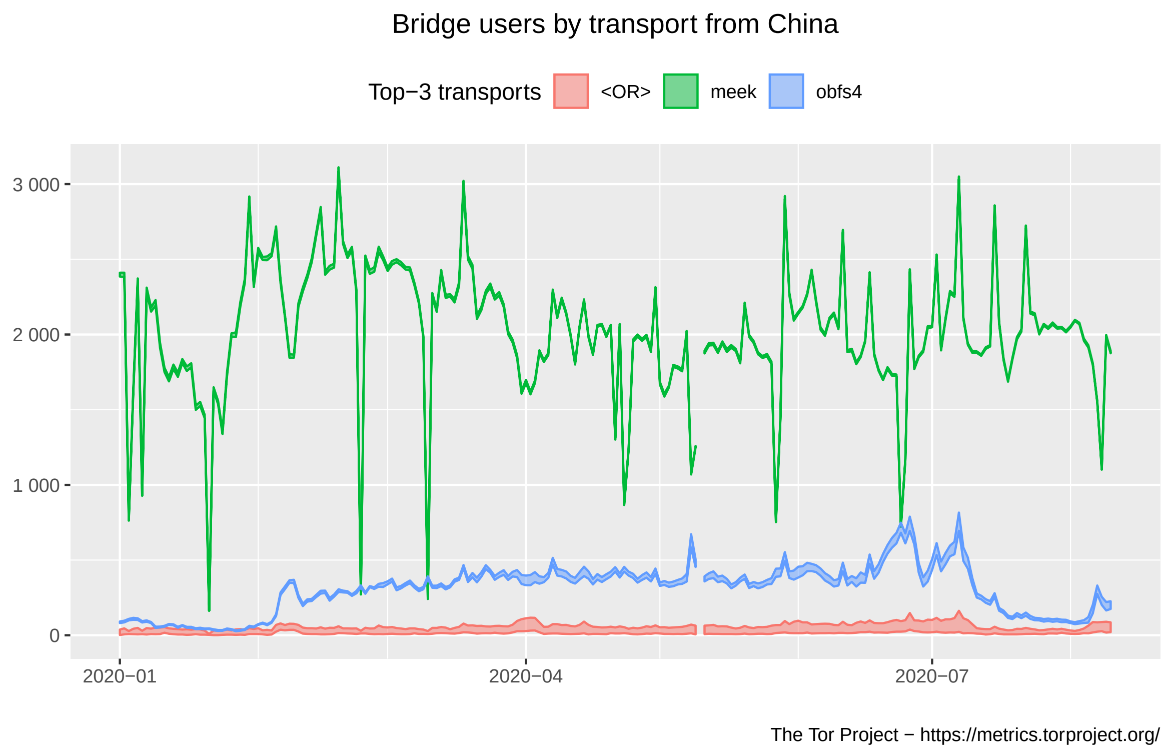


Figure 2.2: **Estimated Number of Tor Clients Connecting via Bridges in China** — meek appears to be the only reliable transport in China—a country with sophisticated censorship apparatus.

Usage of unblocked websites prevents censors from simply blocking a proxy's IP address or a DNS hostname in a targeted matter, protecting against enumeration attacks. Censors will either have to block all unblocked sites with Refraction Networking router on the path to them or find a way to fingerprint and block Refraction Networking connections only.

One of the biggest challenges for Refraction Networking is convincing network operators to deploy it. There are clear disincentives for ISPs, CDNs, and IXPs to deploy the technology due to inline blocking. Inline blocking refers to a setup where all traffic is stopped and inspected by the Refraction Networking router, allowing the router to handle circumvention traffic, and pass non-circumvention traffic through. According to the authors of TapDance [164], who had previously attempted to deploy Telex, even ISPs that are sympathetic to the Internet Freedom cause, were unwilling to deploy schemes that rely on inline blocking due to inevitable operational impact. This impact is prohibitive for the network operator since it includes reduced quality of service, caused by an extra router on the path, and the possibility of station malfunction, which would lead to a complete denial of service for the route. TapDance addresses this concern by not requiring an inline blocking component and being able to work using a passive tap, dramatically reducing deployment risks.

TapDance also introduced a new tagging mechanism. Instead of hiding the tag in the ClientHello, TapDance hides it in TLS ciphertext. The computation of plaintext, needed to get the desired ciphertext, is simple for stream ciphers, which xor plaintext with a keystream, meaning that TapDance developers simply needed to xor the desired ciphertext with the keystream. The authors also proposed a more complicated scheme to write into the ciphertext of CBC cipher suites. The corresponding plaintext is an incomplete HTTP request to ensure the unblocked website does not respond and waits for the remainder of the request. The larger tag provides more bits of security for the key exchange and allows the client to include arbitrary data or control messages in the initial request.

The deployability of TapDance, however, comes at a price, as it struggles with many security problems that are trivially solved with inline blocking: defenses against replay attacks, based on ServerHello, and straightforward blocking of active probes to reachable sites. Since the TapDance client and station communicate inside a still active connection between the client and reachable site, in the event of an unblocked website sending any data back, the connection would visibly break. This adds two limits for TapDance connections: a maximum amount of data client could send before the unblocked website fills its TCP window and responds, and a maximum amount of time the unblocked website will stay waiting for the client before sending a TCP packet with correct (as far as the unblocked website is concerned) TCP acknowledgment number. The TapDance client needs to re-establish the connection before either limit is reached, affecting the performance of the transport.

Another significant improvement of TapDance over Telex, which is not unique among such technologies, is that TapDance does not require to see both sides of communication to work, which further simplifies deployment. On the other hand, only observing the client's traffic makes (p-)replay defense more complicated. The paper suggests two remedies: recording of previous tags to prevent reuse, and clients including recent timestamps. TapDance paper also mentions that a censor could issue fake TLS certificates from a CA under its control, but argues that it is risky to do so after the roll-out of certificate transparency. However, the TapDance paper does not suggest any technique to defend against the replay attack, which remains a potential direction of future work.

There is a trade-off between deployability and observability, and Slitheen [18] approach sacrifices the former to produce an outstanding result in the latter. Slitheen achieves "perfect imitation" of unblocked websites by replacing data in so-called "leaf"(e.g., ones that never contain a request for an additional resource) content types, sent by an unblocked website, preserving original sizes and timings. The Slitheen station sends covert traffic to clients in replaced leafs, while non-leaf resources are sent as-is, consuming extra bandwidth. There are practical shortcomings to Slitheen, the most important one being that it requires inline blocking and symmetric flows, exactly like Telex, which was challenging to deploy. Another potential issue is performance: authors of Slitheen in their evaluation found that 85% of top 10000 TLS sites had less than 1 MB of downstream leaf content, while some popular websites, including Facebook, Yandex, and Netflix, had next to zero replaceable data.

Finding the right balance between deployability, detectability, and performance remains one of the main challenges for successful Refraction Networking deployment. Limited deployment, combined with overhead of Refraction Networking systems, may present a scalability issue going forward. However, those systems will likely remain a good fit for the role of emergency rendezvous channel for updates and node discovery.

In this chapter, I will describe my contributions to the development of Refraction Networking. In section 2.2, I will report the results of the first ISP-scale deployment of TapDance, which helped censored users circumvent censorship, and paved the way for the further advancement of Refraction Networking systems. In section 2.3, I will describe Conjure, a next-generation Refraction Networking system, that addresses multiple shortcomings of TapDance while maintaining its deployability advantage.

2.2 An ISP-scale deployment of TapDance

2.2.1 Introduction

Censorship circumvention tools typically operate by connecting users to a proxy server located outside the censoring country [33, 89, 119, 150]. Although existing tools use a variety of techniques to conceal the locations of their proxies [43, 54, 106, 147, 158], governments are deploying increasingly sophisticated and effective means to discover and block the proxies [46, 47, 160].

Refraction networking [123][1] is a next-generation circumvention approach with the potential to escape from this cat-and-mouse game. Rather than running proxies at specific edge-hosts and attempting to hide them from censors, refraction works via Internet service providers (ISPs) or other network operators, who provide censorship circumvention functionality for any connection that **passes through** their networks. To accomplish this, clients make HTTPS connections to sites that they can reach, where such connections traverse a participating network. The participating network operator recognizes a steganographic signal from the client and appends the user's requested data to the encrypted connection response. From the perspective of the censor, these connections are indistinguishable from normal TLS connections to sites the censor has not blocked. To block the refraction connections, the censor would need to block all connections that traverse a participating network. The more ISPs participate in such a system, the greater the extent of collateral damage that would-be censors would suffer by blocking the refracted connections.

A variety of refraction networking systems have been proposed in recent years [18, 45, 72, 82, 164, 165], representing different trade-offs among practicality, stealthiness, and performance. The basic idea is to watch all of the traffic passing through a router, selecting flows which are steganographically tagged as participating in the protocol, and then modifying that traffic by extracting and making the encapsulated request on behalf of the client. While each of these schemes has been prototyped in the lab, implementing refraction within a real ISP poses significant additional challenges. An ISP-scale deployment must be able to:

- Identify client connections on high-speed backbone links operating at 10–40 Gbps or more. This is

[1] Previous works used the term **decoy routing**, which confusingly shares the name of a specific refraction scheme. We use refraction networking as an umbrella term to refer to all schemes.

at the limits of commodity network hardware.

- Be built within reasonable cost constraints, in terms both of required hardware and of necessary rack space at crowded Internet exchange points.
- Operate reliably without disrupting the ISP's network or the reachable sites clients connect to.
- Have a mechanism for identifying reachable sites for which connections pass through the ISP, and for disseminating this information to clients.
- Coordinate traffic across multiple Internet uplinks or even multiple ISPs.

To demonstrate that these challenges can be solved, we constructed a large trial deployment of the TapDance refraction scheme [164] and operated a trial deployment in partnership with two mid-sized network operators: a regional ISP and a large university. Our goal was to understand: (**i**) the scale of traffic a refraction system built within reasonable constraints today can realistically process, (**ii**) the experience for users of refraction in contrast to traditional proxy-based circumvention, and (**iii**) the impact on ISPs of operating refraction infrastructure.

This paper presents initial results from that deployment. We discuss the design and engineering considerations that we dealt with in its construction, and we present the first data supporting the real-world practicality of refraction at ISP scale. Our client and station code is publicly available at `https://refraction.network`.

2.2.2 Deployment Overview

Over the years, several refraction schemes have been proposed. The initial generation includes Cirripede, Curveball, and Telex [72, 82, 165]. Each of these schemes employed an inline blocking element located at an ISP that could observe network flows, look for tagged connections, and selectively block flows between the censored client and the reachable server. This style of system is difficult to deploy, as it introduces a high-risk inline blocking device into an ISP's network.

We chose to implement a second generation refraction networking scheme, called TapDance [164], which was designed to be easier for ISPs to deploy than earlier proposals. TapDance operates by co-locating a "station" at each of the ISP's Internet uplinks. Unlike with previous schemes, the TapDance station does not need to alter or drop any traffic on the link; rather, it merely needs to be able to passively inspect a copy of the traffic and to inject new packets. This can be accomplished using either an optical splitter or a router port configured to mirror the link. TapDance clients send incomplete HTTPS requests to reachable sites, which are chosen so that the client's route to the site passes by the TapDance station. Clients tag the ciphertext of these connections in a way that is observable to the TapDance station, but not to a censor. Since the HTTPS request is incomplete, the reachable site will not respond. Meanwhile, the TapDance station will impersonate the server and covertly communicate with the client.

We partnered with two network operators: Merit Network, a medium-sized regional ISP and University of Colorado Boulder, a large public university. We worked with each to deploy TapDance stations in a configuration that would have visibility into most of the traffic entering and exiting their respective autonomous systems. In all, we deployed four stations, with three at Merit and one at the University of Colorado.

In order to achieve a large user base in a short time, we partnered with Psiphon [119], a widely used proxy-based circumvention tool. We implemented a TapDance client as a modular transport, and it was distributed to a fraction of Psiphon's user base during the trial, as a software update to the Android version of the Psiphon application. From users' perspective, our client was invisible; it simply provided another method of reaching circumvention service. This arrangement also helped ensure user privacy, since data was encrypted end-to-end from the clients to servers operated by Psiphon.

Our successful at-scale deployment of TapDance for censored users provides strong evidence that the technique operates well within the bandwidth, latency, and middlebox constraints that are typical of heavily censored network environments. The scale and duration of the deployment were small enough that we do not believe we came to the attention of censors, so we do not claim to have obtained evidence about TapDance's resistance to adversarial countermeasures. However, the trial did exercise our system's operational behavior and ISP logistics.

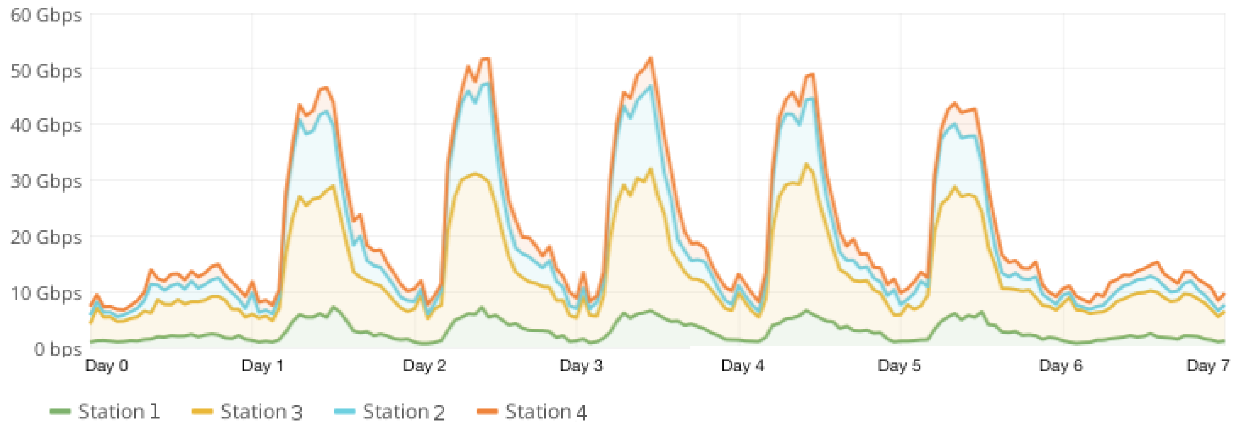


Figure 2.3: **Station Traffic** — Our four stations processed mirrored traffic from 10 and 40 Gbps ISP upstream links in order to detect connections from TapDance clients. We show traffic processed over one week, which peaked above 55 Gbps.

The trial took place during the spring of 2017. We have decided not to disclose the precise time window of the trial, in order to mitigate risks to end-users and the people supporting them.

2.2.3 Scaling TapDance

TapDance was designed to be easy for ISPs to deploy, and it proved to be neither technically difficult nor expensive for either network operator to provide the required traffic visibility. This was accomplished by configuring gateway routers to mirror traffic on a port that was attached to our station hardware. For packet injection, we simply used the stations' default network interfaces, which were not subject to any filtering that would prevent spoofing.

Bandwidth and traffic volume varied by station location, with two of the stations (both at Merit) operating on 40 Gbps links and the other two on 10 Gbps links. Figure 2.3 shows the traffic processed by each of the four stations during one week of the trial.

Space constraints at the peering locations limited each TapDance station to a single commodity 1U server. This limited the amount of CPU, RAM, and network hardware available to each station and required us to engineer for efficiency. Station 3, which handled the most traffic, was provisioned with an 8-core Intel Xeon E5-2667v3 CPU (3.20 GHz), 64 GB of RAM, and a quad-port Intel X710 10GbE SFP+ network adapter. The hardware specifications of other stations varied slightly.

2.2.3.1 Handling 40Gbps Links

We reimplemented the TapDance station from scratch, primarily in Rust. Rust is an emerging low-level systems language that provides compile-time guarantees about memory safety, which lowers the risk of remote compromise. To efficiently process packets at 40 Gbps line rates, our implementation is built on the PF_RING library and kernel driver [113], operating in zero-copy mode. By splitting incoming traffic onto multiple cores we were able to handle full line rate traffic with only occasional dropped packets, which, due to the design of TapDance, do not interfere with the normal operation of an ISP. Our experience demonstrates that, even in large installations, a software-based implementation of TapDance can be practical, avoiding the need for costly specialized hardware.

The station uses C in three main areas. First, it uses a C implementation of the Elligator [13] tagging scheme for better performance during initial detection of TapDance flows. Next, it uses OpenSSL to handle TLS work, including manually assembling TLS connections from secrets extracted from TapDance tags. Finally, it uses `forge_socket`, a Linux kernel module that allows a userspace application to create a network socket with arbitrary TCP state, in order to utilize Linux's TCP implementation while pretending to be the server.

Handling multiple TapDance users is highly parallelizable, so the station is designed to use multiple cores by running multiple processes with absolutely no communication among them. However, a single TapDance session typically spans multiple TLS flows. This means that, once a user has begun a TapDance session on a particular station process, that same process must see all future TLS flows in that session. To achieve this, we configure PF_RING to spread flows based on the IP pair, rather than the full TCP 4-tuple, so that the client's changing source port will not send the traffic to a different process.

Our four stations ran on a total of 34 cores (excluding a dedicated PF_RING core per station), with the most loaded station using 14 cores. These 34 cores were able to comfortably handle a peak of close to 14,000 new TLS connections per second, with each connection being checked for a TapDance-tagged request.

2.2.3.2 Client Integration

We implemented a from-scratch TapDance client in Go, a language that can be integrated in mobile applications and that, like Rust, guards against memory corruption vulnerabilities. Our client was integrated into Psiphon's existing circumvention application as a library presenting a new reliable transport interface—that is, the same interface that Go programs see when using TCP or TLS.

Through the trial, we logged only a minimal amount of information from users due to privacy concerns. For example, we did not log the source IP address of clients nor the destination page they requested. Instead, we logged a connection identifier generated by the client and used for a session duration, as well as the aggregate number of bytes uploaded and downloaded in the session.

2.2.3.3 Selecting Reachable Sites

Unlike previous lab-scale prototypes, our implementation required a mechanism by which clients could discover reachable, uncensored sites with our stations on-path. We achieved this by compiling a list of potential sites and shipping it with the client. We updated the list daily and pushed updates to clients the first time they connected each day.

To construct a list of candidate sites, we started by using Censys [40] to retrieve the list of all hosts serving HTTPS with browser-trusted certificates from within the AS or a customer AS of our participating network operators. This initial set was typically around 5500 sites, although there was a small amount of churn.

Next, we retrieved `robots.txt` from each site. Sites could decline to be used by our clients by adding a specific entry to this file. To advertise this, we included a URL in the client's `User-Agent` string that led to a site explaining how TapDance works and how to opt out. No websites chose to be excluded during our month-long trial.

A server's suitability also depends on its TCP window size and HTTP timeout duration. In TapDance, a client can only send up to one TCP window of data before the server will respond with stale ACK packets. Similarly, a TapDance connection can only be used until the point where the real server would respond with an HTTP timeout or otherwise attempt to close the connection.

Maximizing these parameters enables connections to last longer and transfer more data. We measured them for each site and included them in the list sent to clients. CDFs of these measurements are shown in Figures 2.4 and 2.5. Requiring a minimum 15 KB TCP window eliminated 24% of candidate sites, and a 30 second HTTP timeout eliminated 11%. Together, these minimums reduced the number of viable sites by 34%. Additionally, for sites to be suitable, they need to support the TLS ciphersuites that our client implements (the `AES_128_GCM` family). This requirement eliminated an average of 32% of candidates.

Finally, we made a test connection to each candidate using an automated version of our client that was positioned so that a station was guaranteed to observe all traffic. If this was successful, we added the host to the list of potential reachable sites sent to clients.

These tests ensure that a client's connection to a server will be usable for TapDance as long as the client's connection passes by a station. However, routing is in general difficult to predict, and so this last condition is not guaranteed to hold. We located our stations so that most traffic from international hosts in the operators' networks would be observed. Since some connections would not be visible to the stations, the clients were programmed to select sites from their lists at random, and to retry with a different site if one did not work.

During the trial, we released only half of the viable hosts to clients, in order to better test the load on individual hosts and to retain backup hosts should the censors begin to block the hosts by IP address. A daily median of 450 hosts were available, with an average overlap of 366 from the previous day.

2.2.3.4 Handling Multiple Stations

Since we had multiple stations within the same ISP, we needed to account for the possibility that a connection would switch from being observed by one station to another mid-flow, due to routing instability. What we found was that individual client-to-reachable site routes were stable and generally only passed by a single station, but that it was common for two different clients to get to the same reachable site using routes that passed by two different stations.

In our design, we decided to minimize communication between stations in order to simplify the

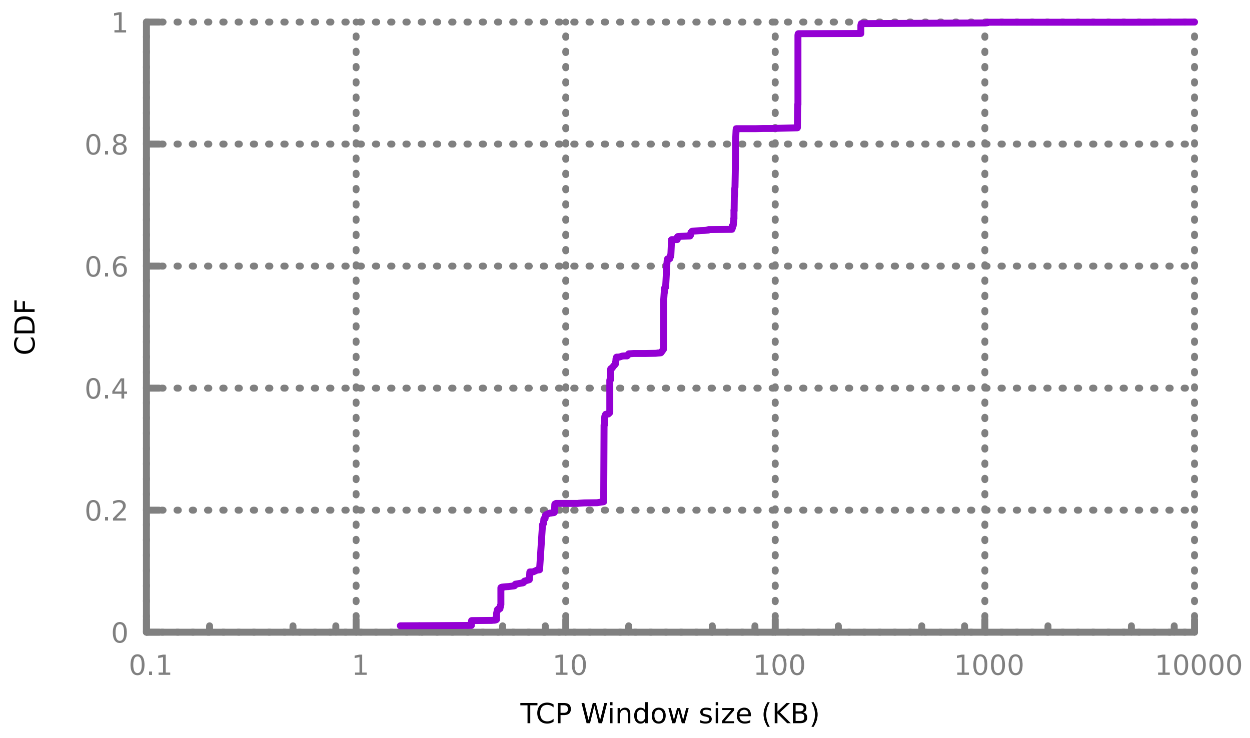


Figure 2.4: **TCP Window Size of Candidate Hosts** — We measured the TCP window size for each candidate reachable host, in order to find ones suitable for TapDance.

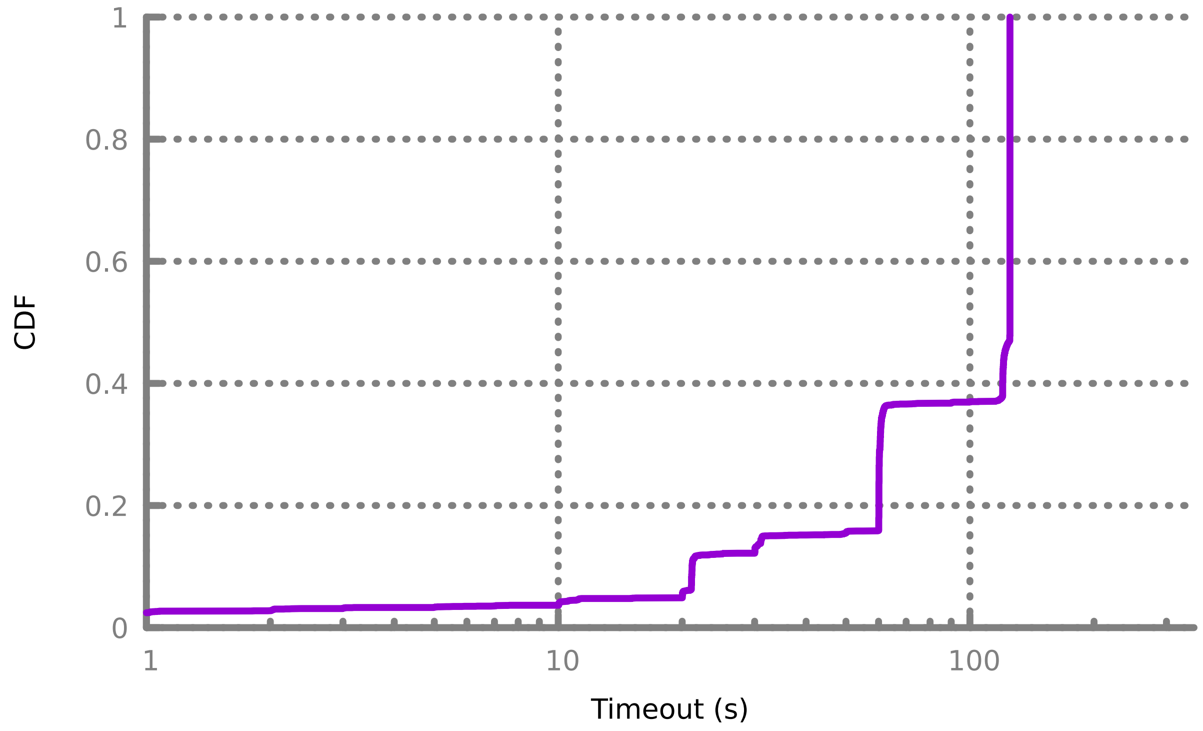


Figure 2.5: **Timeouts of Candidate Hosts** — We measured how long (up to a limit of 120 seconds) candidate reachable hosts would leave open an incomplete HTTP request.

implementation and reduce latency. However, this means that for a client to have an uninterrupted session with the proxy, it must communicate with only a single station. During the session, multiple connections must be made to reachable sites, with the client-to-proxy traffic being multiplexed over them. Therefore, to ensure that these flows all use the same station, we had clients pick a single reachable host for the lifetime of a given session.

2.2.3.5 Managing Load on Reachable Sites

In our tests with our own Nginx and Apache servers, we noticed that under default configurations, these web servers limited the number of simultaneous connections they would allow. For example, after 150 connections, a default Apache webserver will stop responding to new connections until one closes. Since a client will continue using a chosen reachable site for a long period, we were concerned that “hot spots” could develop where particular hosts received disproportionate load.

We addressed this potential issue by attempting to limit the number of concurrent users to any individual site. Because users may reach a site through multiple stations, this requires coordination between stations to track how many active clients are using a particular site.

We added support in our stations to allow a central collector to inform a station that a particular site was overloaded and it should turn away additional clients attempting to use that site. We initially set the per-site limit to 30 concurrent connections to prevent inadvertent overload, and the site load remained well below our expected threshold for the duration of the trial.

In addition to this change, we also gave clients that failed to connect to a site an increasing timeout before trying the next site. This exponential back off ensured that, if a station or other failure occurred, the load clients pushed on sites or other network infrastructure would decrease rapidly.

2.2.4 Trial Results

We present an initial analysis of the results from our trial deployment in the spring of 2017.

2.2.4.1 At-Scale Operation

A major goal of this deployment was validating the hardware and operational requirements for running TapDance in a production setting. We were able to run our system while processing 40 Gbps of ISP traffic from commodity 1U servers. The observed traffic over our measurement week is shown in Figure 2.3, showing a cumulative processing peak of 55 Gbps across stations, and a single station processing more than 20 Gbps. During our trial, the CPU load on our stations remained below 25%, although this figure alone is not representative of achievable performance, which is heavily dependent on scheduling between the processors and network card.

2.2.4.2 User Traffic

Over the trial, we served over 50,000 unique users, according to Psiphon statistics. For privacy reasons, we did not track identifying information of users during our trial. Instead, Psiphon clients reported the last successful connection time, whenever they reconnected. This allowed Psiphon to query for connections with a last connect time before the current day to get a unique daily user count, as shown in Figure 2.6. Figure 2.7 shows the number of concurrently active users over the trial week. At peak, TapDance served over 4,000 users simultaneously, with peaks on a single station over 3,000 concurrent users. Interestingly, we see two peaks per day, which is not explained by user timezone distributions.

On day four of the trial, we discovered a bottleneck that was limiting use on the most popular station (Station 2). At this station, the injection interface was misconfigured to be over a 100 Mbps management interface. This link was saturated by the amount of traffic being injected by our station, making downloads noticeably slower for users of this station. The station was at a remote facility where it was impractical to reconfigure the network, so we worked with our Merit to shift traffic headed to a large subset of reachable sites so that it passed a different station (Station 3) with a 1 Gbps injection interface. This solution improved download performance considerably, reflected in the observed usage in Figure 2.8.

Figure 2.9 shows a lower bound estimate on the bandwidth available to users. The average throughput of 5 KB/s shown in this CDF does not reflect users' actual experience, as many sessions included in the CDF

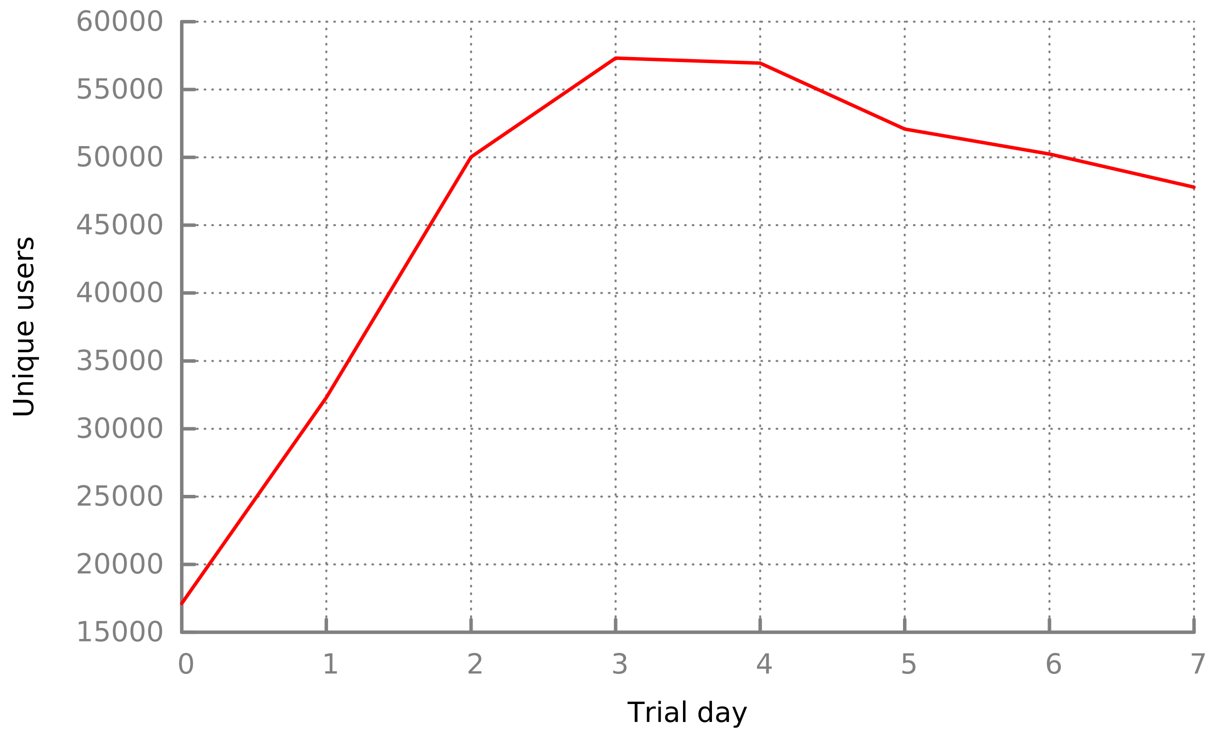


Figure 2.6: **Unique Daily Users** — The number of unique connected clients over each day of our measurement week. At peak, we handled over 57,000 unique users.

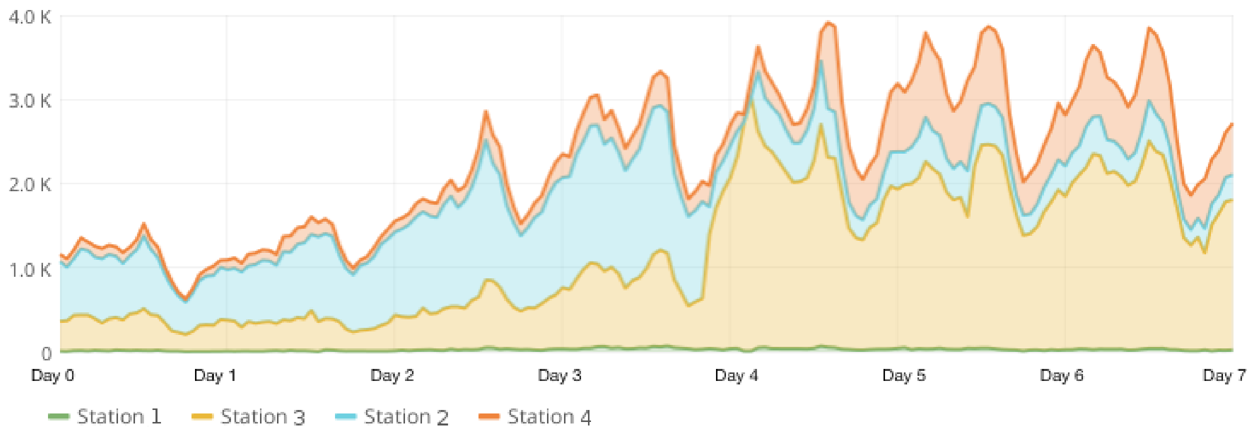


Figure 2.7: **Concurrent Sessions** — This plot shows the number of active TapDance user sessions, which peaked at 4,000.

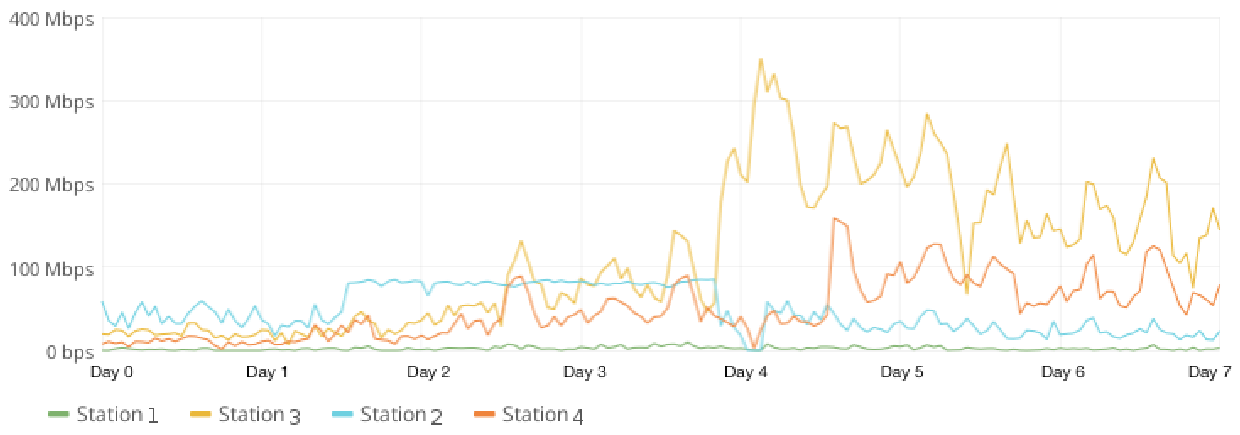


Figure 2.8: **User Traffic (Downstream)** — This plot shows how much user traffic our deployment served to clients. In response to the saturation of the 100 Mbps link on Station 2, we took steps on Day 3 to migrate user load to Station 3.

did not see real usage. The structure of our deployment was such that a users' entire device's internet traffic would be tunneled through a single, long-running TapDance session. These sessions must be occasionally restarted, due to network instability or other problems (see Figure 2.10). Therefore, many of the observed sessions happened entirely while the user was not using their device at all.

Figure 2.10 shows the distribution of client session durations. The median individual session length to a station was on the order of a few minutes. We reviewed logs to determine that about 20% of sessions ended in a timeout (where the client was not heard from for over 30 seconds).

2.2.4.3 Impact on Reachable Sites

During the trial, we measured the impact of multiple clients using the same site to reach TapDance. As described in Section 2.2.3.3, we are attentive to the number of clients simultaneously using a given site, as large numbers of clients could lead to resource exhaustion.

Figure 2.11 shows the load on the median, 90th, and 99th percentile sites over the trial. The median load remained generally evenly spread, with the typical site seeing around 5 clients connected simultaneously, with only 10% of sites ever having more than 20 simultaneous users.

Mid-week, we shifted more users toward Station 4 by increasing how often clients picked sites that were likely to be served by that station. We simultaneously increased the maximum simultaneous connections allowed to each site from 30 to 45. This is reflected in the 99th percentile site load, which peaked at 37 concurrent users to a single site.

2.2.5 Acknowledgements

The authors thank the anonymous reviewers and our shepherd Roger Dingledine for their thoughtful comments and guidance. We also wish to thank our partners, especially Merit Network, Psiphon, and the University of Colorado IT Security and Network Operations. Finally, we thank Christine Jones and Daniel Ellard of Raytheon BBN Technologies for their help in assessing and testing the system.

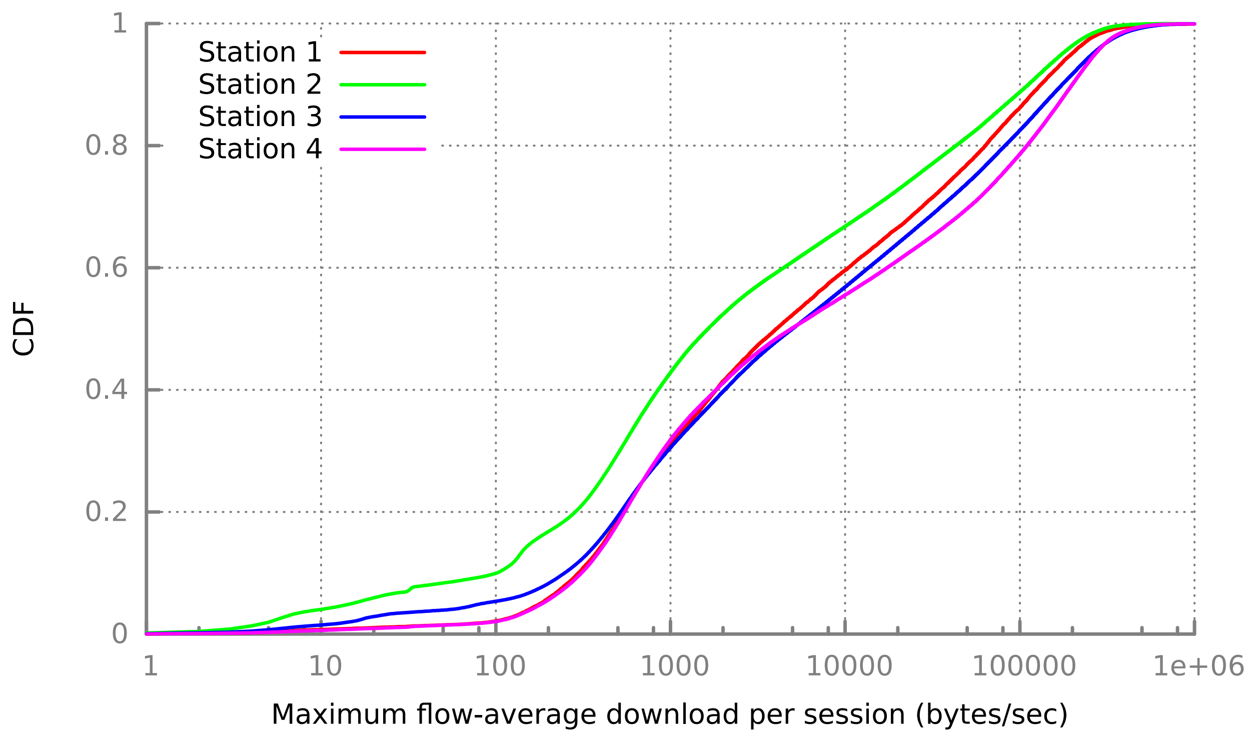


Figure 2.9: **Session Throughput** — This CDF shows the average downstream throughput achieved by the fastest sub-flow during each client session. Note that this is a lower bound for actual throughput, as it is possible that sub-flows were not actively used for their entire duration.

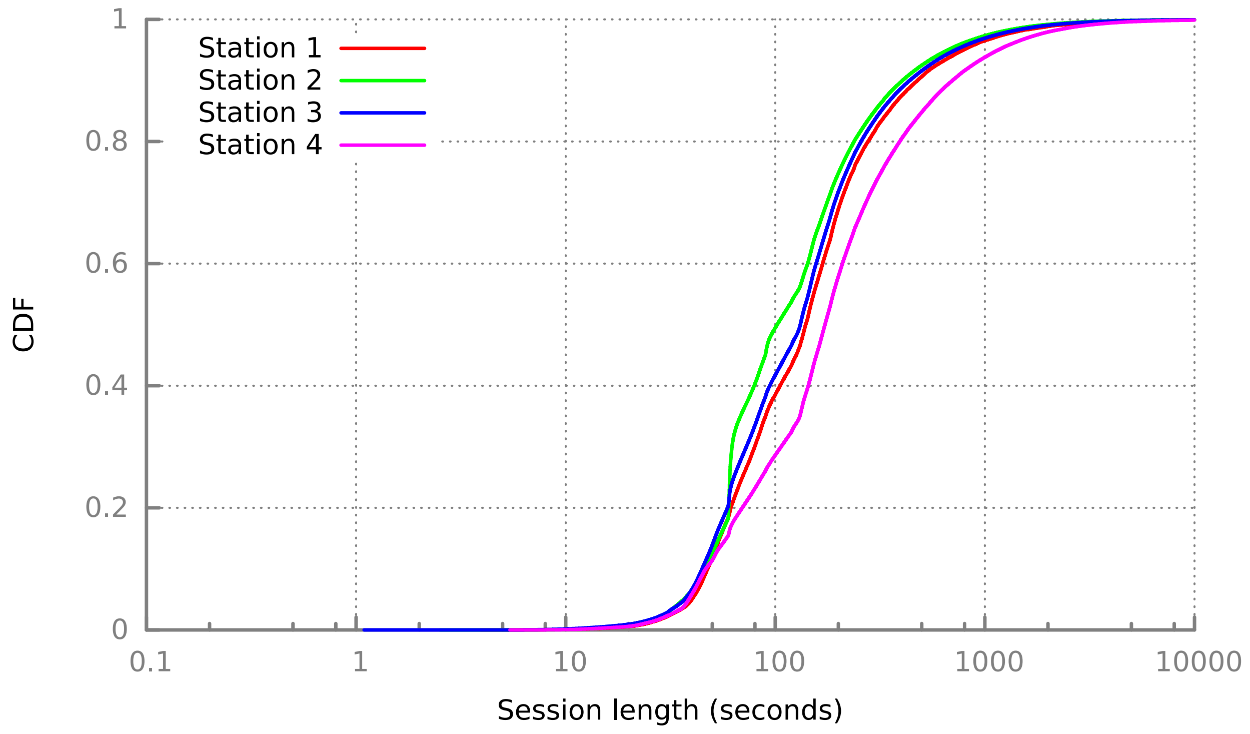


Figure 2.10: **Session Length** — This CDF shows client session durations over our trial, i.e., the time each client stayed continuously connected to a station without interruption. When sessions were interrupted (due to network failures or timeouts), the client automatically reconnected.

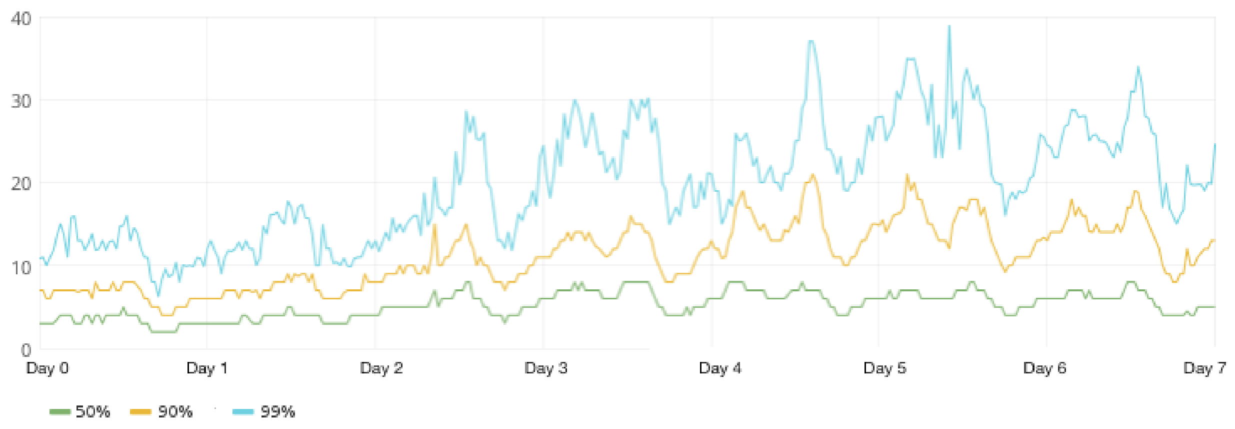


Figure 2.11: **Clients per Site** — This plot shows how many clients were connected to the average, 90th-, and 99th-percentile reachable site during our trial deployment.

2.2.6 Conclusion

Refraction networking offers a new path forward in the censorship arms race, potentially shifting the balance in favor of Internet freedom. At the same time, real-world deployment poses significant challenges. In this paper, we presented initial results from the first real-world refraction networking deployment. Our experience demonstrates that refraction can operate at ISP-scale and support tens of thousands of real users. Through this trial, we have identified and overcome several challenges related to scaling, operation, and maintenance of refraction networking. The lessons of our experience are likely to be useful for future research and deployment efforts. We hope that this first success paves the way for wider refraction networking deployments in the near future.

2.3 Conjure: Summoning Proxies from Unused Address Space

2.3.1 Introduction

Over half of Internet users globally now live in countries that block political, social, or religious content online [57]. Meanwhile, many popular tools and techniques for circumventing such censorship have ceased to function, because censors have evolved new ways to block them [47, 100] or infrastructure they rely on has become unavailable.

For example, domain fronting [54] was a popular circumvention strategy used by meek (in Tor), as well as by the Signal secure messaging app to get around censorship in countries where it was blocked [102, 145]. But in May 2018, Google and Amazon made both technical and policy changes to their cloud infrastructures that removed support for domain fronting [98]. While meek continues to use other cloud providers that (for now) continue to allow domain fronting, Signal abandoned the strategy altogether [101]. There is an urgent need for new, more robust approaches to circumvention.

A family of techniques called Refraction Networking [18, 45, 72, 82, 111, 164, 165], formerly known as Decoy Routing, has made promising steps towards that goal. These techniques operate in the network's core, at cooperating Internet Service Providers (ISPs) outside censoring countries [123]. Clients access circumvention services by connecting to a “decoy site”—any uncensored website for which the connection travels over a participating ISP. Upon recognizing a steganographic signal from the client, the ISP modifies the connection response to return censored content requested by the user. Censors cannot easily block access without also blocking legitimate connections to the decoy sites—collateral damage that may be prohibitive for censors if Refraction Networking is widely deployed [129].

However, deploying such schemes is more difficult than with most edge-based circumvention techniques, since ISPs must be convinced to operate the systems in their production networks. To date, only TapDance [164], one of six Refraction Networking proposals, has been deployed at ISP scale [59].

TapDance was designed for ease of deployment. Instead of in-line network devices required by earlier schemes, it calls for only a passive tap. This “on-the-side” approach, though much friendlier from an ISP's perspective, leads to major challenges when interposing in an ongoing client-to–decoy connection:

- Implementation is complex and error-prone, requiring kernel patches or a custom TCP stack.
- To avoid detection, the system must carefully mimic subtle features of each decoy's TCP and TLS behavior.
- The architecture cannot resist active probing attacks, where the censor sends specially crafted packets to determine whether a suspected connection is using TapDance.
- Interactions with the decoy's network stack limit the length and duration of each connection, forcing TapDance to multiplex long-lived proxy connections over many shorter decoy connections. This adds overhead and creates a traffic pattern that is challenging to conceal.

In this paper, we present **Conjure**, a Refraction Networking protocol that overcomes these challenges while retaining TapDance's ISP-friendly deployment requirements[2] . Our key innovation is an architecture that avoids having to actively participate in client-to-decoy connections.

In our scheme (Figure 2.12), clients register their intentions to connect to phantom hosts in the "dark" or unused address space of the deploying ISP. Once registered, clients can connect to these phantom hosts IP addresses as if they were real proxy servers. The Conjure station (deployed at an ISP) acts as the other end of these connections, and responds as if it were a legitimate site or service. To the censor, these phantom hosts appear as legitimate sites or services, and even active probes will not reveal information that would allow the censor to block them.

Phantom hosts are cheap to connect to, and greatly expand the number of viable proxy endpoints that a censor must consider. This increases the cost for censors to block, as they must detect and block in real time. Meanwhile, even a censor that could theoretically detect 90% of phantom hosts with confidence does not significantly reduce the effectiveness of a circumvention system, giving Conjure an advantage in the censor/circumventor cat-and-mouse game.

Conjure supports both IPv4 and IPv6, though we note that the technique is especially powerful in IPv6, where censors cannot exhaustively scan the address space ahead of time to identify addresses that change

[2] We have released the open source implementation of the Conjure client at `https://github.com/refraction-networking/gotapdance/tree/dark-decoy`.

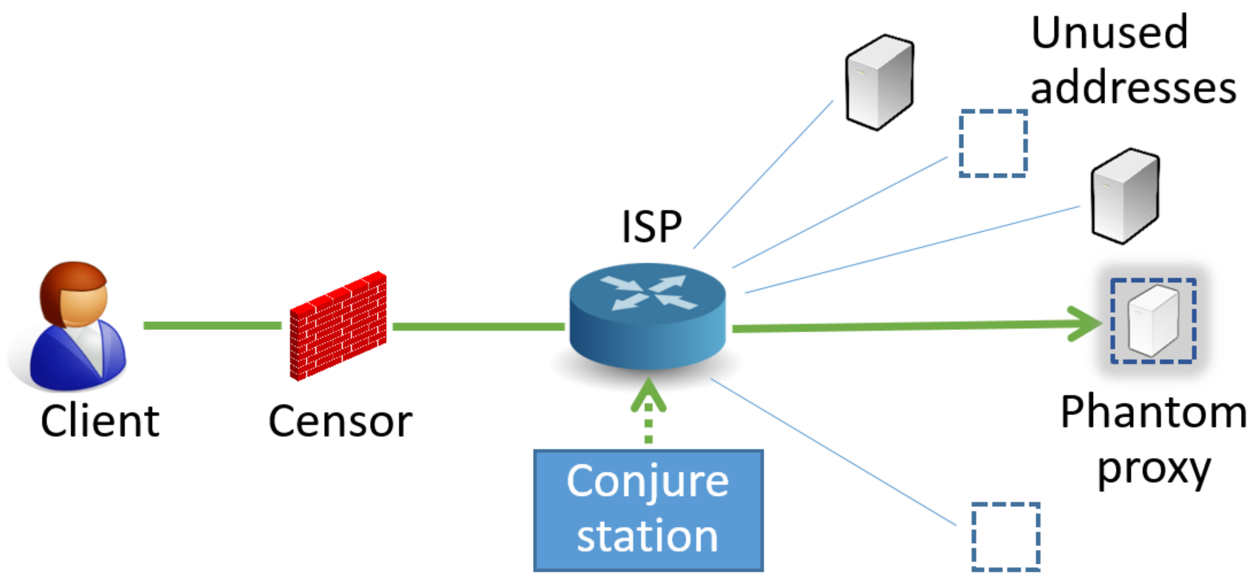


Figure 2.12: **Conjure Overview** — An ISP deploys a Conjure station, which sees a passive tap of passing traffic. Following a steganographic registration process, a client can connect to an unused IP address in the ISP's AS, and the station will inject packets to communicate with the client as if there were a proxy server at that address.

behavior. Because we fully control the proxy transport, connections can live as long as needed, without the complexity faced by TapDance.

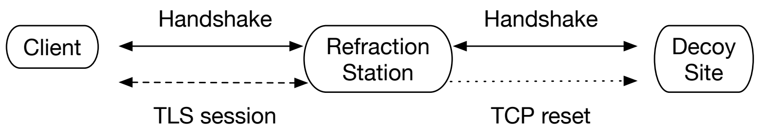
We introduce the Conjure protocol (Section 2.3.4) and analyze its security, finding that it resists a broader range of detection attacks than TapDance. We have implemented Conjure (Section 2.3.5) and deployed it on a 20 Gbps ISP testbed similar to the TapDance deployment [59]. Compared to TapDance, we find that Conjure has reduced complexity and substantially improved performance (Section 2.3.6): on average, Conjure has 20% lower latency, 14% faster download bandwidth, and over 1400 times faster upload bandwidth. In addition, Conjure provides significantly more flexibility than existing Refraction Networking protocols, allowing maintainers to respond to future censor techniques. We believe that these advantages make Conjure a strong choice for future Refraction Networking deployments.

2.3.2 Background

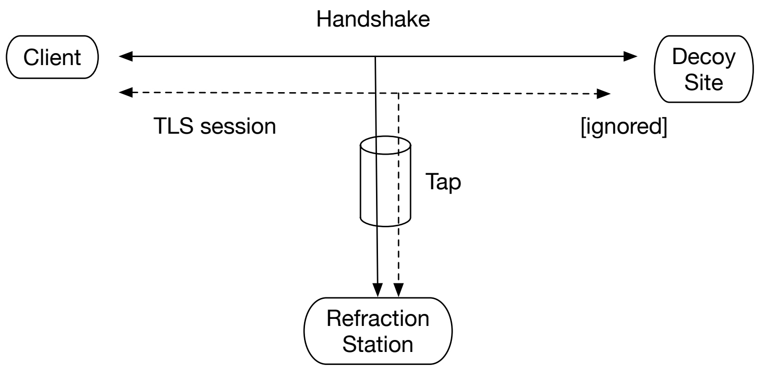
Refraction Networking operates by injecting covert communication inside a client's HTTPS connection with a reachable site, also known as a decoy site. In a regular HTTPS session, a client establishes a TCP connection, performs a TLS handshake with a destination site, sends an encrypted web request, and receives an encrypted response. In Refraction Networking, at least one direction of this exchange is observed by a **refraction station**, deployed at some Internet service provider (ISP). The station watches for a covert signal from the client that this connection is to be used for censorship circumvention. Upon seeing the signal, the station will take over the HTTPS session, and establish a proxy session with the client that can then be used for covert communication.

One of the key challenges for Refraction Networking is in taking over a session. The station must start responding to the client's traffic as if it were the decoy destination, and at the same time prevent the destination from sending its own responses back to the client. A simple approach is to have the refraction station act as an inline transparent proxy (Figure 2.13a) that forwards the traffic between the client and the decoy site. After a TLS handshake has been completed, the station terminates the connection with the decoy site by sending a TCP reset and takes over the session with the client.

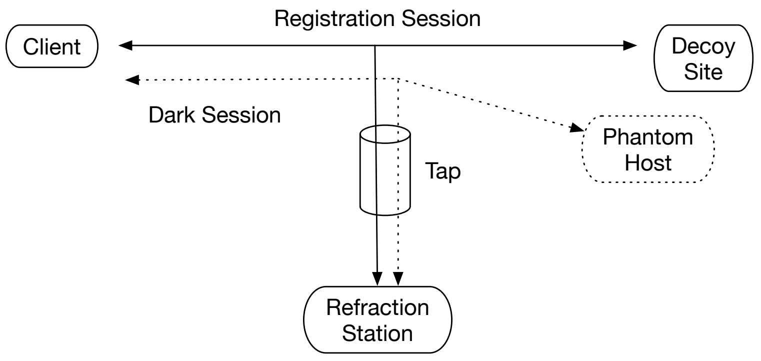
An inline element, however, can significantly affect the reliability and performance of the regular,



(a) **First Generation Systems** for Refraction Networking, such as Telex and Cirripede, operated as inline network elements, with the ability to observe traffic and block specific flows. ISPs worried that if the inline element failed, it could bring down the network.



(b) **TapDance** is a second-generation Refraction Network scheme that operates without flow blocking, needing only to passively observe traffic and inject packets. TapDance has recently been deployed at a mid-size ISP, but the techniques used to silence the decoy site and participate in the client–decoy TCP connection mid-stream add significant complexity, performance bottlenecks, and detection risk.



(c) **Conjure**, our third-generation Refraction Networking design, overcomes these limitations. It uses two sessions. First, the client connects to a decoy site and embeds a steganographic registration message, which the station receives using only a passive tap. Second, the client connects to a “phantom host” where there is no running server, and the station proxies the connection in its entirety.

Figure 2.13: **Evolution of Refraction Networking**

non-refraction traffic of an ISP. Cirripede [72] and Telex [165] attempted to mitigate this by dynamically adding router rules to forward only a subset of traffic from a registered client or an active session through the element, but this nevertheless presented a deployability challenge.

TapDance [164] offered an alternative design that did not require the blocking or redirection of traffic, but used a mirror port instead (Figure 2.13b). In TapDance a client sends an incomplete HTTP request, which causes the decoy site to pause waiting for more data while the station takes over the connection in its place. After a client would receive a packet initiated by the station, its TCP sequence numbers would become desynchronized with the decoy site, causing the decoy to ignore the packets sent by the client.

This approach reduced the barriers to deployment and TapDance was used in production during a pilot study, serving upwards of 50,000 real-world users [59]. The tap-based approach, however, has some disadvantages. A decoy site will only ignore packets as long as the sequence numbers stay within its TCP window, and will terminate the connection after a timeout. Frolov et al. report that in their pilot, they eliminated roughly a third of potential decoy sites due to their measured window or timeout values being too small [59]. Even so, sessions that try to upload non-trivial data amounts (in excess of about 15 KB) or last longer than the timeout value (ranging from 20–120 s) require the user to create new refraction connections, adding overhead, complexity, and opportunities for errors. Additionally, keeping the connections to the decoy site open for tens of seconds uses up the site's resources; Frolov et al. found that a default configuration of the Apache web server would only keep 150 simultaneous connections open, while the pilot deployment would often result in dozens of connections to the same decoy site, creating a scaling concern.

Conjure is able to avoid these problems by creating proxies at unused IP addresses, allowing the station full control over a host it has created, rather than forcing it to mimic an already existing decoy (Figure 2.13c). This design obviates the need for taking over a session already in progress, which both simplifies the implementation and eliminates certain attacks, as we will discuss in Section 2.3.7.

Registration Signal In all implementations of Refraction Networking, a client must send a covert signal to the station to initiate communication. This covert signal is embedded inside communication fields that must be indistinguishable from random by a censor without access to a secret/private key available to the station. Past implementations have used TCP initial sequence numbers [72], the ClientRandom field inside a

TLS handshake [82, 165], and the encrypted body of an HTTPS request [164]. In principle Conjure can use any of these mechanisms for registration, but in our prototype we used the HTTPS request body as it offers the greatest flexibility for the amount of data that can be sent with the registration.

2.3.3 Threat Model

Our deployment model is identical to that of TapDance: we only require a passive tap at the deploying ISP, and the ability to inject (spoofed) traffic from phantom hosts. Furthermore, we assume asymmetric routing (i.e. that the tap might only see packets from but not to the client). However, we assume a stronger threat model for the adversary than TapDance, as our design resists active attacks.

We assume the censor can block arbitrary IP addresses and networks, but faces a cost in doing so if it blocks potentially useful resources. In particular, we assume it is difficult for the censor to have complete knowledge of legitimate addresses used, and so instead resorts to a blacklist approach to blocking proxies and objectionable content. Whitelists are expensive for censors to maintain and can stifle innovation, and are rarely employed by country-level censors.

We assume that the censor can know what network the Conjure station(s) are deployed in and the prefixes phantom hosts are selected from, but that blocking those networks outright brings a collateral damage the censor is unwilling to suffer. Instead, the censor aims to identify the addresses that are phantom hosts, and block only those. We note this assumption supposes that the censor does not mount effective routing decoy attacks [73, 134]; we discuss these attacks further in Section 2.3.8.2.

We allow the censor access to the client to register and use its own phantom hosts, so the system should ensure that these will not reveal the phantom hosts of other users. The censor can also actively probe addresses that it sees users accessing, and can employ tools such as ZMap [42] to scan large network blocks, excepting large IPv6 prefixes (e.g. a /32 IPv6 prefix contains 2^{96} addresses).

Finally, we assume the censor can replay or preplay any connections that it suspects involve phantom hosts (or their registration) in an attempt to confirm. However, the censor wishes to avoid disrupting any connections before it knows for certain they are from Conjure clients, lest they disrupt legitimate connections. This means that injecting false data or corrupting TLS sessions is outside the scope of the censor, but that the censor can send non-disruptive probes (such as stale TCP acknowledgements) that would normally be ignored. We emphasize that TapDance is observable by censors that can send TCP packets in suspected connections, but that our protocol is robust against this class of censor.

2.3.4 Architecture

Conjure involves two high-level steps. First, clients **register** with a Conjure station deployed at an ISP, and derive a phantom host IP address from a seed shared in the registration. Then, clients **connect** to the agreed upon phantom address, and the station listening on the tap relays the client's traffic to a local application. The application brokers traffic to a proxy application specified by the client in their registration providing a probe resistant tunnel. Figure 2.14 describes a high-level overview of the Conjure registration and connection behavior.

Similar to TapDance [164], our design does not require expensive in-line flow blocking and is accomplished with only a passive tap at the ISP, imparting little to no burden on their infrastructure and service reliability. Our architecture is also modular, in that the registration and connection steps operate independently, allowing a wide range of flexibility to evade censors. We describe each of these components, and then describe our implementation and deployment in Section 2.3.5.

2.3.4.1 Registration

We use a form of Refraction Networking similar to TapDance to register directly with the station, though Conjure registration is significantly simpler and more difficult to detect than vanilla TapDance. This is because registration flows are unidirectional; a client communicates their intent to register to the station without any response from the station itself. This makes registration flows very difficult to detect as they can be sent to any site hosted behind the ISP, and display no evidence of proxy behavior.

While this model of registration gives the client no definitive indication that their registration was successful, the client can attempt to register multiple times concurrently with the same information, and expect that one gets through. Alternatively clients can register intent to use the proxy in other covert ways. For instance, they could use email [74] or any other existing intermittently available proxies to register.

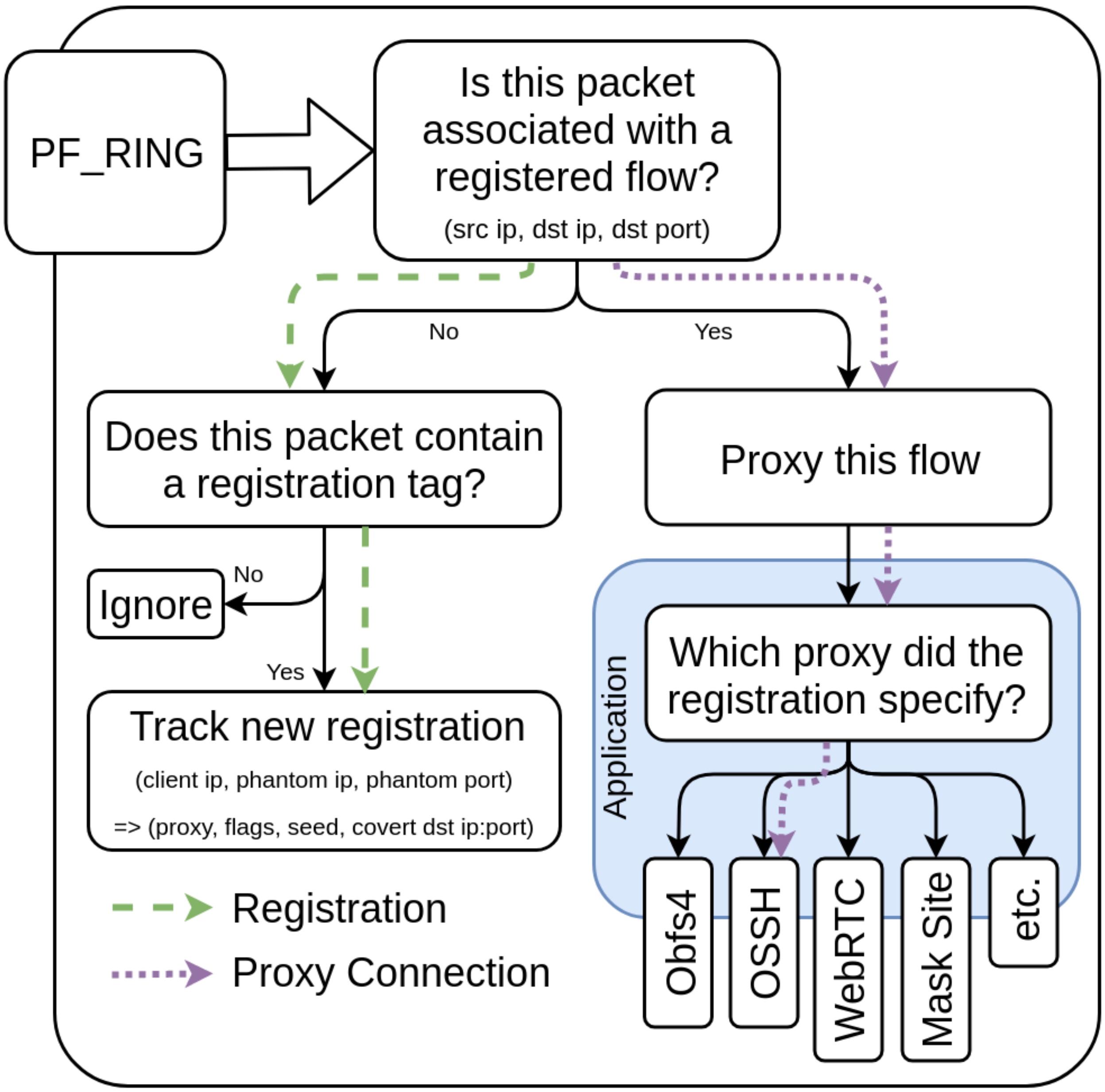

PF_RING
Is this packet associated with a registered flow?
(src ip, dst ip, dst port)
No
Yes
Does this packet contain a registration tag?
Proxy this flow
No
Ignore
Yes
Track new registration
(client ip, phantom ip, phantom port)
=> (proxy, flags, seed, covert dst ip:port)
Application
Which proxy did the registration specify?
Obfs4
OSSH
WebRTC
Mask Site
etc.
Registration
Proxy Connection

Figure 2.14: **Conjure Operations** — A Conjure session is constructed in two pieces. First a client registers by sending a TLS connection to the decoy host (see Figure 2.13c) with a steganographically tagged payload. This registration contains the true "covert" address that the client would like to connect to and a seed which the Conjure station uses to derive the Phantom IP address for the session (green dashed flow). Second, the client derives the same Phantom IP address and connects directly to it using the proxy protocol (e.g. OSSH) specified in the registration (purple dotted flow). The station identifies registered packets by source IP, destination IP, and destination port (client IP, phantom IP and port respectively). The traffic is passed to a proxy server handling the specific proxy protocol, which ultimately proxies traffic to the covert address. Secrets for the individual application proxy service are shared out of band or derived from the secret seed shared between client and station during registration, preventing an adversary from being able to successfully connect to the application in follow up probes.

A registration connection is sent to a **registration decoy**: any site that supports TLS behind the ISP relative to the client. The client completes the handshake, and sends a normal HTTPS request that embeds a ciphertext tag in the payload. Conjure leverages the same steganographic technique as TapDance to encode the ciphertext [164, §3], however we send a complete request allowing the registration decoy to respond or close the connection. The tag is encrypted such that it is only visible to the station. The Conjure station passively observes tagged flows obviating the need to mimic the decoy. In addition, more potential decoys can be used in comparison to TapDance, as there is no need to exclude decoys that have timeouts or TCP windows unfavorable for keeping connections open. In our deployment, this results in a modest increase of 25% more decoys that could be used than in TapDance.

The tag contains a public key (encoded to be indistinguishable from random using Elligator [13]), and a message encrypted under the shared secret derived from a Diffie–Hellman key exchange with the station's long-term public key hard-coded in the client. The station uses its private key to compute the same shared secret from the (decoded) client public key, and decrypts the message in the tag. The censor, without knowledge of either the station or client's private key, cannot derive the shared secret, preventing it from being able to decrypt the message, or even learn of its existence.

Inside the message, the client communicates a random seed, the covert address they would like to connect to, the transport protocol they will use, and other configuration-specific information, such as flags to signify version, and feature support. The client and server hash the seed to determine which specific IP address from the set of shared CIDR prefixes will be used and registered as a phantom host. It may seem intuitive to instead have the client send the specific IP address to register, but allowing the client arbitrary choice also allows the censor to register suspected phantom hosts and block them if they can be used as proxies. By using a hash of a seed, the censor would have to pre-image the hash to obtain a seed it could use to register for a desired IP address. We discuss the intricacies of phantom IP address selection in Section 2.3.6.2.

Once the phantom host IP address has been selected, the station watches for packets with the source address of the original client and the phantom as the destination, and forwards them on to the a local application handling transports. The station only forwards packets that originate from the IP address of the registering client, making the phantom host appear firewalled off to everyone but the client. We note that censors have been observed taking over client IP addresses for follow-up probing [47]. This would allow censors to hijack registrations if they can connect within the small window between client registration and connection. However this only allows the censor to communicate to the local application that handles transports, it does not connect them to the covert address that the client indicated in their registration. Filtering by client IP and phantom IP also prevents censors from enumerating the address space before hand, as they would have to do so from every potential client IP address. Simply scanning the prefixes with ZMap [42] from a single vantage point would not reveal hosts that only respond to specific IPs (e.g., firewalled subnets).

2.3.4.2 Transports

Once the client has registered, packets sent to the phantom host IP address are detected at the station and passed to the local **application** which provides proxy access to the client. A viable Conjure transport has two main requirements: first, the protocol it uses with the client must be difficult for the censor to **passively detect** and block by traffic inspection. Second, the endpoint must resist **active probes** by the censor (who does not know some shared secret).

Any protocol that satisfies these criteria can be used as an effective transport with Conjure. In this section, we describe various existing protocols (OSSH and obfs4) as well as introduce our own (Mask sites, TLS 1.3 ESNI, and phantom WebRTC clients) that can be used in Conjure, and evaluate how each meet the necessary requirements. Table 2.1 compares the application protocols. Conjure uses a modular approach to transports because research into proxy detection is ongoing. Having a variety of supported transports gives clients a quick way to pivot and maintain proxy access even when new proxy protocol vulnerabilities are discovered.

Obfuscated SSH

Obfuscated SSH [91] (OSSH) is a protocol that attempts to mask the Secure Shell (SSH) protocol in a thin layer of encryption. This makes it difficult for censors to identify using basic packet filters, as there are no identifying headers or fields to search for. Instead, Obfuscated SSH clients first send a 16-byte random seed, which is used to derive a symmetric key that encrypts the rest of the communication. Early versions of OSSH were passively detectable by censors, who could observe the random seed and derive the key, allowing

	OSSH [91]	obfs4 [147]	Mask Sites	TLS eSNI	WebRTC
Active probe resistant	●	●	●	●	●
Randomized or Tunneling	R	R	T	T	T
Known passive attack	[51]	[153]	-	-	-
Conjure implementation	●	○	●	○	○

Table 2.1: **Conjure Applications** — "Active probe resistant" protocols are designed to look innocuous even if scanned by a censor. "Tunneling" (T) protocols use another protocol (e.g. TLS) to blend in, while "Randomized" (R) ones attempt to have no discernable protocol fingerprint or headers. For existing protocols, we list any known attacks suggested in the literature that let censors passively detect them. We also list if we have implemented the application in our prototype.

them to de-obfuscate the protocol. These versions also did not protect against active probing attacks, as a censor could easily create their own connections to confirm if a server supports the protocol.

More recent versions of OSSH, such as those used by Psiphon [119], mix a secret value into the key derivation, thwarting the naive passive detection/decryption attack. The secret is distributed out-of-band along with the proxy's address, and is unknown to a passive or active-probing censor. If a client connects and cannot demonstrate knowledge of the secret, the OSSH server does not respond, making it more difficult for censors to discover OSSH servers via active probing attacks.

obfs4

obfs4 [147] is a pluggable transport used by Tor [33] designed to resist both passive detection and active probing. Traffic is obfuscated by encrypting it and sending headerless ciphertext messages. Similar to OSSH, clients can only connect to obfs4 servers by proving knowledge of a secret. Probing censors that do not have the secret get no response from obfs4 servers, making it difficult for censors to confirm if a host is a proxy. Server IPs and their corresponding secrets are normally distributed out-of-band through Tor's bridge distribution system.

During registration, the Conjure client and station could use the registration seed to derive the obfs4 secrets (NODE_ID and server private/public keys) needed for the client to connect. The station could then launch an obfs4 server instance locally for the client to connect to as a transport using the derived secrets. If a censor attempts to connect to the phantom address (even using the client's IP), it will not receive a response, as it does not know the registration seed used to derive the obfs4 secrets.

Using obfs4 as a Conjure application has the added benefit that servers and secrets do not need to be distributed out-of-band, eliminating one of the main ways censors currently block existing obfs4 instances [30]. Instead, each Conjure obfs4 instance is private to its registering client, and there is no public service that censors can use to discover them.

TLS

TLS is a natural protocol for Conjure applications, because it is ubiquitous on the Internet (making it difficult for censors to block), while also providing strong cryptographic protection against passive and active network adversaries. However, there are several challenges to make it robust against censors that wish to block a particular service.

One challenge is that TLS sends important server-identifying content in plaintext during the TLS handshake. This includes the Server Name Indication (SNI) in the ClientHello message that specifies the domain name, and the X.509 Certificate of the server.

To evade censors, we must send a plausible SNI value (sending no SNI is uncommon and easily blocked—only 1% of TLS connections do not send the SNI extension [62]), and we must have the server respond with a plausible (and corresponding) certificate. Even if we manage to avoid sending either in the clear (e.g. using session resumption), censors could actively probe the server in a way that would normally elicit a certificate.

Encrypted SNI

TLS 1.3 [124] offers several features that may greatly simplify Conjure transport design. For instance, TLS 1.3 handshakes include encrypted certificates, removing a strong traffic classification feature. Unfortunately, TLS 1.3 currently still sends the SNI in the (plaintext) Client Hello, meaning we would have to choose a realistic domain to fool a censor.

However, there are proposals to encrypt the SNI in the ClientHello [126], though none have been implemented or deployed as of 2019. Nonetheless, if widely adopted, Encrypted SNI (ESNI) would offer a powerful solution for Conjure applications by allowing the client to use plain TLS as the transport while remaining hidden from the censor. Censors could still try to actively probe with guesses for the SNI, but servers could respond with generic “Unknown SNI” errors. If such responses were common for incorrect SNI, the censor’s efforts to identify phantom hosts would be frustrated.

Mask Sites

Another option to overcome active and passive probing attacks is to mimic existing TLS sites. In this application, we simply forward traffic between any connecting clients and a real TLS site. To a censor, our phantom site will be difficult to distinguish from the actual “mask” site, making it expensive for them to block without potentially blocking the real site. TLS connections to the Conjure station will terminate exactly as connections to the mask site would, with Conjure acting as a transparent TCP-layer proxy between the client and mask site. However, this leaves the application unable to introspect on the contents of the TLS connection to the mask site, as it does not have the client-side shared secrets, and it cannot overtly man-in-the-middle the connection before knowing it is communicating with the legitimate client (and not the censor).

To covertly signal to the relaying application, the client changes the shared secret it derives with the mask site to something that the Conjure station can also derive. The client’s first `Application Data` packet is thus encrypted under a different secret than the client/mask site secret. Specifically, the client uses the **seed** sent during registration to derive the pre-master secret for the connection. The new pre-master secret is hashed along with the client and server randoms of the current (mask site) TLS connection to obtain the master secret that determines encryption/decryption/authentication keys.

The Conjure station can determine if the client did this by trial decryption with the master secret derived from the known seed shared at registration. If it succeeds, the client has proved knowledge of the seed, and the application can respond as a proxy. If not, the application simply continues to forward data between the client and the mask site, in case a client’s IP was taken over by a censor after registration. As the censor does not have knowledge of the **seed** used in registration, it cannot coerce the application to appear as anything besides the mask site.

Mask Site Selection

Selecting which sites to masquerade as must be done carefully to avoid censors being able to detect obvious choices. For example, if a small university network has a phantom host in their network that appears to be `apple.com`, it would be easy for a censor to block as a likely non-legitimate host. Likewise, if a phantom host at an IP address pretends to be a domain that globally resolves to a single (different) IP address, the censor could also trivially identify and block the phantom host.

Nearby sites — pick websites that are legitimately hosted in or near the network of the phantom host addresses effectively creating copies of legitimate sites. However, other signals such as DNS may reveal the true mask site.

Popular sites — choose mask sites from a list such as the Alexa top site [2] list. Although it may be wise to avoid sites that are obviously not hosted in the phantom host address range, such as large companies that run their own data centers and own their own ASN. The list could also be filtered to domains that resolve to different IP addresses from different vantage points, making it harder for a censor to know if a phantom host corresponds to a domain's IP.

Passive observation — collect sites by passively observing DNS requests, TLS SNI, or certificates that pass by at the network tap. This would allow for building a realistic set of sites that are plausibly in the vicinity of the phantom host addresses that pass by the tap.

In practice, clients can often try multiple phantom hosts/mask sites over several attempts, as blocking the client outright may negatively impact other unrelated users behind the same network (e.g. in the case of NAT). Thus, even a censor that can block most but not all mask site usage (i.e. by employing website fingerprinting) only delays access, and doesn't prevent it outright.

Phantom WebRTC Clients

Phantom hosts could also pretend to be clients instead of servers. This may potentially give censors less to block on, as actively probing clients commonly returns few or no open ports. A censor may also be hesitant to block client-to-client communication, as it could block peer-to-peer applications as well as many video conferencing protocols. WebRTC is a natural choice for a client-to-client transport in censorship circumvention, and is already used in existing schemes like Snowflake [36]. Conjure could also use WebRTC as the transport protocol, convincing the censor that two clients are communicating.

2.3.5 Implementation

We implemented Conjure and deployed a station at a mid-sized transit ISP tapping traffic from a 20 Gbps router. We used PF_RING to consume the 20 Gbps link, and feed it to a custom **detector** written in Rust. The detector processes all packets and watches for new registrations. Once a registration is detected the local application is notified via an out-of-band ZMQ [167] connection, which provides the registering client's IP address, the seed, and other configuration information. We note that this is not along a critical timing path for proxying connections and no client packets are sent over ZMQ.

The detectors forwards all packets destined for a (registered) phantom host address to the local **application** via tun interfaces and `iptables` DNAT rules that rewrite the destination IP, allowing the local application to accept and respond to connections using the native operating system's interface. Figure 2.15 shows the overall architecture of our implementation, which we describe next.

2.3.5.1 Detector

We implemented our detector in approximately 1,800 lines of Rust (compared to over 5,000 lines for TapDance)[3] .

To achieve performance needed to operate at 20 Gbps, we used PF_RING [113] to load balance incoming packets across 4 CPU cores, which each run a dedicated detector process. PF_RING supports load balancing packets based on either their flow (5-tuple), or their source/destination IPs, which allows connections to be processed by a single process without requiring communication across independent cores.

However, in Conjure, registration connections and phantom proxy connections could end up being load balanced to different cores. In order for an individual detector process to forward phantom proxy connections to the application, it must know about the original registration, even if that registration was observed by a different core. To address this, we used Redis [86] to allow each core's process to broadcast (publish) newly registered decoys to the other cores so they can add them to their local list of registered phantom host addresses. Broadcasting registrations across all cores ensures that each detector process sees all registrations, and can forward phantom proxy connections accordingly.

[3] excluding (from both) about 3,000 lines of auto-generated protobuf code

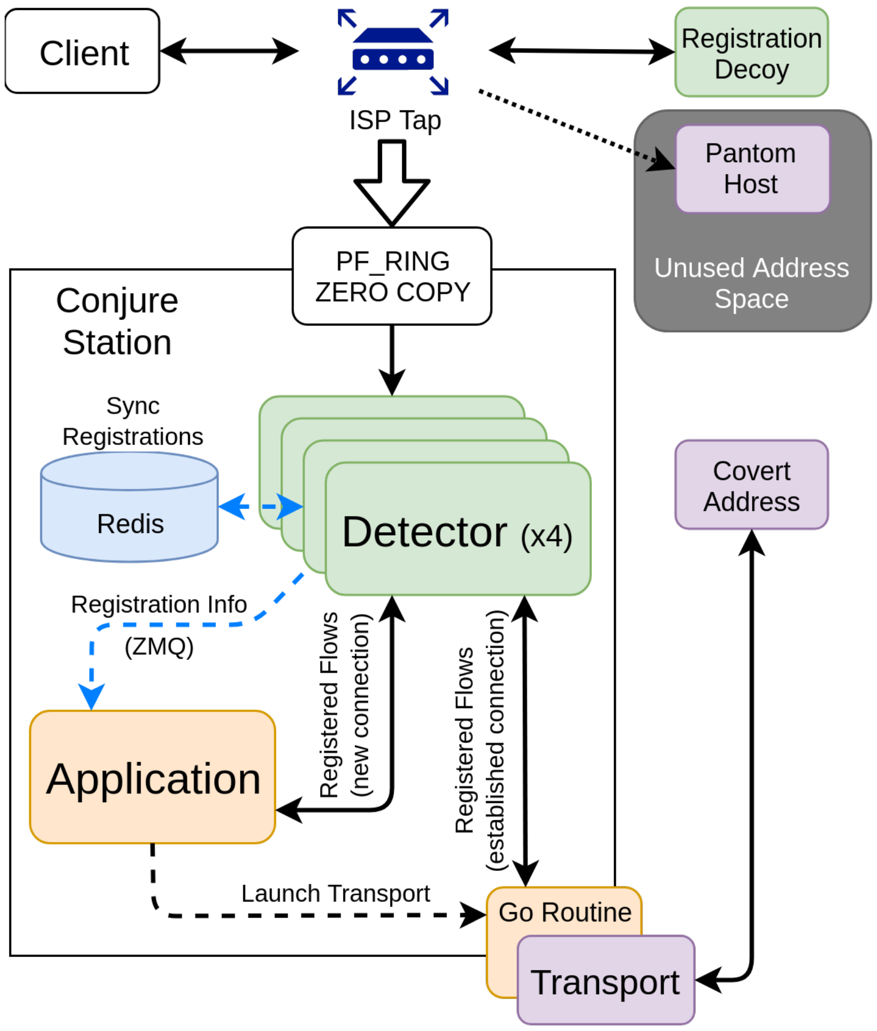


Figure 2.15: **Station Architecture** — We used PF_RING to receive packets from a 20 Gbps tap which we load balance across 4 CPU cores. The detector processes identify registrations, sending notification to all other cores via redis and to the local application via ZMQ when a new registration is received. The detector also identifies packets associated with registered flows and diverts them to the local application which handles proxying connections. The local application determines which transport the flow should use based on a parameter specified in the client's registration and initializes a goroutine to handle forwarding.
A censor trying to interfere in this connection would need to take over the client's IP address in a precise window (after the registration but before the client connects to the phantom) and correctly guess the phantom host's IP address (or connect to all possible phantom IPs) before the active probe traffic gets to the local application. At this point their connection will fail in a manner specific to the transport that the client specified in the registration (e.g. connecting to OSSH without knowledge of the pre-shared secret).

A second problem arises when considering how to **timeout** unused phantom proxies in the detector. When a phantom proxy is timed out, it no longer responds to any active probes. A naive implementation might simply have each core timeout unused phantom proxies after they go unused for a set time. However, this could leave one core (that sees active proxy use) forwarding packets to the application, while other cores (that do not see use) would timeout the proxy. A censor could probe the phantom proxy and observe this behavior: if the censor's packets are processed on a forwarding core, the censor can establish a TCP connection with the phantom application. Otherwise, if they are processed on a timed out core, the censor's packets will be ignored. Through multiple connections, the censor could use this strange behavior (of intermittent TCP response) to identify and block potential phantom proxies.

To address this issue of differing core timeouts, we implemented a new load-balancing algorithm in PF_RING to select the core based only off the destination IP address of a packet. This means that **all** packets sent to a particular phantom proxy address are processed by the same detector core, which allows the phantom proxy's timeout state to be consistent regardless of what source attempts to connect to it.

2.3.5.2 Client

We created a Conjure client and integrated it with the Psiphon [119] anticensorship client. Psiphon has millions of users in censored countries, and we are in the early stages of rolling Conjure out to real-world users.

Our Conjure client is written in Golang, and uses the Refraction Networking tagging schema (Section 2.3.4.1) for registration. We note that this protocol will be more difficult for censors to observe than normal TapDance because it consists of only a single (complete) request, and the station does not have to spoof packets as the decoy during registration, only passively observe them.

We implemented support for two of the transports in our client: Obfuscated SSH used by Psiphon (Section 2.3.4.2) and our TLS Mask Site protocol (Section 2.3.4.2). Our client signals which transport it will use in the registration tag along with a flag indicating whether the client supports IPv6, allowing IPv4-only clients to derive exclusively IPv4 phantom hosts. After registration, the client connects to the derived phantom host address, and speaks the specified transport protocol, which tunnels between the client and either the mask site transport local to the station or Psiphon's backend servers. A SOCKS connection can be initiated through this tunnel to allow for connection multiplexing.

2.3.5.3 Application & Transports

We implemented our station-side application in about 500 lines of Golang. This includes support for OSSH (via integration with Psiphon) and Mask Sites, both specifiable by the client during registration. We note that support for other transport protocols (e.g. obfs4 or WebRTC) can be added as development continues.

OSSH

Our Conjure implementation includes support for OSSH through integration with Psiphon. Client traffic is forwarded to a Psiphon server by using Conjure as a transparent proxy. This symbiotic relationship provides Conjure with active and passive probe resistance, while preventing censors from being able to inexpensively block Psiphon's backend servers individually.

Mask Sites

We implemented a mask site mimicking proxy, that pretends to be a mask site when actively probed by the censor. Once the station accepts accepts a connection for a registered flow, it initially acts as a transparent proxy to a mask site specified by the client during registration. The application parses the handshake, forwarding packets back and forth between client and mask site without modification, extracting the server and client randoms. The application attempts to decrypt the first application data record from the client using a key derived from the secret seed, client, and server randoms. We use the uTLS library [62, 112] on both the application and client to allow us to change the TLS secrets being used after the handshake.

If the decryption is successful, the application switches to forwarding (decrypted) data back and forth with a client-specified endpoint, such as a SOCKS proxy, which can provide multiple secure connections over the single connection to the phantom host.

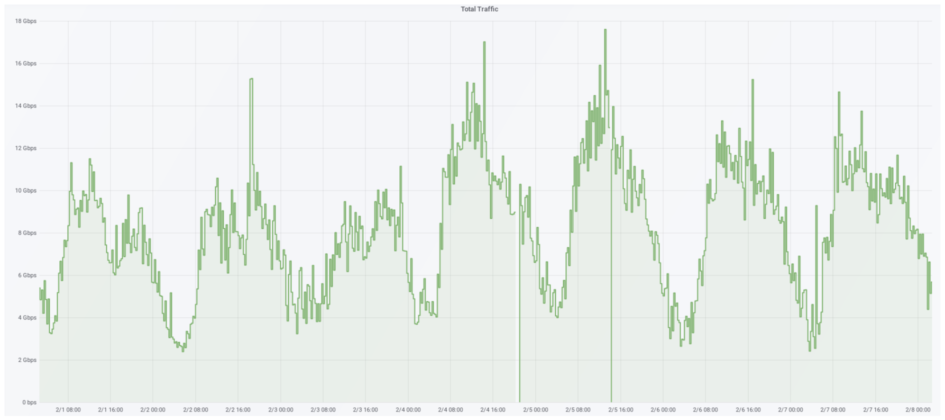


Figure 2.16: **Tap Bandwidth** — We deployed our Conjure implementation in a realistic ISP testbed on a tap of a 20 Gbps router. As shown in a typical week, traffic ranges from 2–17 Gbps.

2.3.6 Evaluation

To evaluate our Conjure implementation, we compare its bandwidth and latency to that of TapDance in a realistic ISP setting. We used a 20 Gbps network tap at a mid-sized ISP and run both implementations on a 1U server with an 8-core Intel Xeon E5-2640 CPU, 64GB of RAM, and a dual-port Intel X710 10GbE SFP+ network interface. A typical week of bandwidth seen on the tap is shown in Figure 2.16, ranging from 2.4 Gbps to peaks above 17 Gbps.

2.3.6.1 Performance

We evaluated the performance of a client from an India-based VPS. Figures 2.17 and 2.18 show the upload and download bandwidth as measured by iperf for TapDance, our Conjure implementation (using the mask site application), and a direct connection to our iperf server in the ISP's network.

TapDance must reconnect if the amount of data sent by the client exceeds a short TCP window (typically on the order of 32 KB) or the connection persists until a timeout (18–120 seconds). At each reconnect, the TapDance client naively blocks until a new TLS connection to the decoy and station has been established. Thus, when uploading files, TapDance has to create a new TLS connection for every 32 KB of data it sends, limiting its average upload bandwidth to around 0.1 Mbps due to the high overhead. In contrast, our Conjure implementation is able to maintain the same connection during large uploads, and achieves

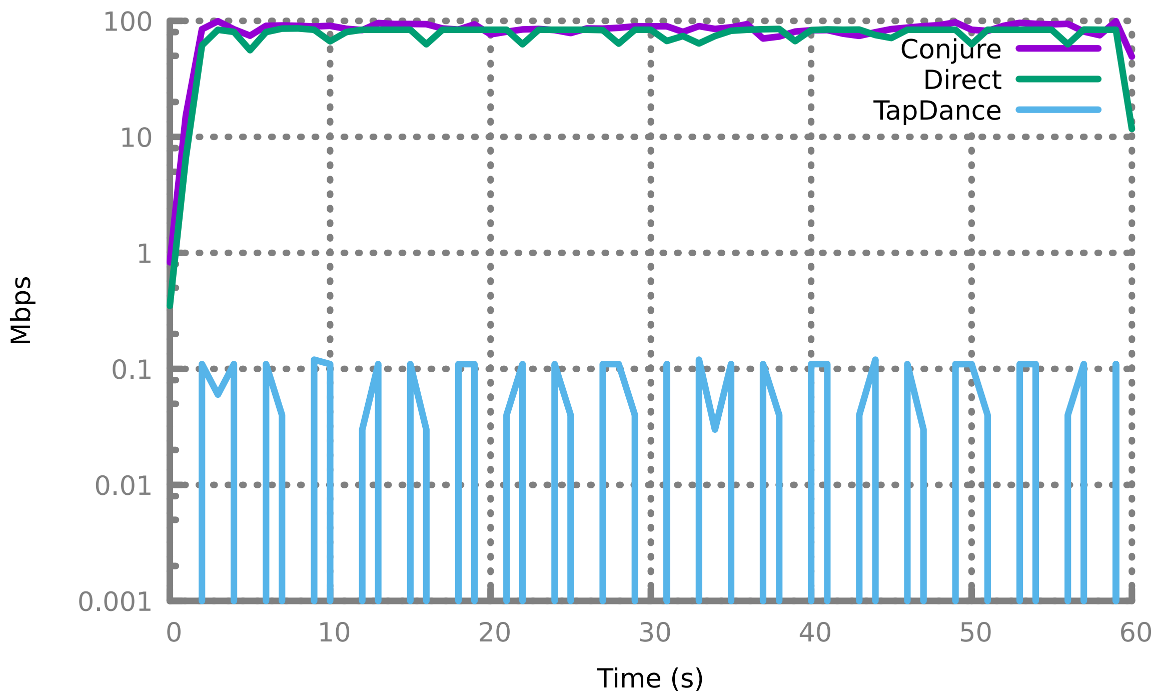


Figure 2.17: **Upload Performance** — We used iperf to measure the upload bandwidth for a direct connection, TapDance, and Conjure. As expected, TapDance's upload performance is several orders of magnitude lower than the link capacity, due to the overhead of frequent reconnects to the decoy.

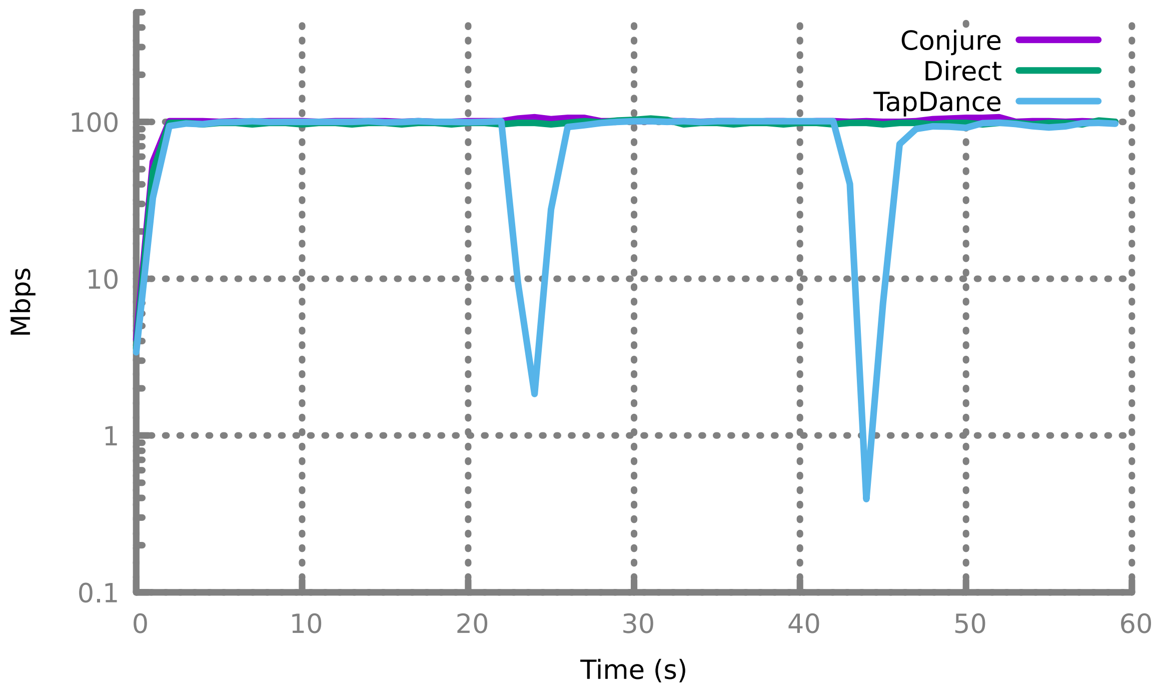


Figure 2.18: **Download Performance** — Using iperf, we compare the download performance of a direct connection, TapDance, and Conjure. While TapDance can achieve link capacity download, it still has to occasionally reconnect to the decoy, as seen by the periodic dips. These reconnects are more common the more data the client sends (e.g. requests or any upload data).

performance inline with the direct connection, over 1400 times faster.

During download-only workloads, TapDance is able to better utilize the network, but must still reconnect before the decoy times out. In our tests, we see TapDance reconnect every 25 seconds, which can negatively impact the performance of downloads or any real-time streaming applications. Again, our Conjure implementation is able to maintain a single connection and provide the maximum download rate without interruption, 14% faster than TapDance.

We also measure the latency of repeated small requests. In both Conjure (using the OSSH protocol) and TapDance we establish a single session tunnel using our integrated Psiphon client, and make 1,000 requests through each using Apache Benchmark (ab). We find that our India-based VPS throttles TLS but not OSSH, making TapDance twice as slow as Conjure. We repeated these tests on a US-based VPS which does not have such throttling, and show results in Figure 2.19. TapDance's frequent reconnects adds significant latency to about 10% of requests. In addition, the median latency of Conjure is about 19% faster, due to the added overhead of TLS and the complex path that TapDance data packets take through the station compared to Conjure.

2.3.6.2 Address Selection

Phantom host IP addresses must be derived from network blocks that are routed (so they pass the Conjure station) and contain other legitimate hosts (so that censors cannot block the entire network without collateral damage). Because of the large number of IPv6 addresses, even moderately-sized network prefixes have astronomical numbers of addresses: a single /32 prefix has 2^{96} possible addresses. Therefore, client-chosen seeds have negligible probability of corresponding to addresses that are already being used by legitimate hosts. This allows us to select phantom host addresses from network prefixes that contain legitimate hosts—crucial to discouraging the censor from blocking them outright—without worry that registrations could interfere with legitimate services.

IPv4 While Conjure works best with IPv6, it can also support IPv4, with some careful caveats.

First, in IPv4, there are substantially fewer addresses, allowing censors to potentially **enumerate all the network prefixes** that pass by the ISP station, compose the list of innocuous sites, and block other

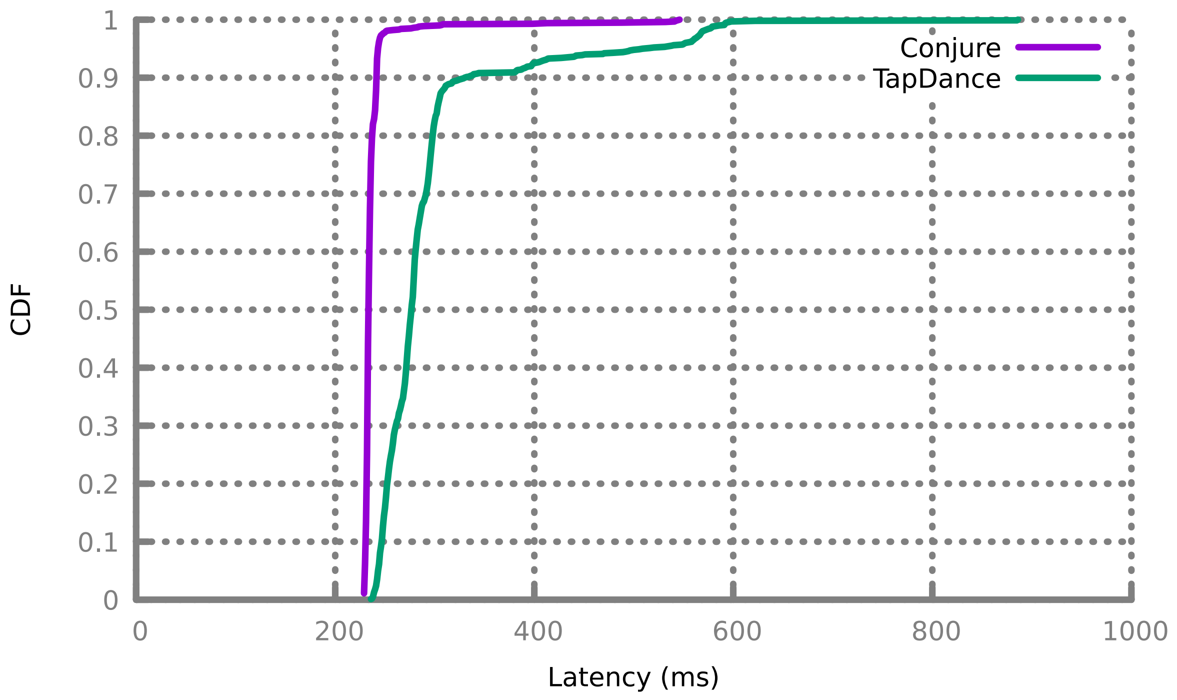


Figure 2.19: **Latency** — We compare the CDF of latency for requests between Conjure and TapDance. In each case, we have a long-lived session over which requests are being made using Apache Benchmark (ab). At the median, Conjure has 44 ms (19%) faster latency than TapDance. In addition, TapDance requests are frequently slowed by its intermittent reconnections to the decoy, as shown in the longer tail of TapDance's latency CDF. Conjure has no such reconnects, and thus has more uniform latency. At the 99th percentile, Conjure is 281 ms (92%) faster than TapDance.

websites, as they are being summoned by Conjure. To address this, Conjure phantom hosts are firewalled from all IPs other than the client that registered them, providing a reason why the address hasn't been seen in an enumerating scan, conducted by a censor from a single vantage point. Censors could attempt to scan the network from **all** potential client vantage points, by co-opting client IPs to perform scans—a behavior previously observed by the Great Firewall of China to scan for Tor bridges [47]. To prevent this, for IPv4 Conjure, we dynamically generate the TCP port of the phantom host (along with its IP) from the registration seed, which further makes exhaustive scans infeasible: a censor that must enumerate from the vantage point of a /10 of client IPs (4 million IPs) to a /16 (65K IPs) of potential phantom proxies on each of 65K potential ports would take nearly 50 years of scanning with ZMap at 10 Gbps. We note that while the use of non-standard ports could potentially be suspicious, several successful circumvention tools—including Psiphon [119] and obfs4 [147]—use random ports on their obfuscated protocols. Finally, we note that censors that whitelist either standard ports or discovered hosts from enumerations scans would over-block new services that came online after their scans.

A second problem in IPv4 Conjure is that the limited range of IPs (and ports) makes it possible for a censor to **pre-image the hash** used to derive the phantom address from the seed. Even with the /16 of IPs and all 65K ports, in order to find a seed for any desired address a censor needs to only test an expected 2^{32} possible seeds. The censor could then register a suspected address, and see if it provides proxy access. If it does, the censor learns there is no legitimate service there, and can block it. To combat this, we allow only a single client to register for a particular phantom address at a time. A censor could attempt to register all addresses in an attempt to deny proxy service to legitimate users, but this would be easily observed at the registration system, where rate limits via client puzzles or account-based fees could be enforced.

Finally, legitimate **IPv4 addresses density** is much higher than IPv6, increasing the potential for users to accidentally register seeds that derive phantom addresses corresponding to live hosts. To address this, the station sends probes to potential IPv4 phantom hosts during registration, and ignores the registration if a real host responds. Censors that try to register specific phantom proxies will be unable to distinguish if another user has registered it or a legitimate host is there, as in both cases we ignore the censor's registration. This also serves to prevent abuse by attackers that attempt to use the Conjure station to interfere with innocuous services.

IPv6

In IPv6 Conjure address enumeration and pre-image attacks are infeasible due to a large amount of potential IP addresses. Our ISP routes a /32 IPv6 prefix, which provides 2^{96} potential phantom host IP addresses. However, IPv6 addresses often have long runs of 0 bits in them, and rarely use all 128-bits equally. For instance, google.com has the address `2607:f8b0:400f:806::2004`, which only has 24 bits set to 1. Censors might try to use this observation and block high entropy IPv6 addresses.

To measure and quantify this problem, we collected and analyzed 16 hours of netflow data at our ISP tap. We extracted 4013 IPv6 addresses observed in the /32 IPv6 prefix routed by the ISP (out of 32,817 total observed). To confirm the hypothesis that 0 bits are more common in IPv6 addresses, we counted the number of bits set in each address. Figure 2.20 shows a histogram of the number of bits set and compares it to the histogram of a uniformly random set of addresses.[4] Although these distributions are distinguishable, they do have significant overlap. Given enough samples, a censor could trivially tell if the addresses were chosen randomly or were legitimate hosts. However, a censor's job is significantly harder, and they must tell from a single sample which distribution it comes from. The presence of random-looking addresses makes it difficult for censors to block such hosts outright.

Prior work by Foremski et al. [56] has developed models to generate likely IPv6 addresses from a set of known addresses, useful for discovering new hosts to scan given known ones. We use their Entropy/IP tool to analyze the addresses we collected. Figure 2.21 shows the normalized entropy of each address 4-bit nibble and the total entropy (18.8 out of 32). Nibbles that were constant across all addresses (such as the /32 network prefix nibbles) have zero entropy, while those that equally span the range of values have maximum entropy (normalized to 1). In our addresses, we observe an entropy of over 75 bits, with more entropy in the later segments of the address. While not quite the full 96 bits that uniformly random would produce, this is still a significant amount for phantom hosts to hide in.

For a specific deployment, operators should be careful to observe the distribution of addresses in the subnets they use, and possibly limit to randomizing "realistic" bits (e.g. the upper and/or last 32 bits within

[4] The random distribution's center is skewed from 64 due to the number of set bits in our fixed /32 network prefix.

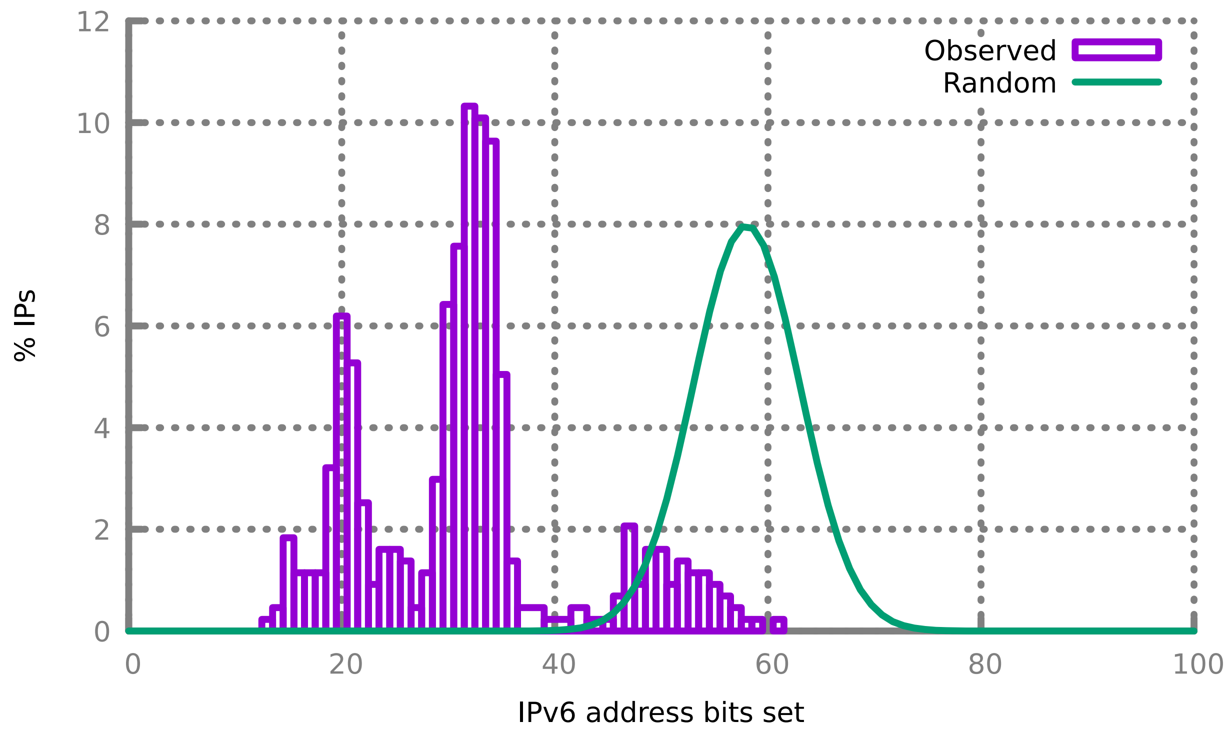


Figure 2.20: **IPv6 Bits Set** — We measure the number of bits set to 1 in IPv6 addresses in our ISP's /32 and observed by our tap, and compare it to the Binomial distribution (Random) we would expect to see if the 96 non-network bits of each address were chosen randomly. In practice, we observe many fewer bits set. Nonetheless, the significant overlap of these distributions would make it difficult for censors to block any individual phantom hosts without collateral risk of blocking legitimate services.

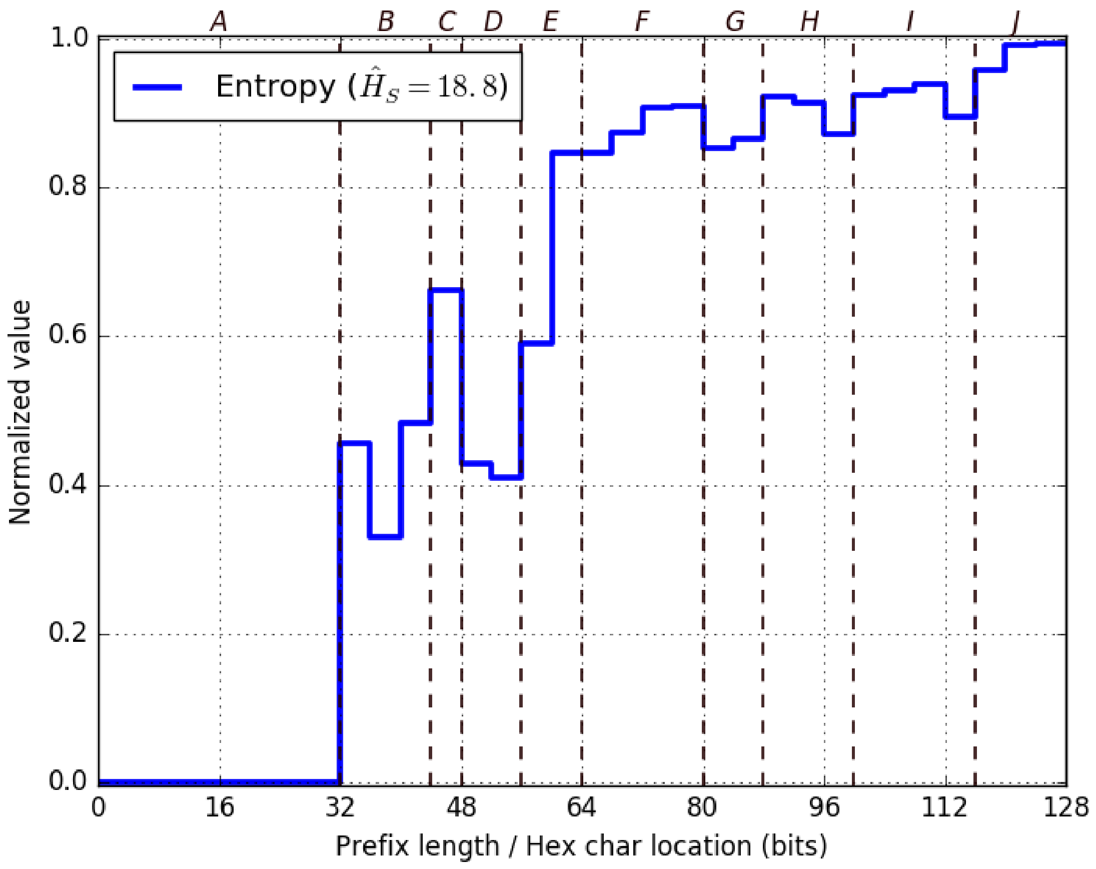


Figure 2.21: **IPv6 Entropy** — Censors may be able to distinguish random-looking phantom host IPv6 addresses from legitimately used ones based on the addresses' entropy. We used the Entropy/IP tool [56] to analyze the entropy of 4,013 IPv6 addresses observed by our tap. This plot from the tool shows the normalized entropy of each nibble in the addresses, which is fairly high for most of the non-network-prefix nibbles. In total, these addresses have about 75 bits of entropy (out of 96 expected), and the relatively high entropy present in each nibble would make it difficult for a censor to block without significant false positives / negatives. While distinguishable from random given enough samples, we can also use the Entropy/IP tool to generate addresses from the Bayesian Network model it produces.

the given /32). As an improvement, we could also use the Entropy/IP tool [56] to generate the random IPv6 phantom hosts from the registration seed based on the Bayesian Network model created by the tool.

2.3.7 Attacks and Defenses

In this section, we discuss several attacks a censor might attempt to either block phantom hosts from being registered or used.

2.3.7.1 Probing phantom hosts

Censors may attempt to actively probe suspected phantom hosts to determine if they are proxies or real hosts. China has been observed using exactly this technique to discover existing proxies and Tor bridge nodes [30, 47, 75, 160]. In response to China's active probing, Tor and other circumvention tool developers have developed **probe-resistant** protocols, such as obfs4 [147], Obfuscated SSH (OSSH) [91], and Shadowsocks [137]. Each of these protocols require the client to know a special secret distributed alongside the original proxy address. Without knowledge of this secret, active-probing censors will not receive any response from these hosts, making it difficult for censors to tell if a server is a proxy or a non-responsive legitimate host.

Obfuscated SSH Modern OSSH protocols are intended to be probe-resistant. Censors that attempt to probe suspected OSSH servers without knowledge of the secret receive no data response, making it difficult to distinguish them from other non-responsive hosts. To confirm this, we use ZMap [42] to scan over 1 billion random IP/port combinations, and sent 25 random bytes (corresponding to the OSSH handshake) to the approximately 800,000 servers that responded. We expect very few of these to actually be OSSH servers as they are simply random servers on random TCP ports. However, over 99.4% of them did not respond with any data, behavior mirrored by OSSH servers. Furthermore, 7.42% of servers closed the connection with a TCP RST after we send our random data, a response we also see with OSSH. While there may be other ways to differentiate OSSH servers from others online, our tests suggest censors could face steep false-positive rates in identifying OSSH servers with active probing.

obfs4 Unlike previous versions of obfsproxy, the obfs4 protocol is designed to be resistant to active probing attacks, requiring the client prove knowledge of a secret before the server will respond. We verified that naive active probing attacks (where we attempt to connect to an obfs4 server without knowledge of the secret and see if it provides proxy access) do not work against obfs4. In addition, although China is effective at blocking `obfs3`, the more recent probe-resistant obfs4 remains a viable proxy in the country.[5]

Mask sites The censor could also attempt to fingerprint a masked site and compare it to a suspected Conjure application. For instance, if a phantom host IP responds to a censors' probes pretending to be `example.com`, the censor could probe real instances of `example.com` on different ports and see how it responds. Then, the censor can probe the phantom host IP, and see if it responds similarly (e.g. with the same set of open ports and payloads for certain kinds of probes). To defend against this, we forward **all** traffic destined to the phantom host to the masked site, including ports that are not relevant to the proxy application (e.g. non 443). This ensures that above the TCP layer, we appear to be the mask site. However, there may be differences in TCP/IP implementations, for instance, how IP IDs are incremented, or how TCP timestamps are incremented (or supported at all) that may be different from the mask site. To combat this, we can filter mask sites by those that have identical TCP/IP stacks to ours, as we use a common Linux implementation. We also note that this attack only applies when we use the mask site as an application, and that Conjure can support other applications (e.g. obfsproxy, WebRTC, etc) that do not have this issue.

We acknowledge that perfect mimicry of the real site is likely infeasible: Houmansadr et al. [71] demonstrated the difficulty in fully mimicking known applications, showing that staying fully-feature compatible is an expensive and potentially intractable problem for circumvention systems. However, their study focuses on **application-specific** mimicry, used by circumvention tools like CensorSpoofer, StegoTorus, and Skypemorph. These tools generally attempt to mimic a specific protocol used by a complex application such as Skype, where only one or a handful of official popular clients are used. Houmansadr et al. [71] show that such mimicry is difficult in practice due to the complexity and opaqueness of specifications, side protocols that applications may use, and intra- and inter-dependence on connections and other protocols.

[5] `https://metrics.torproject.org/userstats-bridge-combined.html?start=2019-06-27&end=2019-09-25&country=cn` shows obfs4 clients from China successfully using Tor.

In contrast, mask sites have the advantage that we can mimic any sites from a large set of heterogeneous real websites, making it harder for a censor to fully fingerprint and characterize **all** potential mask sites. As websites may themselves use distributed or heterogeneous server implementations, it could be more difficult for a censor to block all mask site application use. Nonetheless, it is still possible for censors to use active probing attacks to detect either protocol feature, payload size, or timing differences between mask sites and the real sites they attempt to mimic.

2.3.7.2 Passive identification

Censors might also attempt to use passive methods to distinguish and block communication with registration decoys or phantom hosts. For example, Website Fingerprinting [67, 139, 156] (WF) uses the patterns of encrypted traffic to identify which website a client is connected to. WF uses a classifier to label the traffic as belonging to one of several known websites, or (in some variants), an unknown or background class. Though WF is commonly studied in the context of anonymous web browsing, a censor could also use WF to detect Conjure applications by distinguishing known traffic patterns of legitimate website use from traffic generated by Conjure (unknown class). Alternatively, the censor could monitor traffic for patterns consistent with tunneled access to a set of blocked websites.

Wang et al. [153] demonstrated several effective classifiers that can passively distinguish normal traffic from obfuscated circumvention protocols including `obfs3`, obfs4, Format-transforming encryption [43], and meek [54]. However, despite the low false positive rate of these classifiers, Wang et al. [153] caution that the base rate of normal traffic compared to obfuscated traffic would force censors that employ such attacks to suffer significant collateral damage in degrading performance and reliability of legitimate traffic. Indeed, to date, censors have not been observed using WF or other passive techniques, possibly due to the inaccuracy of these identification techniques in practice. Even small false positive rates means blocking mostly legitimate connections, and false negatives could allow clients to retry until they gain access.

More importantly, Conjure applications offer great flexibility in deploying traffic analysis defenses; for example, traffic shaping strategies such as implemented in Slitheen [18] could be easily employed in Conjure. Conjure clients could also choose from a large set of potential applications, forcing censors to have to block access to all of them to block use.

2.3.7.3 Blocking registration

To make registration more difficult, censors could block TLS connections to all of the limited decoys available in a deployment. Using TapDance's current deployment, this would involve blocking over 1500 sites. We note that such an attack would completely disable all existing Refraction Networking schemes, as none work without being able to access legitimate decoys past the station. In Conjure, this would only block new registrations, and would not impact users that previously registered. Furthermore, registrations could also occur over email, or over lower-bandwidth covert channels, such as port (or even IP) knocking past the station, that would be more difficult for the censor to block.

2.3.7.4 ICMP

Censors can use ping or traceroute utilities (via ICMP) to probe potential phantom hosts. Because there is (usually) no host at the phantom host address, these probes will timeout and produce no response. They might also produce "Destination Unreachable" responses from routers depending on how they are configured. We performed a scan of 10 million IPv6 addresses in a routable /32 prefix to see if it is common to respond with such tell-tale ICMP messages for unused messages. We found only 0.016% of addresses responded with any ICMP messages (mainly "Time Exceeded" and "Destination Unreachable").

Many legitimate hosts and routers do not respond to or forward ICMP packets, and it is common for firewalls to block traceroutes from penetrating inside networks. Thus, simply ignoring ICMP messages (or low TTL packets that might be used by traceroute) may be a viable strategy. Alternatively, we could spoof responses to convince an adversary that a phantom host is part of a particular network. However, this strategy requires careful consideration of what network makes sense for a mask site to be in. Also, the censor may try to probe for addresses around the phantom host (but still likely to be in the same network), which must also be responded to.

2.3.8 Related Work

We first compare Conjure with other Refraction Networking schemes and then discuss other related work.

2.3.8.1 Prior Refraction Networking Schemes

Since 2011, there have been several proposed Refraction Networking schemes. Telex [165], Cirripede [72] and Decoy Routing (aka Curveball) [82] are "first generation" protocols with nearly identical features. These designs require inline flow blocking at the ISP to allow the station to intercept flows with detected tags and act as the decoy host for them. However, inline blocking is difficult for ISPs to deploy, as it requires special-purpose hardware to be placed inline with production traffic, introducing risk of failures and outages that may be expensive for the ISP and potentially violate their contractual obligations (SLAs).

TapDance [164] solves the issue of inline-blocking by coercing the decoy into staying silent, and allowing the station to respond instead. However, as previously described, this trick comes at a cost: the decoy only stays silent for a short timeout (typically 30–120 seconds), and limits the amount of data the client can send before it responds. TapDance clients must keep connections short and repeatedly reconnect to decoys, increasing overhead and potentially alerting censors with this pattern. Conjure addresses this issue and allows clients to maintain long-lived connections to the phantom host.

Rebound [45] and Waterfall [111] both focus on routing asymmetries and routing attacks by the censor. Rebound modifies the client's packets on the way to the decoy, and uses error pages on the decoy site to reflect data back to the client. Waterfall only observes and modifies the decoy-to-client traffic, similarly using error pages on the decoy to reflect communication from the client to the station. These schemes also provide some resistance to traffic analysis, as they use the real decoy to reflect data to the user. Thus, the TCP/TLS behavior seen by the censor more closely matches that of a legitimate decoy connection. However, latency and other packet-timing characteristics may be observable, and both schemes require some form of inline flow blocking.

Slitheen [18] focuses on addressing observability by replacing data in packets sent by the legitimate decoy. Thus, even the packet timings and sizes of a Slitheen connection match that of a legitimate decoy connection. However, Slitheen also requires inline-blocking, and introduces a large overhead as it has to wait for the subset of data-carrying packets from the decoy that Slitheen can safely replace. We note that the Slitheen model of mimicry is compatible with Conjure, as we could use Slitheen as the application protocol. Despite using a passive tap, our scheme is effectively inline to the phantom host (which won't otherwise respond).

Bocovich and Goldberg propose an asymmetric gossip scheme [17] that combines a passive monitor on the forward path from the client to the decoy with an inline blocking element on the return path. These elements work in concert to allow schemes such as Telex and Slitheen to work on asymmetric connections. This approach, however, still requires inline blocking on one direction, and further complicates deployment by requiring the installation of more components and potentially complex coordination between them. MultiFlow [99] uses refraction networking only as a forward mechanism to communicate a web request to the station, and then uses a bulletin board or email to deliver the response back. It does not require inline flow blocking as it does not modify users' traffic at all, but it fundamentally relies on a separate data delivery mechanism, similar to other cloud- or email-based circumvention tools [20, 74].

Conjure allows a large amount of flexibility compared to previous schemes. Because we have significant degrees of freedom in choosing the specific application the phantom host will mimic or talk, our scheme can combine the best of existing Refraction Networking protocols to achieve high performance, be easy to deploy, and also be resistant to active attacks such as replaying or probing by the censor. Table 2.2 lists the existing Refraction Networking schemes and their features, as compared to Conjure.

2.3.8.2 Decoy Placement and Routing Attacks

Houmansadr et al. [72] found that placing refraction proxies in a handful of Tier 1 networks would be sufficient for them to be usable by the majority of the Internet population. Cesareo et al. [24] developed an algorithm for optimizing the placement of proxies based on AS-level Internet topology data. Schuchard et al. [134] suggested that a censor may actively change its routes to ensure traffic leaving its country avoids the proxies, but Houmansadr et al. [73] suggested that real-world constraints on routing make this attack difficult

	Telex [165]	Cirripede [72]	Decoy Routing [82]	TapDance [164]	Rebound [45]	Slitheen [18]	Waterfall [111]	**Conjure**
No inline blocking	○	○	○	●	○	○	○	●
Handles asym. routing	○	●	○	●	●	○	●	●
Replay attack resistant	●	●	●	○	●	●	●	●
Traffic analysis resistant	○	○	○	○	◐	●	◐	○
Unlimited Session Length	●	●	●	○	○	○	○	●

Table 2.2: **Comparing Refraction Networking Schemes** — "No inline blocking" corresponds to schemes that can operate as a passive tap on the side without needing an inline element in the ISP network. "Handles asymmetric routes" refers to schemes that work when only one direction (either client to decoy or decoy to server) is seen by the station. "Replay attacks" refers to censors who may replay/preplay previous messages or actively probe the protocol. "Traffic analysis" includes latency, inter-packet timing, and website fingerprinting. "Unlimited Sessions" shows schemes that do not need to repeatedly reconnect to download or upload arbitrarily large content.

to carry out in practice. Nevertheless, Nasr et al. [110] propose a game-theoretic framework to optimize proxy placement in an adversarial setting, and the design of Waterfall [111] is in part motivated by resilience to routing attacks, as it is more difficult for the censor to control the return path from a decoy site, rather than the forward path.

In practice, deployment of refraction networking has so far been at access, rather than transit ISPs [59]. This may be in part because a transit ISP has a large number of routers and points-of-presence, significantly raising the costs of deployment [66].[6] Likewise, we expect Conjure to use address space announced by the ISP, rather than addresses relayed by it, which mitigates routing-based attacks. Depending on the size of the ISP, however, a censor may decide to block the entirety of its address space, which would incur smaller collateral damage than blocking all addresses seen by a transit ISP.

2.3.8.3 Avoiding Destination Blocking

Traditionally, proxies deployed for censorship are eventually identified and blocked by the censor. Several proposals have been made to carefully control the distribution of proxy addresses, using social

[6] We note that Gosain et al. [66] use an estimate of $885,000/proxy, while Frolov et al. [59] report line-rate TapDance deployment using commodity hardware that costs only several thousand dollars.

connections and reputation [38, 103, 155]. Nevertheless, keeping this information secret is challenging; additionally, censors often employ active scanning techniques to discover proxies [39]. Refraction networking generally assumes that clients have no secret information, and instead relies on the collateral damage that would result from blocking all the potential decoy destinations. Conjure furthers this goal by creating a large number of destinations out of the dark space. A similar approach was conceptualized in DEFIANCE [95], where censored Tor clients connect to pools of addresses that are volunteered to run Tor bridge nodes. DEFIANCE also requires volunteer web servers to run specialized servers to distribute information. Unlike Conjure, DEFIANCE was not designed to run at an ISP, and involves many moving parts that present single points of failure if blocked by a censor. In contrast, Conjure has a relatively simple yet flexible design, allowing it to easily respond to censors. Another similar approach was taken by CensorSpoofer [154], which spoofed traffic from a large set of dummy destinations. CensorSpoofer, however, could only send information in one direction—to the client—and had to rely on a separate out-of-band channel for client-to-proxy communication. As an alternative approach, flash proxy [53] and Snowflake [36] allow users to run WebSocket-based or WebRTC-based proxies within their browser to allow censored users to connect to the Tor network with the potential to greatly increase the number. In practice, these proxies served a very small number of users, as compared with other Tor bridge transports.[7]

2.3.9 Conclusion and Future Work

Conjure provides a much larger degree of engineering flexibility than previous Refraction Networking schemes. Due to its modular design, different registration protocols and proxy transports can be used interchangeably by the client. The flexibility of proxy transports and simplicity of registration allows Conjure to incorporate state of the art censorship circumvention tools and resist nation-state censors.

One obvious future direction is to study new options for registration and proxy transport. For instance, while Conjure currently uses a TapDance-style covert channel for registration, we could potentially cut down on the overhead of one-time registration by using port-knocking or using a Telex-style [165] tag (in the

[7] `https://metrics.torproject.org/userstats-bridge-transport.html?start=2017-01-01&end=2019-02-15&transport=!<OR>&transport=websocket&transport=snowflake`

ClientHello rather than Application Data).

Client-side applications Conjure provides an interesting opportunity to explore **client mimicking** phantom hosts. Rather than pretend to be a server (e.g., a mask site), our transport itself could connect to a newly registered client from the phantom host address. Possible protocols could include WebRTC, mentioned in Section 2.3.4.2, or other peer-to-peer protocols such as BitTorrent, Skype, or Bitcoin.

Traffic analysis Conjure could also support applications that tradeoff performance for observability. While Slitheen offers ideal mimicry of decoys, it comes at a high cost of overhead. Conjure transports such as mask site could implement Slitheen in order to perfectly mimic the decoy site's latency, packet timings, and payload sizes. In addition, careful choice of mask sites may allow for higher performance, as sites with more replaceable content can carry more covert data.

Long-term deployment Ultimately, the goal for Refraction Networking protocols is to be useful in circumventing censorship. While it has taken many years for research protocols to mature, we are excited to see schemes like TapDance deployed in practice [59]. We believe Conjure can be even easier to deploy at scale, and we hope to leverage the existing success of TapDance to place Conjure stations at real ISPs.

2.3.10 Acknowledgements

The authors thank the incredible partner organizations that have made deployment of Refraction Networking a reality, especially Merit Network and Psiphon. We also thank the University of Colorado IT Security and Network Operations staff. This material is based in part upon work supported by the U.S. National Science Foundation under Awards CNS-1518888 and OAC-1925476.

Chapter 3

TLS Fingerprinting

3.1 Background

TLS Fingerprinting has several advantages over Active Probing and Enumeration Attacks. Detected connections may be terminated on the spot by the same DPI device that detected the fingerprint with no time delay. TLS Fingerprinting also does not require the usage of additional devices, which would be necessary for probing or running client software for enumeration, nor is there a need to propagate the IP addresses to block across the censorship apparatus. TLS Fingerprinting could also be used in combination with Active Probing: the fingerprint raises suspicion, while an active probe confirms it. As such, censors have been using this attack for a long time. As early as January 2011, Iran has been reported [1] to fingerprint Tor connections based on the Diffie–Hellman parameter offered in TLS ClientHello. In September of the same year, Iran blocked Tor based on their TLS certificate lifetime, and in December, China was reported [81] to send their active probes based on ciphersuites. In 2012, Iran blocked Tor twice again using TLS fingerprinting [1], and was joined by Ethiopia [131], Kazakhstan [132], United Arab Emirates [133], Philippines [1], and Syria [1]. The high prevalence of TLS Fingerprinting attacks made development of countermeasures a high priority.

The long history of the use of TLS Fingerprinting by numerous censors motivates the development of a countermeasure that would defeat this attack. In the next section, I describe how we organized the collection of ClientHello fingerprints, used the data to test censorship circumvention tools' fingerprintability, found vulnerable ones, and developed a practical defense against the attack.

3.2 The use of TLS in Censorship Circumvention

3.2.1 Introduction

The Transport Layer Security (TLS) protocol is quickly becoming the most popular protocol on the Internet, securing network communication from interference and eavesdropping. Already, 70% of page loads by Firefox users make use of TLS [142], and adoption continues to grow as more websites, services, and applications switch to TLS.

Given the prevalence of TLS, it is commonly used by circumvention tools to evade Internet censorship. Because censors can easily identify and block custom protocols [71], circumvention tools have turned to using existing protocols. TLS offers a convenient choice for these tools, providing plenty of legitimate cover traffic from web browsers and other TLS user, protection of content from eavesdroppers, and several libraries to choose from that support it.

However, simply using TLS for a transport protocol is not enough to evade censors. Since TLS handshakes are not encrypted, censors can identify a client's purported support for encryption functions, key exchange algorithms, and extensions, all of which are sent in the clear in the first **ClientHello** message.

In fact, popular tools such as Tor have already been blocked numerous times due to their distinctive SSL/TLS features [1, 32, 92, 131–133, 160]. Even tools that successfully mimicked or tunneled through other popular TLS implementations have suffered censorship. For example, in 2016, Cyberoam firewalls were able to block meek, a popular pluggable transport used in Tor to evade censors, by fingerprinting its TLS connection handshake [49]. Although meek used a genuine version of Firefox bundled with Tor, this version had become outdated compared to the rest of the Firefox user population, comprising only a 0.38% share of desktop browsers, compared to the more recent Firefox 45 comprising 10.69% at the time [141]. This allowed Cyberoam to block meek with minimal collateral damage.

The problem was temporarily corrected by updating to Firefox 45, but only a few months later, meek was blocked again in the same manner, this time by the FortiGuard firewall, which identified a combination of SNI extension values sent by meek and otherwise matching the signature of Firefox 45 [50]. At that time, Firefox 47 had been released, supporting a distinguishable set of features. The rapid pace of new implementations and versions is a difficult task to keep up with.

Another motivating example of these challenges is found in the Signal secure messaging application [145]. Until recently, Signal employed domain fronting to evade censorship in several countries including Egypt, Saudi Arabia, and the United Arab Emirates [54, 102]. However, due to a complicated interaction with the library it used to implement TLS, we find that ClientHello messages sent by Signal while domain fronting differ from their intended specification, ultimately allowing them to be distinguished from the implementations they attempted to mimic and easy for censors to block[1] .

These examples demonstrate the difficulties in making TLS implementations robust against censorship. To further study this problem, we collect and analyze real-world TLS handshakes, and compare them to handshakes produced by several censorship circumvention tools. Our study examines TLS connections from a 10 Gbps tap at the University of Colorado Boulder, serving over 33,000 students and 7,000 faculty. We collected over 11 billion TLS connections over a 9 month period. For each connection, we generate a hash (fingerprint) [87, 127] over unchanging parts of the ClientHello message, allowing us to group connections that were made by the same implementation together. We also collect information on corresponding ServerHello messages, and anonymized SNI and destination IPs to assist further analysis.

Using our data, we find several problems across many circumvention tools we analyze, including Signal, Lantern, and Snowflake, and uncover less serious but still problematic issues with Psiphon and meek. To enable other researchers to use our dataset, we have released our data through a website, available at `https://tlsfingerprint.io`.

To address the challenge faced by existing circumvention tools, we have developed a client TLS library, **uTLS**, purpose built to provide fine-grained control over TLS handshakes. uTLS allows developers to specify arbitrary cipher suites and extensions in order to accurately mimic other popular TLS implementations. Moreover, we integrate our dataset with uTLS to allow developers to copy automatically-generated code from our website to configure uTLS to mimic popular fingerprints we have observed.

We describe and overcome several challenges in correctly mimicking implementations, and we implement multiple evasion strategies in uTLS including mimicry and randomized fingerprints, and finally

[1] Signal has since phased out domain fronting for unrelated reasons

evaluate each of these strategies using our dataset. In addition, we have worked with several existing circumvention tools to integrate our uTLS library into their systems.

In our data collection, we have made sure to collect only aggregates of potentially sensitive data to protect user privacy. We have applied for and received IRB exemption from our institution for this study, and worked closely with our institution's networking and security teams to deploy our system in a way that protects the privacy of user traffic. Our findings were disclosed responsibly to the projects and tools impacted by our results.

Our contributions are as follows:

- We collect and analyze over 11 billion TLS ClientHello messages over a 9 month period, as well as 5.9 billion TLS ServerHellos over several months. We intend to continue collecting future data.

- We analyze existing censorship circumvention projects that use or mimic TLS, finding that many are trivially identifiable in practice, and potentially at risk of being blocked by censors.

- We develop a library, **uTLS**, that allows developers to easily mimic arbitrary TLS handshakes of popular implementations, allowing censorship circumvention tools to better camouflage themselves against censors. We use our collected data to enhance uTLS, allowing **automated mimicry** of popular TLS implementations.

- We release our dataset through a website[2] , allowing researchers to browse popular TLS implementation fingerprints, search for support of ciphers, extensions, or other cryptographic parameters, and compare the TLS fingerprints generated by their own applications and devices.

We present background in Section 3.2.2, the design of our collection infrastructure in Section 3.2.3, and high level results from our dataset as it pertains to censorship in Section 3.2.4. We present our analysis and findings on circumvention tools in Section 3.2.5, and present defenses and lessons in Section 3.2.6. We go on to describe our uTLS library in Section 3.2.7, discuss future and related work in Sections 3.2.9 and 3.2.10, and finally conclude in Section 3.2.11.

[2] `https://tlsfingerprint.io`

3.2.2 Background

TLS typically takes place over a TCP connection between a client and server[3] .

After a TCP connection is established, the client and server perform a TLS handshake that allows them to authenticate identities and agree on keys, ciphers, and other cryptographic parameters to be used in the connection. The remainder of the connection is encrypted with the agreed upon methods and secrets. Figure 3.1 shows an overview of the TLS 1.2 handshake.

The first message in the TLS handshake is a ClientHello message. This message is sent in the clear, and allows the client to specify features and parameters that it supports. This includes what versions of TLS the client supports, a list of supported cipher suites and compression methods, a random nonce to protect against replay attacks, and an optional list of extensions. Extensions allow clients and servers to agree on additional features or parameters. While there are over 20 extensions specified by various TLS versions, we pay extra attention to the contents of few of them in this paper, which we use as part of the ClientHello fingerprint.

Server Name Indication (SNI) allows a client to specify the domain being requested in the ClientHello, allowing the server to send the correct certificate if multiple hosts are supported. As this is sent before the handshake, SNIs are sent unencrypted.

Supported Groups This extension specifies a list of supported mathematical groups that the client can use in key exchanges and for authentication. For example, the client can specify support for groups such as `x25519` or `secp256k1` to specify support for Curve25519 and the NIST P-256 Koblitz curve for use in ECDHE key exchanges.

Signature Algorithms Clients can specify combinations of hash and signature algorithms they support for authenticating their peers. Traditionally these have come in the form of a signature and hash algorithm pair, such as `rsa_sha256` or `ecdsa_sha512`. More recently, signature algorithms have been expanded to specify alternate padding schemes (such as RSA PSS).

Elliptic Curve Point Format specifies the encoding formats supported by the client when sending or

[3] Although TLS can happen on top of other protocols (such as UDP), for the purposes of this paper, we focus on TLS over TCP.

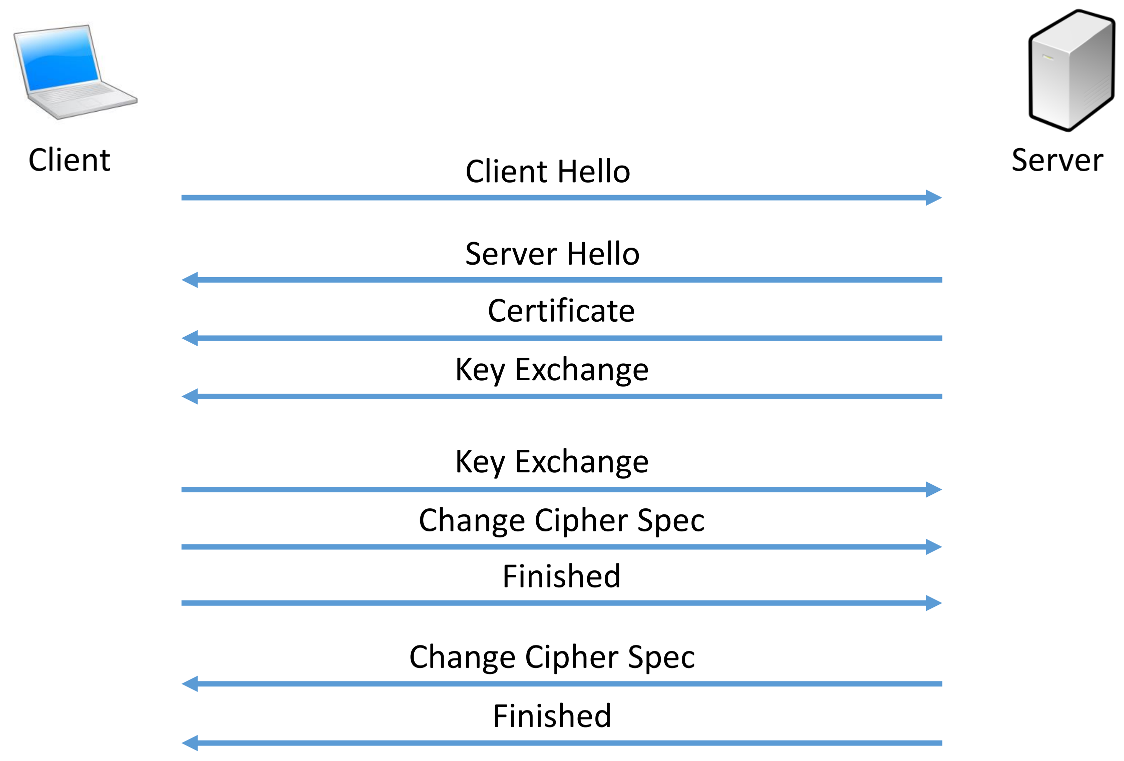


Figure 3.1: **TLS Handshake** — The TLS handshake contains several messages sent unencrypted, including the ClientHello. This allows us to fingerprint client implementations by the features and parameters they send in this initial message.

receiving elliptic curve points.

Application Layer Protocol Negotiation (ALPN) allows clients to negotiate the application protocols they support on top of TLS. For instance, a web browser could specify HTTP/1.1, HTTP2 [11], and SPDY [35]. As these are a list of arbitrary strings, unlike most other extensions, there is no standard set of possible application protocols.

GREASE (Generate Random Extensions And Sustain Extensibility) is a TLS mechanism intended to discourage middleboxes and server implementations from "rusting shut" due to ubiquitous static use of TLS extensibility points [12]. If clients only ever send a subset of TLS extension values, subpar (but still widely deployed) implementations may be tempted to hardcode those values or parse for them specifically. If this happens, later versions or implementations of TLS that attempt to include additional extensions may find that they cannot complete a TLS connection through buggy middlebox implementations. To discourage this, GREASE specifies that clients may send "random" extensions, cipher suites, and supported groups. Google has deployed GREASE in recent versions of Chrome, discouraging buggy server implementations that reject unknown extensions, cipher suites, or supported groups. Instead, such buggy implementations would be quickly discovered to not work with Google Chrome, prompting maintainers to fix the server or middlebox before it was widely deployed.

3.2.3 Measurement Architecture

Our institution's network consists of a full-duplex 10 Gbps link for the main campus network, including campus-wide WiFi traffic, lab computers, residence halls, and web services. In cooperation with our university's networking and IT support, we deployed a single 1U server with a dual-port Intel X710 10GbE SFP+ network adapter, with an Intel Xeon E5-2643 CPU and 128 GB of RAM. We received a "mirror" of the bi-directional campus traffic from an infrastructure switch. Of the packets that reach our server, we suffer a modest drop rate below 0.03%.

We used PF_RING to process packets from the NIC and load balance them across 4-cores (processes). We wrote our packet processing code in 1400 lines of Rust. We ignored packets that were not TCP port 443 or had incorrect TCP checksums, and kept an internal flow record of each new connection. Upon seeing a TCP

SYN, we recorded the 4-tuple (source/destination IP/port) in a flow record table, and waited for the first TCP packet carrying data in the connection. We attempted to parse this data as a TLS ClientHello message, which succeeded 96.7% of the time. We note that this method will miss out-of-order or fragmented ClientHello messages.

3.2.3.1 Collected Data

In our study, we collected 3 kinds of information from the network, including counts and coarse grained timestamps of unique ClientHello messages, a sample of SNI and anonymized connection-specific metadata for each unique ClientHello, and ServerHello responses. We applied for and received IRB exemption for our collection, and worked with our institution's network and IT staff to ensure protection of user privacy.

ClientHellos For successfully parsed ClientHellos, we extracted the TLS record version, handshake version, list of cipher suites, list of supported compression methods, and list of extensions. When present, we extracted data from several specific extensions, including the server name indication, elliptic curves (supported groups), EC point formats, signature algorithms, and application layer protocol negotiation (ALPN). We then formed a fingerprint ID from this data, by taking the SHA1 hash of the TLS record version, handshake version, cipher suite list, compression method list, extension list, elliptic curve list, EC point format list, signature algorithm list, and ALPN list[4] . We truncated the SHA1 hash to 64-bits to allow it to fit a native type in our database. Assuming no adversarially-generated fingerprints, the probability of any collision (given by the birthday paradox) in a dataset of up to 1,000,000 unique fingerprints is below 10^{-7}. For each fingerprint, we recorded a count of the number of connections it appeared in for each hour in a PostgreSQL database.

Connection-specific information To provide more context for identifying fingerprints, we also recorded the destination server IP, server name indication (if present), and anonymized client IP /16 network from a small sample of connections, along with the corresponding ClientHello fingerprint. This sample data helps us determine the source implementation or purpose of a particular fingerprint.

ServerHellos In addition to ClientHello messages, we also collected the corresponding server hello

[4] We specifically exclude server name from our fingerprint

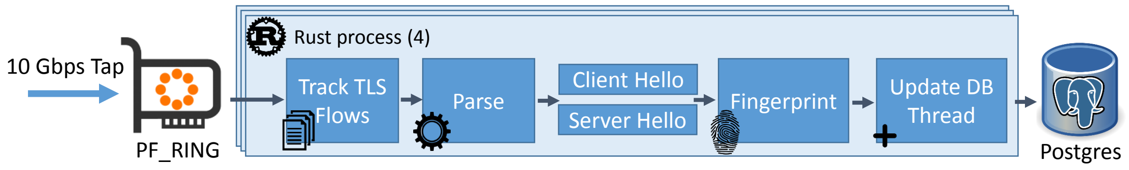


Figure 3.2: **Collection Architecture** — We implemented our 10 Gbps collection infrastructure using PF_RING and 1400 lines of Rust, utilizing 4 processes. TLS client and ServerHello fingerprints and counts were aggregated and periodically written to a PostgreSQL database in a separate thread to avoid blocking the main packet parsing loop.

in each connection, allowing us to see what cipher suite and extensions were negotiated successfully. For each ServerHello message, we parsed the TLS record version, handshake version, cipher suite, compression method, and list of extensions. We also included the data from the supported groups (elliptic curves), EC point format, and ALPN extensions.

Figure 3.2 shows a high level overview of our collection architecture.

BrowserStack In order to help link fingerprints to their implementations, we used BrowserStack—a cloud-based testing platform—to automatically conscript over 200 unique browser and platform combinations to visit our site, where we linked user agents to captured ClientHello fingerprints. Combined with normal web crawling bots and manual tool tests, our website collected over 270 unique fingerprints, with over 535 unique user agents.

3.2.3.2 Collection Details

Multiple Fingerprints Some TLS implementations generate several fingerprints. For example, Google Chrome generates at least 4 fingerprints, even from the same device. This is due to sending different combinations of extensions depending on the context and size of TLS request. Due to a server bug in the popular F5 Big IP load balancer, ClientHello messages between 256 and 511 bytes cause the device to incorrectly assume the message corresponds to an SSLv2 connection, interrupting the connection. When Google Chrome detects it would generate a ClientHello in this size range (for example by including a long TLS session ticket or server name value), it pads the ClientHello to 512 bytes using an additional `padding` TLS extension.

Browsers can also send different fingerprints from default due to end-user configurations or preferences. For example, Google Chrome users can disable cipher suites via a command line option [44].

We discuss clustering similar fingerprints in Section 3.2.4.

GREASE Because GREASE adds "random" extensions, cipher suites, and supported groups, implementations that support it would create dozens of unique fingerprints unless we normalize them. The specification provides 16 values that can be used as extension IDs, cipher suites, or supported groups, ranging from 0x0a0a to 0xfafa. While BoringSSL, used by Google Chrome, chooses these values randomly, we find their position is deterministic (e.g. first in the cipher suite list, and first and last in the extension list). We normalize these values in our dataset by replacing them with the single value 0x0a0a, which preserves the fact that an implementation supports GREASE while removing the specific random value that would otherwise generate unique fingerprints.

3.2.4 High-level results

We collected TLS ClientHello fingerprints for approximately 9 months between late October 2017, and early August 2018 (ongoing). From December 28, 2017 onward, we additionally collected SNIs, destination IPs and anonymized source IPs from a sample of traffic, and we collected ServerHello messages starting on January 24, 2018.

Overall, we successfully collected and parsed over 11 billion TLS ClientHello messages. A small fraction (about 3.3%) of TLS connections failed to produce a parseable ClientHello, most commonly due to the first data received in a connection not parsing as a TLS record or ClientHello. This can happen for packets sent out of order or TCP fragments. We also ignored packets with incorrect TCP checksums, which happened with negligible frequency (0.00013%)[5] .

Our collection suffered two major gaps, first starting on February 5 we only received partial traffic, and second between March 28 and April 19 we lost access to the tap due to a network failure. Figure 3.3 shows the number of Client Hello messages parsed over time, showing our gaps as well as the diurnal/weekend/holiday

[5] Due to a bug in the underlying `pnet` packet library, as many as 10% of packets were falsely reported to have incorrect checksums. We fixed this bug on January 25, 2018, but believe it had minimal impact on data-carrying TCP packets

pattern of traffic.

Long tail fingerprint distribution

Our 11 billion TLS ClientHellos comprised over 230,000 unique fingerprints, a surprisingly high amount if we naively assume each fingerprint corresponds to a unique implementation. Figure 3.4 shows the total number of unique fingerprints over time, rising from an initial 2,145 to 230,000 over the course of several months.

Immediately visible are large steps in the number of unique fingerprints, signifying discrete events that produced a large number of fingerprints. We discover that these events correspond to a single monthly Internet scanner that produces very few connections, but appears to be sending a significant number of random unique fingerprints. In fact, over 206,000 of these fingerprints were only seen a single time, generally from the same /16 network subnet. While this scanner had a large impact on the number of unique fingerprints, its impact on connections was negligible. In the remainder of this paper, we report on the percent of connections that a fingerprint or fingerprints comprise, effectively allowing us to highlight common fingerprints without influence from this single scanner.

Figure 3.5 shows the CDF of connections seen per fingerprint for the most popular 5,000 fingerprints (for both Client and ServerHello messages). 99.96% of all connections use one of top 5000 ClientHello fingerprints, and one of top 1310 ServerHello fingerprints.

Many of the top ten fingerprints (shown in Table 3.1) are variants generated by the same TLS implementation: for example, Google Chrome 65 generates two fingerprints (with and without the padding extension), ranked 1st and 4th over the past week in our dataset. Chrome may also optionally include ChannelID extensions depending on if the connection is made directly or via an AJAX connection.

Fingerprint clusters

As mentioned, some implementations generate multiple TLS fingerprints, due to buggy middlebox workarounds, types of TLS connection, or user preferences. This raises the question of how many fingerprints does a typical implementation produce? We compared fingerprints by performing a basic Levenshtein distance over the components we extracted in the ClientHello. For example, two fingerprints that only differ by the presence of the `padding` extension would have a Levenshtein distance of 1, while a pair of fingerprints

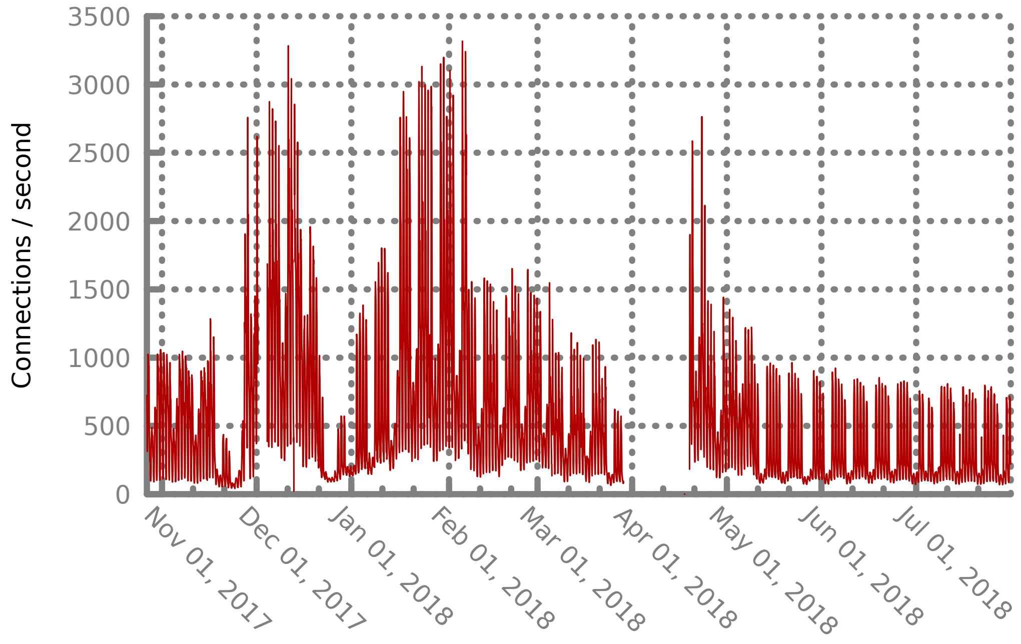


Figure 3.3: **Connections Observed** — We collected 9 months of TLS connections on our 10 Gbps campus network, observing over 11 billion connections. Noticeable is the diurnal pattern of traffic, as well as a decrease in traffic on weekends and holiday breaks.

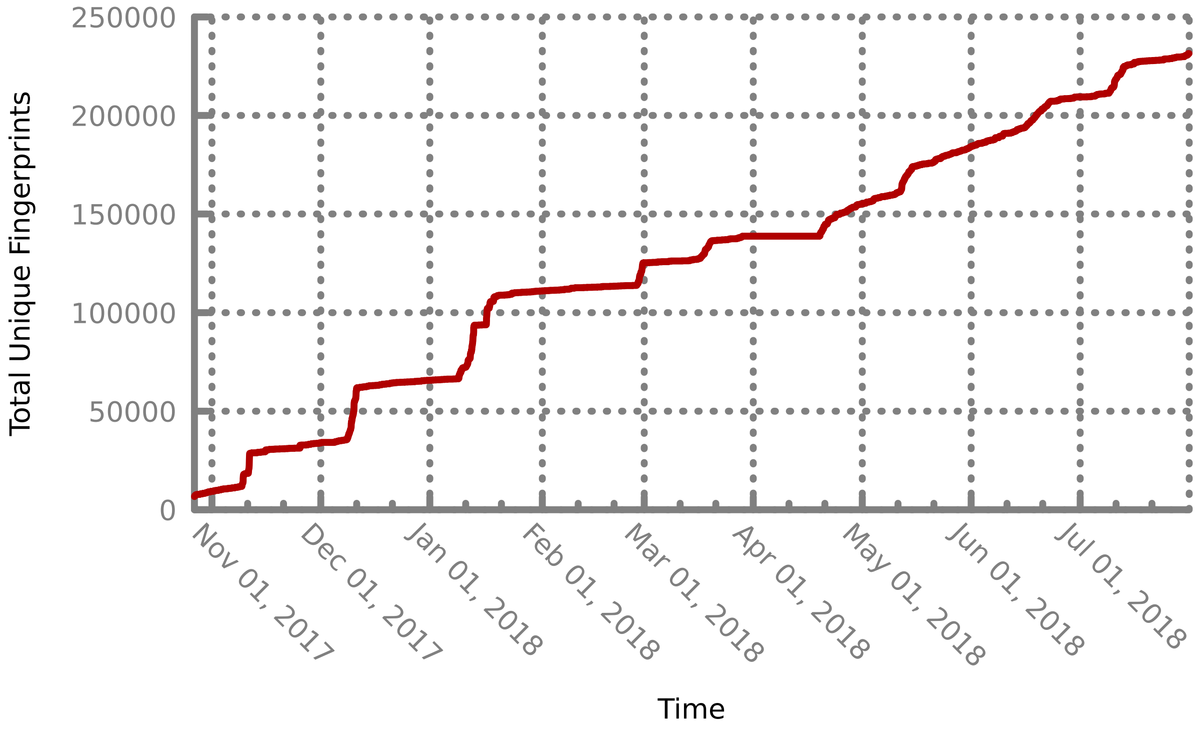


Figure 3.4: **Total Unique Fingerprints** — The number of unique TLS ClientHello fingerprints observed rises slowly but steadily over time, reaching over 152,000 by May 2018. This rise is punctuated by short but rapid increases in the number of unique fingerprints, which we determined came from a small set of Internet scanners sending seemingly random ClientHello messages.

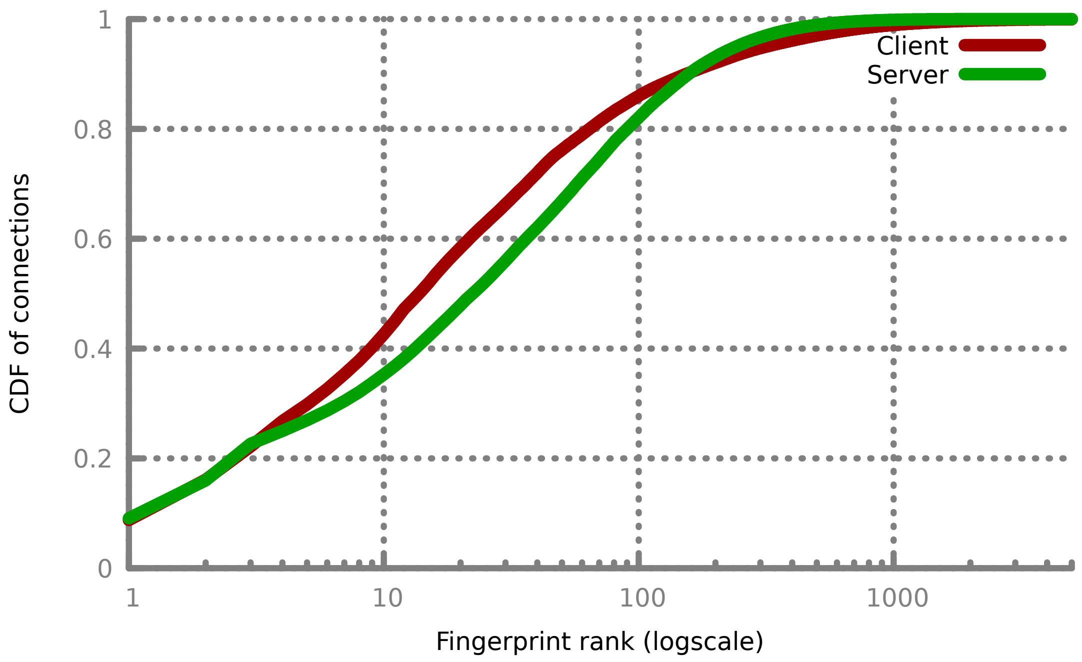


Figure 3.5: **CDF of Connections per Fingerprint** — While we observed over 152,000 ClientHello fingerprints, most connections were comprised of a small number of fingerprints: over half the connections were of one of the top 12 fingerprints, and the top 3000 fingerprints make up 99.9% of connections observed. For servers, only 4,700 fingerprints were observed, with half of the connections using one of the top 19 server fingerprints.

August 2018

Rank	Client	% Connections
1	Chrome 65-68	16.51%
2	iOS 11/macOS 10.13 Safari	5.95%
3	MS Office 2016 (including Outlook)	5.34%
4	Chrome 65-68 (with padding)	4.62%
5	Edge 15-18, IE 11	4.05%
6	Firefox 59-61 (with padding)	3.62%
7	Safari 11.1 on Mac OS X	2.82%
8	iOS 10/macOS 10.12 Safari	2.49%
9	iOS 11/macOS 10.13 Safari (with padding)	2.42%
10	Firefox 59-61	2.22%

December 2018

Rank	Client	% Connections
1	Chrome 70 (with padding)	8.49%
2	iOS 12/macOS 10.14 Safari	7.55%
3	iOS 12/macOS 10.14 Safari (without ALPN)	4.15%
4	Chrome 70	4.10%
5	iOS 12/macOS 10.14 Safari (with padding)	4.09%
6	Edge 15-18, IE 11	3.27%
7	MS Office 2016 (including Outlook)	3.01%
8	iOS 10/macOS 10.12 Safari	2.72%
9	iOS 11/macOS 10.13 Safari	2.68%
10	Chrome 71 (with padding)	2.48%

Table 3.1: **Top 10 Implementations** — The most frequently seen fingerprints in our dataset and the implementations that generate them, for a week in August and December 2018. Despite being only 4 months apart, the top 10 fingerprints changed substantially, as new browser releases quickly take the place of older versions.

that differed by a dozen cipher suites would have a distance of 12.

To determine the prevalence of multiple fingerprint variants, we generated clusters of popular fingerprints. We looked at the 6,629 fingerprints that were seen more than 1,000 times in our dataset (accounting for 99.97% of all connections), and clustered them into groups if they were within a Levenshtein distance of 5 from another fingerprint in the group. This clustering resulted in 1,625 groups, with the largest group having 338 fingerprints in it, corresponding to variants of Microsoft Exchange across several versions. Google Chrome 65 appeared in a cluster with 117 fingerprints, containing two weakly-connected sub-clusters. Half of this cluster represents fingerprints corresponding to an early TLS 1.3 draft, and the other half the current standard. Unsurprisingly, these fingerprints are long-tailed, with the top 10 fingerprints in the group responsible for 96% of the connections from the cluster.

Fingerprint churn

To measure how quickly fingerprints change and how this would impact a censor, we developed a simple heuristic. We compile a list of all fingerprints seen at least 10 times in the first week, and in subsequent weeks, compare fingerprints that were seen a substantial amount (10,000) times. This models the rate at which new fingerprints (with non-negligible use) are observed, and for a censor that employs a whitelist approach, describes the fraction of connections they would inadvertently block if they did not update their whitelist from an initial snapshot.

Figure 3.6 shows the increase in both fingerprints and connections blocked over time as new fingerprints are observed compared to an initial whitelist composed on the first week. In the steady state prior to March, the weekly average increase in blocked connections is approximately 0.03% (0.33% by fingerprints), suggesting that the rate of new fingerprints is steady but small. However, in March, both Google Chrome and iOS deployed updates that included new support for TLS features, creating a substantial increase in connections using new fingerprints. As a result, a non-adaptive whitelist censor would end up blocking over half of all connections after just 6 months.

Such large events could present a difficult situation for a whitelist censor, as new versions would be blocked until new rules were added. We conclude that whitelisting TLS implementations for a censor may be feasible, but requires a potentially expensive upkeep effort to avoid large collateral damage.

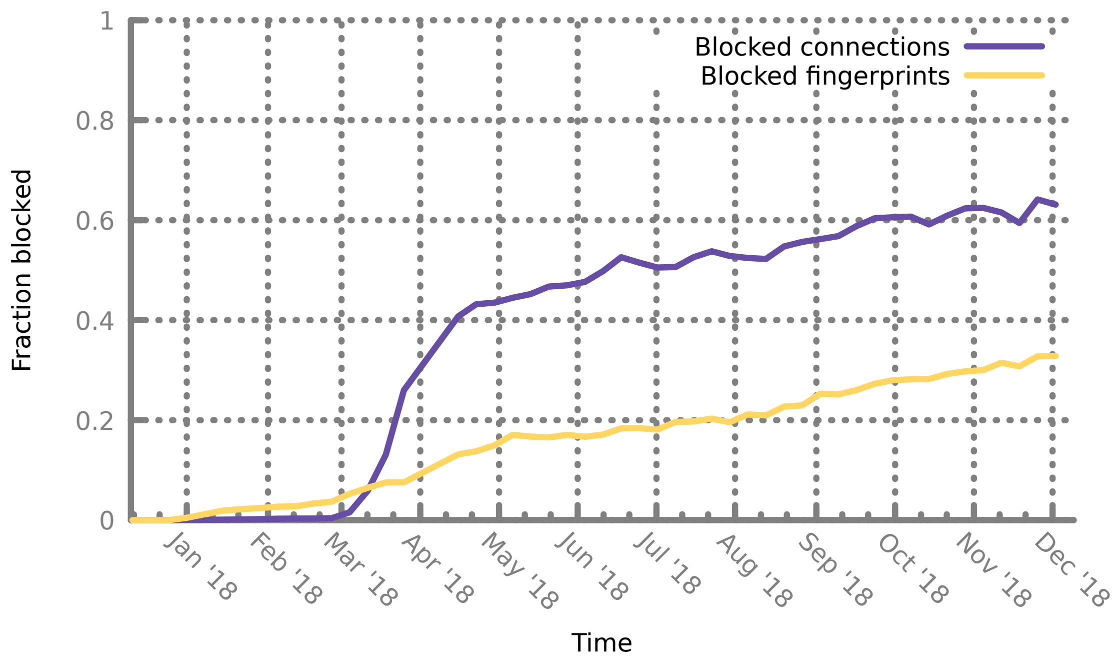


Figure 3.6: **Fingerprint Turnover** — Shows fraction of connections/fingerprints not seen during the first week. This roughly models the fraction that censor would overblock, if they took a static fingerprint snapshot and whitelisted it.

SNI

Subject Name Indication is a widely used TLS extension that lets the client specify the hostname they are accessing. Because it is sent in the clear in the ClientHello message, SNIs can be another feature that censors use to block. Many tools have different strategies for setting the SNI. Some use domain fronting, and set the SNI to a popular service inside the cloud provider (e.g. maps.google.com), while others choose to omit the SNI for ease of implementation or compatibility reasons.

As of August 2018, we observe only 1.41% of connections do not send the SNI extension, indicating that the circumvention strategy of omitting SNIs may potentially stand out and be easy to block. Indeed, the most popular SNI-less fingerprint in our dataset (accounting for 0.20% of all connections) sends a very unique cipher suite list not seen in any other fingerprints. This result impacts many circumvention tools, including Psiphon and Lantern that both produce ClientHello messages that do not include the SNI extension, suggesting that these connections may be easy for censors to block.

3.2.5 Censorship Circumvention Tools

Many censorship circumvention tools use TLS as an outer layer protocol. For example, domain fronting uses TLS connections to popular CDN providers with cleartext SNIs suggesting the user is connecting to popular domains hosted there [54]. Inside the TLS connection, the user specifies a proxy endpoint located at the CDN as the HTTP Host, relying on the CDN's TLS terminator and load balancer to route their traffic to the intended destination. As censors only see the unencrypted SNI, they cannot distinguish domain fronting connections from benign web traffic to these CDNs. Psiphon, meek and Signal use this technique to conceal their traffic from would-be censors [54, 119, 145]. However, mimicking connections in this manner is difficult in practice: any deviations from the behavior of true browsers that typically access these CDNs allows a censor to detect and block this style of proxy [71].

Other tools such as Psiphon and Lantern use TLS in a more natural way, connecting directly to endpoint proxies. These tools must also take care to mimic or otherwise hide their connections to look normal or face blocking.

Two different versions of meek were found to be detectable by Cyberoam [49] and Fortiguard [50] firewalls, which fingerprinted the ClientHello of meek and blocked it. At the time of incident with Cyberoam, meek was mimicking Firefox 38, which had over time dwindled in users, reducing overall collateral damage from blocking meek. Notably, censors were able to substantially reduce collateral damage by combining TLS fingerprinting with simple host blocking: blocking was limited to clients that both looked like Firefox 38 and tried to access specific domains used for fronting. Subsequently, meek started mimicking the newer Firefox 45, but was ultimately blocked by Fortiguard in the same manner when Firefox 45 became less popular.

Given the importance of ClientHello messages in identifying and potentially blocking censorship circumvention tools, we analyzed the fingerprints generated by several popular circumvention tools, and compared the relative popularity of those fingerprints in our dataset. Fingerprints that were seen much more frequently should in theory be more difficult for censors to block outright without collateral damage, while those that we rarely see in our dataset may be at risk of easy blocking. In this analysis, we assume that our dataset contains negligible traffic from these tools, as our institution is not in a censored region and users have little motivation to use them en mass.

We also provide a mechanism to easily test and analyze any application by submitting a pcap file to our website: `https://tlsfingerprint.io/pcap`. Our website will list all the fingerprints extracted from the pcap file, and provide links with more details about their features, popularity, any user agents observed using these fingerprints, and similar fingerprints that can be compared.

3.2.5.1 Signal

Until recently, the Signal secure messaging application used domain fronting to circumvent censorship in Egypt, the United Arab Emirates, Oman, and Qatar [102]. Signal used Google App Engine as a front domain until April 2018, when Google disabled domain fronting on their infrastructure [101]. Signal switched to domain fronting via the Amazon-owned souq.com, but shortly after, Amazon disallowed domain fronting on their infrastructure, and notified Signal that it was in violation of its terms of service [98, 101]. Signal has since stopped using domain fronting, and direct access is now blocked in the above countries.

However, Signal still serves as one of the largest deployments of domain fronting, and we analyze both when it used Google and when it used Amazon.

When Signal was using Google for domain fronting, we analyzed both the iOS and Android versions of the application on real devices and collected the TLS fingerprints it generated when we signed up with phone numbers in the previously mentioned censored countries, triggering its domain fronting logic. For iOS, we found it generates the native iOS fingerprint, which appears in 2.14% of connections, making it the 11th most popular in our dataset at the time. It is unlikely a censor would be able to block Signal iOS from its ClientHello fingerprints alone.

However, on Android, the situation is drastically worse. Even on the same device, Signal Android generates up to four unique fingerprints when using domain fronting. Some of these fingerprints were never seen in our dataset, making it trivial for a censor to detect and block. Even the most popular fingerprint was seen in only 0.06% of connections, making it ranked 130th in popularity. It appears to be used by a small fraction of Android 7.0 clients that access googleapis.com. We confirmed these findings using two devices: a Google Pixel running Android 7.1 with Signal version 4.12.3, and a Samsung G900V running Android 6.0.1 with Signal version 4.11.5.

Signal on Android uses the okhttp[6] library to create TLS connections with three different "connection specs" that define the cipher suites and TLS version to be used in the ClientHello. Signal attempted to mimic 3 different clients for 3 fronts it used: Google Maps, Mail and Play. However, we identify two problems: First, the okhttp library disables certain cipher suites by default such as DHE and RC4 ciphers, which are specified in Signal's connection specs [21]. Although the library supports them, okhttp disallows their use without an explicit API call, which Signal did not use. Instead, the okhttp library silently drops these ciphers from the ClientHello it sends, causing it to diverge from the intended connection spec.

Second, even when the desired cipher suite list is unaltered, these cipher suites still correspond to unpopular clients, due to differences in other parts of ClientHello. For example, the `GMAIL_CONNECTION_SPEC`'s cipher suite list is most commonly sent by Android 6.0.1, and is seen in only 0.17% of connections. However, the Signal ClientHello is trivially distinguishable from the Android 6.0.1 fingerprint, as Signal does not specify support for HTTP/2 in its ALPN extension, while Android does.

[6] https://square.github.io/okhttp/

By April 2018, Signal switched to using the Amazon-owned souq.com, and changed the TLS configuration to only use a single connection spec. This configuration still included 3 DHE cipher suites, which are ignored and not sent by the okhttp library. We also noticed that despite only the single spec, Signal generates two distinct TLS fingerprints, differing in which signature algorithms they list. Each of these fingerprints is seen fewer than 100 times (<0.000001% of connections).

As of May 2018, Signal no longer uses domain fronting, making these issues moot. Nevertheless, these examples illustrate several challenges in mimicking popular TLS implementations. First, libraries are not currently purpose-built for the task of mimicry, emphasizing security over an application's choice of cipher suites. Second, in addition to cipher suites, an application also must specify extensions and their values in order to properly mimic other implementations.

3.2.5.2 meek

meek is a domain fronting-based pluggable transport used by Tor to circumvent censorship [34, 37, 54]. meek tunnels through the Firefox ESR (Extended Support Release) version that is bundled with the Tor Browser. Unfortunately, the ClientHello of Firefox and Firefox ESR may diverge, which eventually makes the fingerprint of ESR versions relatively uncommon, allowing meek to be blocked with smaller collateral damage. As of August 2018, meek uses Firefox 52 ESR whose corresponding ClientHello is the 42nd most popular fingerprint in our dataset, seen in approximately 0.50% of connections. The majority of regular Firefox users have migrated to Firefox versions 59+, whose most popular fingerprint is ranked 6th in our dataset seen in 3.64% of connections.

Thus, meek is using a version of Firefox that is once again approaching obsolescence, potentially allowing censors to block it without blocking many normal users. Unfortunately, meek must wait for updates to the underlying Tor Browser to receive updated TLS features, making it relatively inflexible in its current design.

3.2.5.3 Snowflake

Snowflake [36] is a pluggable transport for Tor that relies on a large number of short-lived volunteer proxies, similar to Flashproxy [53]. Clients connect to a broker via domain fronting, and request information about volunteers running browser proxies, and then connect to and uses those proxies over the WebRTC protocol.

Snowflake is under active development, and its authors were aware of potential TLS fingerprintability issues. Indeed, we find that Snowflake (built from git master branch on April 17, 2018) generates a fingerprint that is close to, but not exactly the same as the default Golang TLS fingerprint. In particular, it diverges by including the NPN and ALPN extensions, and offers a different set of signature algorithms. As a result, this fingerprint is seen in fewer than 0.0008% of connections, making it susceptible to blocking.

3.2.5.4 Outline

Outline is a private self-hosted proxy based on Shadowsocks, a randomized protocol that attempts to look like nothing. Outline provides a GUI interface that guides the user through the Shadowsocks setup process, including purchase of a VM on DigitalOcean. During the purchase, the installation script uses TLS, leveraging the Electron framework (based on Chromium). As of our tests in May 2018, we find that Outline's TLS fingerprint matches that of Chrome version 58-64.

While this fingerprint is decreasing in popularity with the release of Chrome 65, it still remains common: We observed it in 2.10% connections in the first week of May (vs. 8.76% of connections over our full dataset). As of August 2018, the weekly rate of connections with this fingerprint has fallen to 1.72%. Still, the Outline installation process is unlikely to be blocked based on the generated TLS fingerprint in the short term, though it must take care to update before use of this fingerprint wanes further.

We did not evaluate the rest of the communication protocols used by Outline after installation, as it does not use TLS.

3.2.5.5 TapDance

TapDance [164] uses refraction networking to circumvent censors, placing proxies at Internet service providers (ISPs) outside censoring countries, and having clients communicate with them by establishing a TLS connection to a reachable site that passes the ISP. As it connects to innocuous websites, the TapDance client must make its TLS connection appear normal to the censor while it covertly communicates with the on-path ISP.

As of May 2018, TapDance uses a randomized ClientHello, which protects it against straightforward blacklisting. However, the randomized fingerprints generated by TapDance are not found in our dataset of real-world fingerprints, which could allow a censor to block it by distinguishing randomized ClientHellos from typical ones, or by simply employing a whitelist of allowed TLS ClientHello messages.

3.2.5.6 Lantern

Lantern uses several protocols to circumvent censorship, mainly relying on the randomized Lampshade pluggable transport [117]. However, as of February 2018, several parts of Lantern still use TLS as a transport, allowing us (and censors) to capture its fingerprint.

We observed that Lantern uses a Golang TLS variant that sends a Session Ticket extension, and doesn't send the server name extension. This variant does appear in our dataset, however, at a very low rate: approximately 0.0003% of connections, ranked 1867 in terms of popularity.

Lantern uses the Session Ticket to communicate information covertly out-of-band to their server. However, this use makes their fingerprint differ from the default Golang TLS, illustrating that application-level demands may result in observably different handshakes, ultimately allowing a censor to block them.

3.2.5.7 Psiphon

Similar to Lantern, Psiphon also uses several circumvention transports, including domain fronting over TLS. We obtained a version of the Psiphon Android application that only connects using TLS from the app's developers, allowing us to collect TLS fingerprints generated by it. Psiphon cycles through different Chrome and Android fingerprints until it finds an unblocked one that allows them to connect to their servers. Such a diverse set of fingerprints may help evade censorship, even if most fingerprints get blocked. However, a censor with a stateful DPI capability may also be able to use this feature to detect (and ultimately block) Psiphon users.

We find Psiphon successfully mimics Chrome 58-64, generating two fingerprints ranked in the top 20 in our dataset, but is less successful at mimicking legacy Android: fingerprints supposedly targeting Android were seen in fewer than 50 connections out of the 11 billion. We also determined that Psiphon sometimes mimics Chrome without an SNI to evade SNI-based blocking, generating a blockable fingerprint seen in fewer than 0.0002% of connections.

3.2.5.8 VPNs

We analyzed OpenVPN and 3 services that advertised an ability to circumvent censorship: IVPN, NordVPN and privateinternetaccess.com. VPNs tend to use UDP by default for performance benefits, which we did not collect in our measurement system. However, we extracted the cipher suites and extensions from OpenVPN's UDP TLS handshake, and found that the combination is rare in our (TCP-based) dataset. This may suggest that OpenVPN has a distinctive fingerprint, given the unique set of features in its fingerprint (85 cipher suites, 13 groups, 14 signature algorithms, and rare set of extensions), but we cannot be sure without collecting UDP TLS connections.

We did recover one TCP TLS fingerprint from NordVPN, which is a circumvention plugin in Google Chrome. However, this plugin uses the API of the host browser to make TLS requests, making it indistinguishable from the version of Google Chrome it is installed in.

3.2.5.9 Vendor Response

We notified the authors of the censorship circumvention tools about respective fingerprintability issues, and provided additional data about fingerprints as well as potential defenses.

In response to our disclosure, developers of Psiphon, Lantern, and TapDance integrated our uTLS library described in Section 3.2.7 to take advantage of alternative mimicry options and have greater control

Tool	Version/Date	Rank [all time]	% Connections
Psiphon	Jan 2018	1	8.76%
		9	2.42%
		62	0.25%
		198	0.04%
		203	0.04%
		500	0.01%
		2190	0.0002%
		14397	0.0000%
		16814	0.0000%
Outline	May 2018	1	8.76%
meek	TBB 7.5	42	0.50%
Snowflake	April 2018	1378	0.0008%
Lantern	4.6.13	1867	0.0003%
TapDance	May 2018	random	-
Signal	4.19.3	11468	0.0000%
		12982	0.0000%

Table 3.2: **Tool Fingerprintability** — Summary of all TLS fingerprints generated by censorship circumvention tools and their rank and percentage of connections seen in our dataset as of early August 2018. Highlighted in red are fingerprints seen in relatively few ($< 0.1\%$) connections, putting them at risk of blocking by a censor.

over TLS features. The Snowflake and meek authors were aware that the current fingerprint was not ideal, but didn't have immediate plans to fix it. Snowflake is still in active development; and meek will keep tunneling through Firefox ESR in short term, due to the effort involved in changing it, but is considering going back to mimicking, rather than tunneling. We disclosed our findings to Signal via email and a GitHub issue prior to their removal of domain fronting, but did not receive any response besides the GitHub issue being silently deleted.

3.2.6 Defenses & Lessons

While the TLS protocol provides plenty of cover traffic for circumvention tools, there are many challenging details that tools must get right in order to successfully evade censorship. There are two high-level detection evasion strategies that circumvention tools employ when choosing protocols [149]: first, they may try to **mimic** one or more existing implementations, making it difficult to distinguish them and increasing the collateral damage to blocking connections that look like the tool. Second, tools may try to generate **random** protocol features, to prevent censors from being able to identify and block the tool with a blacklist approach. In this section, we investigate how these techniques can be applied to TLS implementations in censorship circumvention tools.

Mimicking Clients that choose to mimic other TLS implementations face several challenges. First, tools must **identify a popular implementation** to mimic, which is typically done by choosing a popular web browser. However, it may also be done by choosing a large number of individually less-popular but collectively popular implementations, and mimicking among them. Second, the tool must **support** the cipher suites, extensions, and features present in the popular implementation(s). For instance, if a tool mimics Chrome and sends but does not actually support a ChaCha20 cipher suite, the server may select that cipher suite for the connection, causing the tool to abort the connection. Not only does this cause compatibility issues, it gives an observant censor a way to identify users of a tool.

We note that this problem can be partially mitigated if the server is controlled by the tool maintainer, as they can choose to select only cipher suites and extensions that they know their tool to support. However, this is not the case in tools that do not control both endpoints, such as domain fronting, refraction networking[7] , and tools that generate cover traffic to other servers. In addition, intricacies of the circumvention protocol may limit the features that a tool can use, even if implemented. For example, some domain fronting tools cannot send the SNI extension to certain CDNs.

Finally, the tool must **maintain** support, as the popular implementations they mimic change over time, as do the features they support due to automated patches and updates. For example, although meek has been successful at mimicking multiple versions of Firefox, it has lagged behind the Sisyphean task of keeping pace with updates to the Firefox TLS implementation.

Randomizing Tools that randomize their generated TLS ClientHello messages have the advantage that they do not have to identify or track support for popular implementations. However, this strategy can only work if there is a sufficient number of similar-looking connections that prevent the censor from distinguishing it. For instance, Figure 3.6 demonstrates the rate at which new connections would hamper a censor's ability to use a whitelist. Censors could also distinguish randomized ClientHello messages by capturing distributions or other heuristics that are not properly mimicked by a tool's randomized fingerprint. For example, if the tool naively picks from a set of supported extensions, a censor may notice that no other implementation supports a particular pair of extensions simultaneously. When the circumvention tool randomly selects both extensions in this pair, the censor can identify and block the user. Thus, the random fingerprint strategy must carefully mimic the **distribution** of the global TLS implementation population.

We note that all tools must implement these features either by creating their own libraries or using existing ones that are generally ill-suited to the task of fine-grained control over the TLS handshake. This challenge is illustrated by Signal's use of the okhttp library, which silently removed cipher suites that Signal specified. Other TLS libraries may ignore supported cipher suite order, making it difficult for applications to produce specific TLS handshakes. In the following section, we describe our library that is purpose-built for providing applications control over their TLS connection.

[7] formerly decoy routing

3.2.7 uTLS

TLS fingerprinting remains a looming threat for anti-censorship tools, and as we have shown, even tools that attempt to defend against it can often fall short. Indeed, mimicking is hard to get right: there are lots of features to keep track of and implement, the mimicked fingerprint could rapidly go obsolete, or the tool's underlying library could silently change the fingerprint.

There may also be unexpected or complicated dependencies that prevents simply parroting ClientHello messages seen. For example, GREASE values generated by Google Chrome used to be deterministic and depend on the value of the ClientRandom, but this was changed in favor of random values. In addition, cipher suites can influence other parts of the header (or server response), such as by having a special meaning (SCSV cipher suites), by defining what TLS version is used, or triggering the inclusion of an extension. Finally, extensions may affect each other, for example, the presence and size of padding extension can depend on the size of the ClientHello. An implementation that failed to mimic these subtleties could be identified by a censor.

To assist censorship circumvention tools, we created a TLS library[8] that aims to protect against TLS fingerprinting and (among other features) allows developers to easily mimic arbitrary ClientHello messages. We develop our library as a fork of Golang's native TLS library `crypto/tls`, adding over 2,200 new lines of code.

As of August 2020, many circumvention tools have adopted the uTLS library, including Psiphon, Lantern, meek, and TapDance, who use uTLS to allow them greater control over TLS features and make it easier to mimic popular implementations.

3.2.7.1 Design

uTLS is designed to be an addition to the standard `crypto/tls` library and minimizes changes to core Go files, enabling us to easily auto-merge from upstream. This allows us to keep uTLS up-to-date with the underlying standard library, and adopt any new features and bug fixes that come in the future. In addition, this allows us to rely on the performance and security of `crypto/tls`: uTLS simply fills the ClientHello and

[8] `https://github.com/refraction-networking/utls`

leaves execution of the TLS handshake up to the standard functionality.

Our choice of Golang as the language for uTLS is motivated by several reasons. First, it is a popular language used in several censorship circumvention tools, including Lantern, meek, Psiphon, Snowflake, and TapDance, allowing easier integration. For tools that are not written in Go, integration should still be possible via Go's language bindings [88]. Second, Golang is memory safe (bounds checked), decreasing our worry of introducing control flow vulnerabilities into tools that integrate uTLS, despite containing network serialization code.

Low-level access to the handshake uTLS provides write access to any fields of the ClientHello message, allowing implementations to specify their own cipher suites, compression methods, client random, extensions, etc. Developers can compose a ClientHello using uTLS structures, or by manually specifying the bytes of a raw ClientHello for uTLS to use. In addition, uTLS provides structured read access to handshake state, including the ServerHello, Master Secret, and key stream.

Mimicry Users can also select from a set of preset built-in ClientHello messages. As of August 2018, uTLS includes defaults for Chrome 64, Firefox 58, and iOS 11, and we plan to add support for new versions of browsers, operating systems, and other popular devices. Mimicking could be hard to get right, but we verified uTLS' ability to mimic popular clients by comparing the fingerprints it generated to those in our dataset.

We note that when uTLS mimics a ClientHello of another implementation, it may potentially advertise support for features it does not actually implement. For instance, it may send a cipher suite that, if selected, uTLS will be unable to use. We measure this risk in Section 3.2.7.2.

Randomized fingerprints Given the long tail distribution of ClientHello fingerprints in our dataset, uTLS also supports generating random fingerprints. Although these are unlikely to correspond to popular implementations and fingerprints seen, censors may have a hard time constructing a comprehensive whitelist of TLS fingerprints, making it difficult to block random ones. Similar techniques have been used by other randomized protocols for censorship circumvention, such as obfsproxy [29] and ScrambleSuit [162], which attempt to look like no protocol at all. Our random fingerprints extends this idea at the TLS layer, ensuring that packets are valid TLS messages, but making it difficult for censors to blacklist the specific implementation.

Using multiple fingerprints Mimicking multiple fingerprints makes it possible for a circumvention tool to operate even when a subset of its fingerprints are blocked. To support this, uTLS can optionally cycle through a popular set of fingerprints in its handshakes until an unblocked working one is found. Thus, if uTLS is able to properly mimic even one implementation, it will be more difficult for a censor to block this strategy. uTLS automatically retries the latest working fingerprint when reconnecting to minimize unnecessary changes to its fingerprint. An example usage may be found in Appendix, Listing 3.1.

Automatic code generation While uTLS makes it easy to manually specify parts of the ClientHello, we provide an additional feature to make this even easier. Our website produces automatically-generated code for each fingerprint in our dataset, allowing developers to simply copy and paste to configure uTLS to mimic a given fingerprint. This feature allows developers to easily keep their tools up-to-date, and could even allow this to potentially be fully-automated: continuous integration scripts could watch for changes in fingerprint popularity, and either alert developers or possibly automatically pull in new code to use more recent fingerprints.

We note that automated fingerprint code may generate ClientHellos that uTLS does not yet fully support, potentially causing the handshake to fail if the server supports those features. We explicitly identify and warn the developer when this is the case, as our automated code tracks the features uTLS supports.

Fake Session Tickets uTLS provides the ability to send arbitrary session tickets, including fake ones. This is useful when the tool developer also controls the server, which must accept the fake session ticket or generate a real one for further out-of-band distribution. This technique allows servers to save a round trip time and avoid sending a Certificate or Key Exchange message, giving the censor fewer messages and information to block on. To support mimicking, we track commonly used Session Ticket sizes on our website[9] .

3.2.7.2 Measuring feature support

To enable mimicking other TLS implementations, uTLS allows the developer to advertise support for TLS extensions and cipher suites that are not actually supported by our library. If the server does not select or use these cipher suites or extensions, the connection will function normally. However, if the server selects a

[9] `https://tlsfingerprint.io/session-tickets`

cipher suite that is not implemented by uTLS, the connection will visibly break. These risks do not impact tools that make connections to servers under the developers' control, as those unsupported features can be easily disabled server side.

We measure how many fingerprints in our dataset uTLS can support without this risk. We classify a fingerprint as "fully supported" if uTLS is able to handle every feature in it (e.g. the server could select any cipher suite, curve, or extension, and the connection would succeed). We classify a second group of "optionally supported" fingerprints that are also supported by uTLS but may include weaker ciphers, such as such as `TLS_RSA_WITH_AES_256_CBC_SHA256`, that were disabled by the underlying Golang library. uTLS users may choose whether to enable those weaker ciphers, or to let the connections fail if the weaker ciphers are chosen by a server.

Note that a fingerprint not being fully supported doesn't always lead to unsupported feature getting chosen by server: it might be a low-priority, or not supported by the server either. Weaker ciphers are only likely to be picked in the wild if the client communicates with an outdated server.

As of August 2018, uTLS fully supports 21,940 fingerprints (9.3%), which were seen in 5.9% of connections collectively. The top ranked fully supported fingerprint is the 9th (all-time) most popular fingerprint in our dataset, which is a fingerprint generated by Chrome 61–64. If weaker "optionally supported" CBC ciphers are allowed, then uTLS supports 22,616 fingerprints (9.6%), which were seen in 37.3% of collective connections. This includes 30 fingerprints in the top 100, including the 3rd most popular fingerprint (generated by Outlook 2016). As mentioned, uTLS code for using all of these fingerprints can be automatically generated by our website, requiring minimal effort from the developer.

We also use our data to learn which additional features would give uTLS the most additional coverage in terms of supported fingerprints, to know which features we might want to focus on adding to the library next. As of August 2018, supporting the ChannelID extension alone will allow us to fully support 245 more fingerprints which would account for an additional 20% of connections seen. We note that these fingerprints could be mimicked in uTLS now, as the Channel ID is unlikely to be supported by servers: only 2.9% of ServerHello messages used the extension. This extension aims to secure connections by cryptographically authenticating client to the server, allowing to bind cookies and tokens to particular client's channel, which has to be explicitly implemented and integrated with the application layer on the server.

3.2.7.3 TLS 1.3

As of March 2018, TLS 1.3 [124] has been standardized and is being rolled out in major browsers. TLS 1.3 offers several advantages over previous versions, including decreased network round trips in new connections (improving performance), and encrypted handshakes (improving privacy).

Our motivation to support TLS 1.3 in uTLS is twofold: first, several features such as encrypted certificates and encrypted SNI[10] (ESNI) may prove useful for evading censors [48]. Second, as popular browsers begin to implement and send TLS 1.3 handshakes, circumvention tools will soon want to mimic them to continue blending in with popular implementations.

Interestingly, TLS 1.3 ClientHellos look similar to those in TLS 1.2: in fact, TLS 1.3 still sends a handshake version corresponding to TLS 1.2 to allow implementations to work in the presence of buggy middleboxes and servers that cannot handle other values. TLS 1.3 instead adds functionality via several new extensions in the Client and ServerHello messages.

Although our fingerprints already include the presence and order of all extensions in a given ClientHello, we only parse and include a handful of extensions' data in our fingerprint. This means if many implementations send the same set of extensions but include different data, we would mistakenly classify them as the same fingerprint. This is particularly a concern for TLS 1.3 handshakes, which heavily rely on new data-carrying extensions. To address this, we reviewed the new extensions in TLS 1.3, and added the body data of popular extensions that do not change per-connection. For example, `supported_versions` contains a list of supported TLS versions ordered by preference. As different implementations can announce different versions they support, we add this data to our fingerprint. So far, we have observed 40 distinct values of this extension announcing support for the various TLS versions and drafts. Table 3.3 shows the most popular extensions we have collected as of December 2018, and highlights the ones whose data we include in our fingerprint.

As uTLS is built on Golang's crypto/tls library, we were able to merge its TLS 1.3 support into uTLS, allowing us to mimic Firefox 63 and Chrome 70, which both send TLS 1.3 handshakes. With some additional

[10] Not yet standardized

Extension	Conns	Extension	Conns
supported_groups	99.4%	GREASE	30.2%
server_name	99.3%	**psk_key_exchange_modes***	28.7%
signature_algorithms	97.8%	**supported_versions***	28.7%
ec_point_formats	96.9%	**key_share***	28.6%
extended_master_secret	86.8%	NPN	27.3%
status_request	85.7%	**compressed_certificate***	24.8%
renegotiation_info	81.9%	ChannelID	20.5%
ALPN	71.9%	heartbeat	5.0%
signed_certificate_timestamp	66.9%	token_binding	3.9%
SessionTicket	56.0%	pre_shared_key*	3.1%
padding	32.3%	**record_size_limit**	2.5%

Table 3.3: **Top Extensions** — While we include the presence and order of all extensions in our fingerprint, **Bold** denotes extensions whose data we additionally parse and include in our fingerprint; * marks extensions new in TLS 1.3.

implementation work to support new extensions, we expect to be able to fully support over 8% of all TLS connections automatically (up from 5% currently), and optionally support over 37% if we enable weak ciphers.

3.2.8 Other Results

In this section, we present on other findings from our TLS dataset that are relevant to censorship circumvention tools.

3.2.8.1 ServerHello Analysis

As of August 2018, we collected approximately 5,400 unique ServerHello fingerprints, substantially fewer than the number of unique ClientHello fingerprints. This is in part due to ServerHello messages having less content, as it specifies only a single cipher suite and compression method, rather than the full list that the server supports. On the other hand, while a client implementation might generate a single or small collection of Client Hello fingerprint, servers can potentially generate distinct fingerprints in response to different ClientHello messages. For example, the single most popular external IP address (corresponding to Google) sent 199 unique server fingerprints from 1,494 ClientHello fingerprints sent to it. Looking at the responses to only the most popular ClientHello message, there are 750 different ServerHello fingerprints, suggesting that the actual number of distinct TLS server implementations and configurations that these clients talk to may be close to this value.

Selected Ciphers

Using our collected information on ServerHello messages, we can compare the set of offered cipher suites by clients, and discover what cipher suites are actually selected and used in practice by servers. This is useful for circumvention tools as it provides evidence of many unselected cipher suites that clients can offer without having to actually support.

Excluding the long tail of fingerprints seen only once, in our ClientHello fingerprints, there were over 7900 unique sets of unique cipher suites. These sets enumerate 522 cipher suite values, which is greater than the number of standardized cipher suites, for reasons described in Section 3.2.8.2.

Analyzing the unique cipher suites that are selected by servers, however, we find just 70 cipher suites ever selected, with the top 10 accounting for over 93% of all connections. Interestingly, the most popular cipher suite across all ClientHellos (`TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA`) is only selected in approximately 1% of connections. This shows there are many cipher suites that servers will rarely or never choose, and circumvention tools are free to present them in their ClientHello messages without having to support those cipher suites.

3.2.8.2 Non-standard parameters

Our collection tool ignores malformed ClientHellos that cannot be parsed, but even well-formed ClientHello messages may still contain invalid parameters. For example, of the 65,536 possible values for 2-byte cipher suites, only 338 values are recognized and standardized by the Internet Assigned Numbers Authority (IANA) [125], with the remainder either unassigned or reserved for private use. Similarly, only 28 values are defined for the 2-byte extension field. We note that TLS 1.3 proposes values for an additional 5 cipher suites and 10 extensions, which we include in our analysis.

This allows us to locate TLS ClientHello fingerprints that specify non-standard extensions or cipher suites. In total, over 138,000 fingerprints accounting for 13.14% of connections contained at least one non-standardized cipher suite or extension value. The commonality of support for non-standard features suggests it may be difficult for a censor to fully understand or whitelist all commonly-used fingerprints, as many do not strictly conform to the standard. Table 3.4 shows the breakdown of non-standard parameters.

3.2.8.3 Weak Ciphers

We observe a small fraction of clients continue to offer weak or known-broken ciphers, including DES [65, 78], Triple-DES (3DES) [16], and RC4 [3, 152]. More concerning, we still see clients supporting export-grade encryption, which negotiates intentionally weakened keys and has been recently found to enable modern vulnerabilities [8, 15].

TLS can also employ hash functions with known collisions, such as MD5 [140, 143] and SHA1 [144, 157]. While collisions may not enable attacks when used in the HMAC construction employed by TLS

	Fingerprints	% Connections
TLS 1.3 draft ciphers	1002	10.626%
Legacy ciphers	82992	1.392%
GOST ciphers	95548	0.051%
Outdated SSL ciphers	106439	0.097%
Unknown ciphers	137999	0.039%
Total non-standard ciphers	**143060**	**12.106**%
TLS 1.3 draft extensions	715	10.626%
Legacy Extensions	441	0.154%
Extended Random	340	1.445%
Unknown extensions	367	0.899%
Total non-standard extensions	**1404**	**11.677**%

Table 3.4: **Non-Standard Parameters** — Breakdown of the number of unique ClientHellos (fingerprints) and the share of connections they appear in that send non-standard cipher suites or extensions. While TLS 1.3 draft ciphers and extensions are the majority, we still find unknown ciphers and extensions in use.

	Fingerprints	% Connections
DES	191459	0.90%
3DES	236859	67.0%
EXPORT	194418	0.66%
RC4	223900	8.19%
MD5 (Cipher)	200608	7.15%
MD5 (Sigalg)	4385	0.74%
SHA1 (Sigalg)	114615	97.6%
TLS_FALLBACK	787	0.03%

Table 3.5: **Weak Ciphers** — We analyzed the percentage of connections offering known weak cipher suites. We also include `TLS_FALLBACK_SCSV`, which indicates a client that is falling back to an earlier version than it supports due to the server not supporting the same version.

cipher suites, they can introduce problems when used in signature algorithms, as collisions there can allow an attacker to forge CA permissions [140]. This means that MD5 and SHA1 may not be problematic as cipher suites, but are when offered as a signature algorithm. We present both uses for completeness.

Clients can also signal that they have fallen back to a lower version of TLS by sending the spurious cipher suite `TLS_FALLBACK_SCSV` [107]. While its presence does not indicate a weakness in a client, it does indicate a suboptimal mismatch between client and server versions.

Table 3.5 summarizes the fingerprints and percent of connections we observed clients offering these weak cipher suites and signature algorithms. While circumvention tools would likely avoid using such weak cipher suites (lest it enable a censor to successfully break their TLS connections), this further demonstrates the wide range of TLS implementations present on the modern Internet, once again showing how long legacy code can remain in use.

3.2.9 Related Work

3.2.9.1 Passive TLS Measurements

Several studies have measured TLS (and SSL) by passively observing traffic as we have in our study. However, the vast majority of these studies focus mainly on certificates and the Certificate Authority ecosystem. For example, in 2011 Holz et al. analyzed 250 million TLS/SSL connections and extracted certificates in order to study the existing landscape of certificate validation [70]. In addition to uncovering the "sorry state" of the X.509 certificate PKI, they briefly analyzed selected cipher suites, finding that RC4-128, AES-128, and AES-256 were the most popular cipher suites used at the time, with TLS_RSA_WITH_RC4_128_MD5 selected in between 20 and 30% of connections. Today, 7 years later, we find that same cipher is selected in only 0.001% of connections, and offered by clients in only 8.4% of connections. Later, Holz et al. studied the use of TLS in email clients [69]. Lee et al. performed active scans of a sample of TLS/SSL servers in order to study ciphers supported and certificates in 2007 [90]. In 2012 the **SSL Notary** studied TLS/SSL certificates collected from live traffic [4]. The SSL Notary continues to run today[11] , though still mainly focused on certificates and servers rather than clients. In 2014, Huang et al. described a way to detect forged SSL certificates via a flash plugin that could observe the raw certificate sent to the user [76].

3.2.9.2 ClientHello Fingerprinting

Several studies have used ClientHello messages to fingerprint TLS implementations. Most notably, in 2009, Ristić described how to fingerprint SSL/TLS clients by analyzing parameters in the handshake, including the cipher suites and extensions list [87, 127, 128]. Several works have since observed that these fingerprints can be used to identify and fingerprint third-party library use in Android applications [122], and detect malware [5, 19]. Durumeric et al. used TLS fingerprints and compared them to browser-provided user agents on a popular website to detect HTTPS interception by antiviruses and middleboxes [41].

While these works used ClientHello message to identify clients, we analyze the distribution of clients, ciphers and tls versions used, and fingerprintability of censorship circumvention tools, which to our knowledge has not been studied in this context.

3.2.9.3 Traffic Obfuscation Analysis

There are 2 general techniques [149] that censorship circumvention tools employ to avoid identification: mimic an allowed type of content, or randomize the traffic shape to prevent blacklisting. In the former case, developers would have to eliminate all disparities between circumvention circumvention tool and imitated protocol, and will protect against whitelisting, as long as mimicked application is popular or important

[11] `https://notary.icsi.berkeley.edu/`

enough to avoid blocking. Houmansadr et al. demonstrated [71] that it is very difficult to successfully mimic an application-layer software. Randomized protocols, such as obfs4, while not being able to defend against whitelist approach, may counter blacklisting, which is used more commonly. A study by Liang Wang et al. examined attacks based on semantics, entropy, timing and packet headers [153], and demonstrated efficiency of an entropy-based classifier in detecting obfs3 and obfs4. In 2013, tunneling loss-intolerant protocols over loss-tolerant cover channels, was shown [64] to allow censors to interfere with the channel safely, without disrupting intended use of cover-protocol. For details on observability and traffic analysis resistance of existing anti-censorship tools, reader can refer to Systematizations of Knowledge [83, 149].

lib•erate [93] is a library, that takes an alternative approach, and instead of hiding the traffic, it features numerous techniques that use bugs in DPI to evade identification. Even though all proposed evasion techniques are susceptible to countermeasures, it might be cheaper for the anti-censorship community to integrate and update lib•erate, than for censors to fix all the bugs in DPI boxes.

3.2.10 Discussion

3.2.10.1 Ethical Considerations

Studying real-world Internet traffic requires care to ensure user privacy. We designed our collection infrastructure to anonymize or discard potentially sensitive data. For example, we collected only the /16 subnet of the source IP address and SNI value for each connection. This allows us to tell if a connection originated on our campus, but not what individual user generated it. For connections originated externally, we often cannot even determine what AS the source was located in.

We applied for and received IRB exemption for our collection methodology, and we worked closely with our institution's IT and networking staff, who approved the specifics of our collection methods. We have responsibly disclosed our findings regarding the observability of the censorship circumvention tools, and are continuing to work with their respective developers to discuss and offer potential solutions.

3.2.10.2 Dataset Limitations

Although we have captured billions of TLS connections, there are limitations to what our infrastructure can collect. For example, fragmented TLS messages and out-of-order TCP packets are not parsed by our system. In addition, because we received the full-duplex 10 Gbps mirror of campus traffic on a single (half-duplex) port, it is possible for our copy of network traffic to saturate when the combined bi-directional traffic exceeds 10 Gbps. This occurs for several hours each day during peak load. We performed a simple experiment to quantify how this impacts our collection of TLS fingerprints.

Every hour, we made 100 TLS connections through our campus with a unique TLS fingerprint that did not appear in our dataset. This allowed us to see at what time of day we dropped fingerprints: any hour where we received fewer than 100 of these fingerprints indicated data loss. Over 88% of the hours we ran our experiment recorded all 100 of our test connections. However, during peak hours, which lasted approximately 5 hours per weekday, the minimum number captured in an hour was 43, and the median was 80. We conclude that the only times we do not capture TLS fingerprints is when the tap switch cannot forward us packets due to congestion, and all other times we receive and properly record practically all connections.

3.2.10.3 Future Work

ClientHello messages provide a rich amount of features useful in fingerprinting TLS implementations, but there are other messages in the TLS connection that could be used to detect or block tools. For instance, once the connection is established and sends encrypted records, the lengths of these encrypted records may reveal differences between implementations [153]. Collecting and better understanding the distribution of these (in conjunction with the information gleaned from Client Hello messages) could greatly help circumvention tools be more robust.

Another direction could be to study **user behavior** to better understand if existing tools that pretend to be users visiting popular CDNs or websites (e.g. in domain fronting or refraction networking) are easily distinguishable by the pattern or timing of connections they make.

Beyond TLS over TCP, measuring UDP TLS may be useful in performing similar analysis on DTLS protocols, such as those used by the VPN tools we investigated.

3.2.11 Conclusion

We have analyzed real-world TLS traffic in the context of censorship circumvention tools, focusing on the first protocol messages sent between clients and servers that may allow censors to identify tools and implementations. We analyzed several circumvention tools that use TLS in various ways, and find problems with nearly all of them. To address these systemic problems, we have developed the uTLS library, designed to generated arbitrary ClientHello messages and provide applications full control over the TLS handshake, enabling them evade identification and blocking with minimal developer effort. We release our collected data, combined with tools to facilitate further analysis at `https://tlsfingerprint.io`.

Acknowledgements

We would like to thank the University of Colorado IT and Network Security team, particularly Dan Jones and Conan Moore for their valuable assistance and feedback in setting up our tap infrastructure. We thank J. Alex Halderman for his feedback in early drafts of this work. We also thank the developers at the circumvention projects we contacted, including Rod Hynes from Psiphon, and Ox Cart from Lantern, and David Fifield for their discussion on uTLS. Finally, we are deeply grateful to Alphabet Jigsaw, particularly Ben Schwartz, who helped to design and implement uTLS at the initial stages of the project and continued discussion thereafter.

3.2.12 Appendices

3.2.12.1 Multiple fingerprints usage

We show the ease of using uTLS as compared to using the standard `crypto/tls` library (which provides no control over TLS). In this configuration, uTLS will mimic popular browsers until an unblocked one is found.

Listing 3.1: Dialing with utls.Roller

```
    utlsRoller, err := tls.NewRoller()
    if err != nil {
        return err
    }

    conn, err := utlsRoller.Dial("tcp",
        "10.1.2.3:443", "golang.org")
    if err != nil {
        return err
    }
    conn.Write([]byte("Hello, Golang!"))
```

Listing 3.2: Dialing with standard crypto/tls and no mimicry

```
    tlsConf := tls.Config{
        ServerName: "golang.org",
    }

    conn, err = tls.Dial("tcp",
        "10.1.2.3:443", &tlsConf)
    if err != nil {
        return err
    }
    conn.Write([]byte("Hello, Golang!"))
```

3.2.12.2 Generated code example

Example of code, auto-generated by `https://tlsfingerprint.io/id/2f32363ab2615c86` to mimic one of the Google Chrome variants. This feature is available for any fingerprint in database, but unsupported extensions will have to have extension bodies filled by the user.

Listing 3.3: Automatically generated code to mimic one of the Chrome fingerprint variants

```
tcpConn, err := net.Dial("tcp", "tlsfingerprint.io:443")
if err != nil {
    fmt.Printf("net.Dial() failed: %+v\n", err)
    return
}

config := tls.Config{ServerName: "tlsfingerprint.io"}
tlsConn := tls.Client(tcpConn, &tlsConfig, utls.HelloCustom)
clientHelloSpec := tls.ClientHelloSpec {
    TLSVersMin: tls.VersionTLS10,
    TLSVersMax: tls.VersionTLS12,
    CipherSuites: []uint16{
        tls.GREASE_PLACEHOLDER,
        tls.TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
        tls.TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,
        tls.TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,
        tls.TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,
        tls.TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305,
        tls.TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305,
        tls.TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA,
        tls.TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA,
        tls.TLS_RSA_WITH_AES_128_GCM_SHA256,
        tls.TLS_RSA_WITH_AES_256_GCM_SHA384,
            tls.TLS_RSA_WITH_AES_128_CBC_SHA,
            tls.TLS_RSA_WITH_AES_256_CBC_SHA,
            tls.TLS_RSA_WITH_3DES_EDE_CBC_SHA,
    },
    CompressionMethods: []byte{
        0x00, // compressionNone
    },
    Extensions: []tls.TLSExtension{
        &tls.UtlsGREASEExtension{},
        &tls.RenegotiationInfoExtension{renegotiation: tls.RenegotiateOnceAsClient},
        &tls.SNIExtension{},
        &tls.UtlsExtendedMasterSecretExtension{},
        &tls.SessionTicketExtension{},
        &tls.SignatureAlgorithmsExtension{SupportedSignatureAlgorithms: []SignatureScheme{
            tls.ECDSAWithP256AndSHA256,
            tls.PSSWithSHA256,
            tls.PKCS1WithSHA256,
            tls.ECDSAWithP384AndSHA384,
            tls.PSSWithSHA384,
            tls.PKCS1WithSHA384,
            tls.PSSWithSHA512,
            tls.PKCS1WithSHA512,
            tls.PKCS1WithSHA1,
        },},
        &tls.StatusRequestExtension{},
        &tls.SCTExtension{},
        &tls.ALPNExtension{AlpnProtocols: []string{"h2","http/1.1"}},
        &tls.SupportedPointsExtension{SupportedPoints: []byte{
            0x00, // pointFormatUncompressed
        }},
        &tls.SupportedCurvesExtension{CurveID{
            tls.CurveID(tls.GREASE_PLACEHOLDER),
            tls.X25519,
            tls.CurveP256,
            tls.CurveP384,
        }},
        &tls.UtlsGREASEExtension{},
    },
}
tlsConn.ApplyPreset(clientHelloSpec)

n, err = tlsConn.Write("Hello, World!")
```

Chapter 4

Active Probing

4.1 Background

Active Probing attacks, unlike TLS Fingerprinting and Enumeration Attacks, are executed against endpoints that are already suspected to provide circumvention services, and allow a censor to increase certainty that a certain endpoint should be blocked. Collateral damage from false positives is of utmost concern to the censor and execution of those attacks allows censors to reduce the false-positive rate. Censors execute this attack after using some other signal that allows them to single out suspected connections.

Over the last decade, Active Probing attacks became an essential tool in the toolkit of the Great Firewall of China. In 2012, Philipp Winter and Stefan Lindskog demonstrated [75] that sending a Tor TLS ClientHello to a Tor bridge and immediately closing the connection is "enough bait to attract active scanners". Censors are able to execute this attack without any delay. A study [47] conducted in 2015 by Ensafi et al. observed that 56% of active probes, sent by the Great Firewall of China, arrived within just one second of the first connection to the proxy. In 2019, the GFW was shown [6] to have successfully expanded the scope of active probing attacks beyond Tor pluggable transports to include Shadowsocks.

In the remainder of this section I provide a brief description of multiple known variants of Active Probing attacks: Protocol Confirmation, Replay Attacks and Response Fingerprinting.

Protocol Confirmation

Protocol Confirmation is a very effective active probing attack. GFW has been observed [75, 159] to execute it after observing a connection that appears to be using a specific censorship circumvention protocol, such as the vanilla Tor Protocol, obfs2, or obfs3. The censor would send an active probe, attempting to speak the suspected protocol, and if this protocol was spoken back, that definitively confirms that the endpoint is a circumvention proxy. As such, the censor will be able to block the endpoint without any risk of collateral damage.

Two different defenses have been proposed to defend against protocol confirmation attacks. The censorship circumvention server could be made to require all connecting clients to prove knowledge of a secret that has been shared with them out of band [147, 162]. If the client fails to prove the knowledge of a shared secret, then the server will not provide a response in their protocol, stripping the censor of opportunity to confirm it, making those protocols “probe-resistant”. Alternatively, censorship circumvention tool developers may tunnel circumvention traffic using a popular non-circumvention protocol or application, such as HTTPS [54] or Skype [106]. While censor would still be able to confirm the protocol, the attack would allow not effectively distinguish circumvention servers from non-circumvention servers.

Replay Attack

Another attack that has been observed to be successfully used by GFW [6] is a Replay Attack. Certain “probe-resistant” proxy designs are vulnerable to replay attacks, allowing censors to re-send previously observed client messages and receive a response.

Some “probe-resistant” proxies, such as Shadowsocks-libev [166], implement a cache to prevent replays of previous connections. GFW managed to circumvent this defense by permuting replays [6]. This is not a fundamental issue with “probe-resistant” proxies, as this bug could be fixed; however, it is possible that new bugs leading to distinguishable behavior could be found and introduced in the future. However, there is a fundamental issue with replay attack protection of the silent “probe-resistant” proxy: the behavior of not responding at all to replays is uncommon. It is possible for censors to record the fact that the suspected endpoint previously responded to a certain message, and subsequently see that the same message does not trigger any responses. This behavior is likely rarely seen on the Internet, and even if not never, it still allows censors to increase confidence that an endpoint should be blocked.

A better way to defend against replay attacks is to tunnel a circumvention protocol inside of a popular encrypted protocol that derives a new unique encryption key from random values generated by both client and the server for every connection, such as TLS, SSH, or QUIC. As a result, clients can send the shared secret inside the encrypted tunnel without the risk of a message being replay, since when the censor connects to the server, the server will derive a new encryption key and reject the unauthenticated probe for a plausible reason.

Response Fingerprinting

Censors may also be able to reduce false-positive rates of their blocking by fingerprinting responses of the circumvention servers to various probes. A wide range of probes may be useful to a censor: randomized probes, probes of specific protocols, or an empty probe (connect and wait), as long as they trigger a response that distinguishes the server from the majority of servers on the Internet. The term “response” in this case encompasses not only the response payload, but also the time it takes for the server to close the connection, whether the connection is closed with a TCP RST packet or a TCP FIN, and potentially other features.

In section 4.2, I provide an extended description of the Response Fingerprinting attack, evaluate the applicability of the attack on several prominent circumvention tools, and find which of them are vulnerable. I also suggest multiple ways to defend against the attack, based on empiric measurements of regular servers in the world, such that anti-censorship tools may blend in with them. In section 4.3, I implement one of those proposed defenses, which allows us to defeat all three Active Probing attacks.

4.2 Detecting Probe-resistant Proxies

4.2.1 Introduction

Internet censorship continues to be a pervasive problem online, with users and censors engaging in an ongoing cat-and-mouse game. Users attempt to circumvent censorship using proxies, while censors try to find and block new proxies to prevent their use.

This arms race has led to an evolution in circumvention strategies as toolmakers have developed new ways to hide proxies from censors while still being accessible to users. In recent years, censors have learned to identify and block common proxy protocols using deep packet inspection (DPI), prompting circumventors to create new protocols such as obfsproxy [29] and Scramblesuit [162] that encrypt traffic to be indistinguishable from random. These protocols are designed to have no discernible fingerprints or header fields, making them difficult for censors to passively detect. However, censors such as the Great Firewall of China (GFW) have started actively probing suspected proxies by connecting to them and attempting to communicate using their custom protocols [47]. If a suspected server responds to a known circumvention protocol, the censor can block them as confirmed proxies. Active probing can be especially effective at detecting suspected proxies, because censors can discover new servers as they are used. Previous work has shown that China employs an extensive active probing architecture that is successful in blocking older circumvention protocols like vanilla Tor, obfs2 and obfs3 [47, 75, 159].

In response to this technique, circumventors have developed **probe-resistant** proxy protocols, such as obfs4 [147], Shadowsocks [137], and Lampshade [117], that try to prevent censors from using active probing to discover proxies. These protocols generally require that clients prove knowledge of a secret before the server will respond. The secret is distributed along with the server address out-of-band, for example through Tor's BridgeDB [148] or by email, and is unknown to the censor. Without the secret, censors will not be able to get the server to respond to their own active probes, making it difficult for the censor to confirm what protocol the server speaks.

In this paper, we investigate several popular probe-resistant proxy protocols in use today. Despite being designed to be probe-resistant, we discover new attacks that allow a censor to positively identify proxy servers using only a handful of specially-crafted probes. We compare how known proxy servers respond to our probes with the responses from legitimate servers on the Internet (collected from a passive network tap and active scanning), and find that we can easily distinguish between the two with negligible false positives in our dataset.

We analyze five popular proxy protocols: obfs4 [147] used by Tor, Lampshade [117] used in Lantern, Shadowsocks [137], MTProto [14] used in Telegram, and Obfuscated SSH [91] used in Psiphon. For each of these protocols, we find ways to actively probe servers of these protocols, and distinguish them from non-proxy hosts with a low false positive rate.

Our attacks center around the observation that **never** responding with data is unusual behavior on the Internet. By sending probes comprised of popular protocols as well as random data, we can elicit responses from nearly all endpoints (94%) in our Tap dataset of over 400k IP/port pairs. For endpoints that do not respond, we find TCP behavior such as timeouts and data thresholds that are unique to the proxies compared to other servers, which provide a viable attack for identifying all existing probe-resistant proxy implementations.

We use our dataset to discover the most common responses and TCP behavior of servers online, and use this data to inform how probe-resistant proxies can better blend in with other services, making them harder to block. We work with several proxy developers including Psiphon, Lantern, obfs4, and Outline Shadowsocks to deploy our suggested changes.

4.2.2 Background

In this section, we describe the censor threat model, and provide background on the probe-resistant circumvention protocols we study, focusing on how each prevents censors from probing.

4.2.2.1 Censor Threat Model

We assume that censors can observe network traffic and can perform follow-up probes to servers they see other clients connecting to. The censor can also perform large active network scans (e.g. using ZMap [42]). We assume the censor knows and understands the circumvention protocols and can run local instances of the servers themselves, or can discover subsets of (but not all) real proxy servers through legitimate use of the proxy system.

In this paper, we focus on identifying potential proxies via active probing. Although censors may have other techniques for discovering and blocking proxies (e.g. passive analysis [153] or distribution system enumeration [31]), these attacks are beyond the scope of this paper. While successful circumvention systems must protect against these and other attacks, it is also necessary for the proxies to be resistant to active probes, which is the focus of our paper.

We assume that the censor does not know the secrets distributed out-of-band for every proxy server. Although the censor can use the distribution system to learn small subsets of proxies (and their secrets) and block them, full enumeration of the distribution system is out of scope.

4.2.2.2 Probe-resistant Protocols

Circumvention protocols must avoid two main threats to avoid being easily blocked by censors: passive detection, and active probing. If a censor can passively detect a protocol and distinguish it from legitimate ones, the protocol can be blocked with minimal collateral damage. Circumvention protocols attempt to avoid passive identification by trying to blend in with other protocols or implementations [29, 43, 62, 106], or by creating randomized protocols with no discernible fingerprints [117, 137, 147, 162].

Alternatively, censors can identify or confirm suspected proxies by **actively probing** them. The Great Firewall of China (GFW) has previously been observed sending follow-up probes to suspected Tor bridges [47, 75, 159, 160]. These probes were designed to confirm if a proxy is a Tor bridge (by attempting to use it to access Tor), and block the IP if it is.

To combat this style of active identification, circumvention protocols now attempt to be probe-resistant. For instance, in obfs4, clients are given a secret key that allows them to connect to the provided server. Without this key, active probing censors cannot complete the initial handshake, and the obfs4 server will close the connection after a brief timeout.

We now describe each of the five proxy protocols we study. An overview of how each protocol attempts to thwart active probing is seen in Table 4.1.

obfs4 obfs4 [147] is a generic protocol obfuscation layer, frequently used as a Pluggable Trans-

Protocol	Client's first message	Server Behavior
obfs4 [147]	$K = server\_pubkey \mid NODEID$ $M = client\_pubkey \mid padding \mid HMAC_K(client\_pubkey)$ send: $M \mid HMAC_K(M \mid timestamp)$	Reads **8192** bytes (max handshake padding length) If no valid HMAC is found, server reads a random **0-8192** additional bytes, then **closes** the connection.
Lampshade [117]	send: $RSA_{encrypt}(server\_pubkey, \ldots \mid seed)$	Reads **256** bytes (corresponding to RSA ciphertext length) and attempts to decrypt. If it fails, the server **closes** the connection.
Shadowsocks [137]	$K = HKDF(password, salt, \text{“}ss-subkey\text{”})$ send: $salt \mid AEAD_K(payload\_len) \mid AEAD_K(payload)$	Reads **50** bytes (corresponding to salt, 2-byte length, and AEAD tag) If the tag is invalid, server **closes** the connection.
MTProto Proxy [109]	$K = sha256(seed \mid secret)$ send: $seed \mid IV \mid E_K(magic)$	Does not close the connection on invalid handshake.
OSSH [91]	$K = SHA1^{1000}(seed \mid secret)$ send: $seed \mid E_K(magic \mid payload)$	Reads **24** bytes, and **closes** the connection if the expected magic value is not decrypted.

Table 4.1: **Probe-Resistant Protocols** — In this table we list the first message sent by the client in the probe-resistant proxy protocols we study, and the server's corresponding parsing behavior. Blue text denotes secrets distributed to the client out-of-band. Servers typically close the connection after they receive an invalid handshake message; however, precisely when a server closes a failed connection can reveal a unique fingerprint that could be used to distinguish them from other services.

port [37] for Tor bridges. obfs4 proxies are distributed to users along with a corresponding 20-byte node ID and Curve25519 public key. Upon connecting to an obfs4 server, the client generates their own ephemeral Curve25519 key pair, and sends an initial message containing the client public key, random padding, and an HMAC over the server's public key, node ID, and client public key (encoded using Elligator [13] to be indistinguishable from random), with a second HMAC including a timestamp. The server only responds if the HMACs are valid. Since computing the HMACs requires knowledge of the (secret) server public key and node ID, censors cannot elicit a response from the obfs4 server. If a client sends invalid HMACs (e.g. a probing censor), the server closes the connection after a server-specific random delay.

Lampshade Lampshade [117] is a protocol used by the Lantern censorship circumvention system, and uses RSA to encrypt an initial message from the client. Clients are given the server's RSA public key out of band, and clients use it to encrypt the initial plaintext message containing a 256-bit random seed, the protocol versions, and ciphers that the client supports. Subsequent messages are encrypted/decrypted using the specified cipher (usually AES-GCM) and the client-chosen seed, with prepended payload lengths encrypted using ChaCha20. Since the censor does not know a server's RSA public key, they will be unable to send a valid initial message with a known seed, and any subsequent messages will be ignored. If the server fails to decrypt the expected 256-byte ciphertext (based on an expected magic value in the plaintext), it will close the connection.

Shadowsocks Shadowsocks is a popular protocol developed and used in China, and has many different interoperable implementations available [27, 80, 166]. Shadowsocks does not host or distribute servers themselves. Instead, users must launch their own private proxy instances on cloud providers, and configure it with a user-chosen password and timeout. Users can then tunnel their traffic to their private proxy using the Shadowsocks protocol. Shadowsocks clients generate a key from a secret password and random salt, and then sends the random salt and encrypted payload using an authenticated (AEAD) cipher to the server. A censor without the secret password will not be able to generate valid authenticated ciphertexts, and their invalid ciphertexts will be ignored by the server.

There are numerous implementations of the Shadowsocks protocol [137]; our analysis covers two: the event driven Shadowsocks implementation in Python [27], and Jigsaw's Outline [80]. We refer to these as Shadowsocks-python and Shadowsocks-outline respectively. In the AEAD-mode, both server implementations wait to receive 50 bytes from the client, and if the AEAD tag is invalid, the server closes the connection.

MTProto MTProto [109] is a proprietary protocol designed and used by the Telegram [146] secure messaging app. In countries that block the service, Telegram employs the MTProto Proxy protocol, which only adds obfuscation, and will be referred to as simply MTProto for the remainder of the paper. MTProto derives a secret key from the hash of a random seed and a secret distributed out-of-band. The key is then used to encrypt a 4-byte magic value using AES-CTR. If the server does not decrypt the client's first message to the expected magic value, it will not respond. Since the censor does not know the secret, they will be unable to construct a ciphertext that decrypts to the proper 4-byte value. Upon handshake failure, the server keeps the connection open forever but does not respond.

There are many unofficial implementations of MTProto servers, including versions in Python [14], Golang [136], and NodeJS [58], but we only investigate servers included with the Telegram Android application by default, which are more likely to be an official implementation.

Obfuscated SSH Obfsucated SSH [91] (OSSH) is a simple protocol that wraps the SSH protocol in a layer of encryption, obfuscating its easily-identified headers. Clients send a 16-byte seed, which is hashed along with a secret keyword thousands of times using SHA-1 to derive a key. The key is used to encrypt a 4-byte magic value along with any payload data (i.e. SSH traffic) using RC4. The server re-derives the key from the seed and secret keyword and decrypts to verify the magic value. We specifically looked at Psiphon's OSSH implementation [120], which uses a 32-byte secret keyword distributed to clients. Without knowing the keyword, censors cannot create valid ciphertext[1] . If an OSSH server receives an invalid first message, it closes the connection.

[1] We note that a 4-byte magic value may be insufficient to validate knowledge of the keyword; 1 in 4 billion random ciphertexts may decrypt to contain any 4-byte value, though this is likely infeasible for censors to practically use

4.2.3 Probe Design

To detect probe-resistant proxy servers, we consider what each protocol has in common. Specifically, each protocol requires knowledge of a secret that is proven cryptographically. If the client does not know the secret, the server will simply not respond, and eventually close the connection. We start with the intuition that such **non-response** is uncommon on the Internet, especially to a large corpus of probes in popular protocols. For instance, it's trivial to distinguish between all HTTP servers and probe-resistant proxies, as HTTP servers will respond to HTTP requests while the proxies will not.

Following this intuition, we create several protocol-specific probes for well-known protocols unrelated to censorship circumvention, and send them to a large sample of Internet hosts to see how they respond. If most or all of the hosts respond, or otherwise close the connection in a way distinguishable from the proxy servers we study, then a censor could use this as an effective strategy for identifying proxy servers and distinguishing them from legitimate hosts.

4.2.3.1 Basic Probes

Our probes are data that we send to a server after connecting over TCP. We limit our probes to TCP as all of the proxy protocols we study use TCP, though our techniques could be expanded to UDP. We started with 6 basic probes: HTTP, TLS, Modbus, S7, random bytes, and an empty probe that sends no data after connecting. For each probe, we record how a server responds in terms of the data (if any) it replies with, the time that it closes the connection (if it does), and how it closes the connection (TCP FIN or RST). We briefly describe each of our initial probes.

HTTP For HTTP, we send a simple HTTP/1.1 GET request with a host header of `example.com`. As HTTP remains one of the most popular protocols on the Internet, we expect many servers will respond with HTTP responses, redirects, or error pages. We note that like any protocol-specific probe, even servers that are not HTTP may respond to this probe with an error message native to their protocol.

TLS We send a TLS ClientHello message that is typically generated by the Chromium version 71 browser. This includes popular cipher suites and extensions, though we note that even if there is no mutual support between an actual TLS server and our ClientHello, the server should respond with a TLS Alert message, allowing us to distinguish it from silent proxy servers.

Modbus Modbus is commonly used by programmable logic controllers (PLC) and other embedded devices in supervisory control and data acquisition (SCADA) environments. We used a probe defined in ZGrab [118] that sends a 3-byte command that requests a device info descriptor from the remote host.

S7 S7 is a proprietary protocol used by Siemens PLC devices. We again used the probe defined in ZGrab [118] which sends a request for device identifier over the COTP/S7 protocol.

Random bytes We also send several probes with differing amounts of random bytes, with the hope that servers that attempt to parse this data will fail and respond with an error message or close the connection in a way that distinguishes them from proxy servers.

Empty probe We also have a specific "probe" that sends no data after connecting. Some protocols (such as SSH) have the server communicate first (or simultaneously) with the client. For other protocols, implementations may have different timeouts for when the client has sent some initial data compared to when no data is sent.

We initially probed a small sample of about 50,000 server endpoints collected from our passive tap (see Section 4.2.3.2) with these initial 6 probes, and compared how they responded (data, connection close type and timing) with how instances of our proxy servers responded. With only these probes, there were still hundreds of servers that responded identical to the obfs4 servers we probed. After manual analysis of these servers, we added two additional probes based on the types of servers we identified:

DNS AXFR Although DNS requests are typically done over UDP, DNS zone transfers are carried out over TCP. We identified several hosts in our initial sample that appeared to be DNS servers based on our manual probing using Nmap [97]. To detect these using our probes, we crafted a DNS AXFR (zone transfer) query probe based on the DNS specification [105].

STUN We discovered several endpoints on TCP port 5004 that we were unable to directly identify. We found many of the hosts also had port 443 open and responded with TLS self-signed certificates that suggested they were Cisco WebEx devices. While this confirmed these were unlikely to be proxy servers, we were also able to find a more direct way to identify these hosts. We used our passive network tap to look at data sent over TCP port 5004, and used the data we collected to help us identify what protocol these devices support and how to generate a probe for them. Although an uncommon port (we only saw 4 data-carrying packets over several days of collection on a 10 Gbps tap), we were able to identify this port as supporting Session Traversal Utilities for NAT (STUN) protocols based on a magic cookie value included in the data. With this knowledge, we implemented a STUN probe based on the ZGrab Golang library, which we confirmed elicits responses from these remaining hosts, allowing us to directly distinguish them from proxies.

4.2.3.2 Comparison Dataset

We create a dataset comprised of known **proxy** endpoints (IP/port pairs) and **common** (non-proxy) endpoints. For each proxy we investigate, we collect a sample set of active endpoints from their respective proxy distribution system (e.g. BridgeDB), contacting the developers, or by running our own instance. We collect over 20 obfs4 proxy endpoints from Tor's BridgeDB, and receive 3 Lampshade proxies from Lantern developers. We obtain 3 OSSH proxy endpoints from Psiphon developers, and discover 3 endpoints of MTProto by using the Telegram application. As Shadowsocks is designed for users to run their own proxies, we set up our own Shadowsocks-python instance (configured using the chacha20-ietf-poly1305 cipher) and received an address of Shadowsocks-outline from developers.

Gathering a realistic "common" (non-proxy) endpoints dataset is trickier. Ideally, we want a large set of endpoints that contains a diverse set of non-proxy hosts. We could potentially create our own set of endpoints by running known implementations of non-proxy services ourselves, such as popular web servers, mail servers, and other network services. But this would fail to capture proprietary servers as well as the long-tail of obscure servers that are present in the real world.

Instead, we collect likely non-proxy endpoints from two sources: active network scans using ZMap [42], and passive collection of netflow data. We use ZMap to send a SYN packet to 20,000 hosts on every TCP port (0–65535) for a total of 1.3 billion probes. We discover 1.5 million endpoints that responded with SYN-ACKs, which we label as our ZMap dataset.

For our passive dataset, we collect endpoints by sampling netflow data from a 10 Gbps router at the

University of Colorado. Our intuition is that due to its network position in a country that does not censor its Internet, the vast majority of traffic seen from this vantage point will not contain proxies. This ISP's users have little motivation to use censorship circumvention proxies, so endpoints collected here are likely to be predominantly other services. Moreover, these hosts are more representative of useful services, as opposed to endpoints in the ZMap scan that may not have any actual clients connect to them beside our scanner.

Over a 3-day timespan, we collected over 550,000 unique server IP/port endpoints from our ISP that we observed sending data, and sent follow-up connections from our scanning server. Of these, 433,286 (79%) hosts accepted our connection (responded with a TCP SYN-ACK), with the remaining majority simply timing out during the attempted connection. We believe this response rate can be explained by two reasons. First, our follow-up scans occurred up to several days after the connections were observed, and some servers could have moved IPs or been taken offline in the meantime. Second, servers might be configured with firewalls that only allow access from certain IPs, potentially blocking our ZMap scanning host. Nonetheless, we are still able to capture over 400,000 unique endpoints in this dataset that we know are servers that have been observed sending data.

Both datasets might still contain some amount of actual proxy endpoints in them, which we investigate further in Section 4.2.5.4. Nonetheless, these two datasets provide a diverse list of common endpoints for us to compare with our limited proxies.

4.2.4 Identifying Proxies

A censor's goal is to find ways to differentiate proxy servers from other benign servers on the Internet, in order to block proxies without blocking other services. In this section, we discuss techniques for differentiating servers from one another for the purpose of uniquely identifying proxy servers.

At a high level, our goal is to identify ways to evoke unique responses from proxy servers in comparison to non-proxy servers. If we are able to get a proxy server implementation to respond in a distinct way from every other server on the Internet, censors could use their unique responses to identify and block proxies. We identify three critical features in the ways that servers respond to probes that can be used to fingerprint modern proxies: response data, connection timeouts, and close thresholds, which we detail next.

4.2.4.1 Response Data

Most servers will respond with some kind of data when sent a probe. For instance, HTTP servers will respond to our HTTP probe, but many other protocols will respond with error messages or information when they receive application layer data that they do not understand. On the flip side, we observe that none of our proxy servers respond with any data for any of the probes. This is due to the proxy strategy of remaining silent unless the client proves knowledge of the secret, which a probing censor is unable to do. However, if the censor has a set of probes that can coax at least one response from all hosts that aren't proxies, then non-response could inform a censor that the host is a proxy and safe to block.

Proxies might try to provide some form of response, though this likely commits the proxy to a particular protocol, which has been shown to be a difficult strategy to employ correctly [71]. Modern probe-resistant proxies never respond with data to any of our probes, so we can easily mark any servers that respond with data to any of our probes as a non-proxy endpoint.

4.2.4.2 Timeouts

Even if a server does not reply with any data, they might still close the connection in unique ways. For example, some servers have an application-specified timeout, after which they will close a connection. Timeouts might also be different depending on the state that a server is in. For instance, a server might timeout a connection after 10 seconds if it hasn't received any data, but timeout after 60 seconds if it has received some initial data from the client.

Servers might also differ in the way that they close the connection after the timeout. TCP sessions end after either a TCP FIN or TCP RST packet is sent as determined by the interactions with the application and the underlying operating system.

4.2.4.3 Buffer Thresholds

Finally, we discover another implementation-specific behavior that can distinguish endpoints even if they have identical timeouts and never respond with data. Consider a server that reads *N* bytes from the client, and attempts to parse it as a protocol header. If the parsing fails (e.g. invalid fields, checksums or MACs), the server may simply close the connection. However, if the client sends only $N-1$ bytes, the server might keep the connection open and wait for additional data before it attempts to parse.

We term such a data limit the **close threshold** of a server. If a client (or probing censor) sends less than this number of bytes, the server will wait, even if those bytes are random and non-protocol compliant. However, as soon as the client sends data beyond the threshold limit, the server will attempt to parse the data (and likely fail in the case of random data), and close the connection with either FIN or RST.

Not every server implementation has a close threshold. Some implementations may instead read from the connection forever after an error is encountered, or only close the connection after a data-independent timeout. Nonetheless, servers that do have a threshold provide an additional way for censors to identify and distinguish them from other servers and implementations.

We also discover a second type of identifying threshold for some servers that close the connection after reading a threshold number of bytes, which we confirmed happens in a typical Linux application. When a program closes a TCP connection, the operating system normally sends a FIN packet and completes a 4-way closing handshake with the remote end. However, in certain cases the connection will be closed with a RST packet instead. On Linux, we find if there is any **unread data** in the connection buffer, the operating system will send a RST packet. We define **FIN threshold** and **RST threshold** as the amount of bytes needed to be sent to a server in order to specifically trigger the FIN or RST, while **close threshold** refers to whichever occurs first (lower number of bytes).

Figure 4.1 illustrates the FIN and RST thresholds and how they commonly relate. Data sent up to the FIN threshold will cause the server to keep the connection open, while data sent beyond that will cause the server to close the connection with a FIN. If enough data is sent to exceed the RST threshold, the server will close with a RST.

These limits are due to **application-specific** behavior. While prior work has demonstrated ways to measure TCP behavior of the underlying OS (e.g. via congestion control [115]), to the best of our knowledge, we are the first to identify and measure these application-specific thresholds in TCP programs.

As an example, obfs4 has a randomized close threshold between 8192 and 16384 bytes. Since the

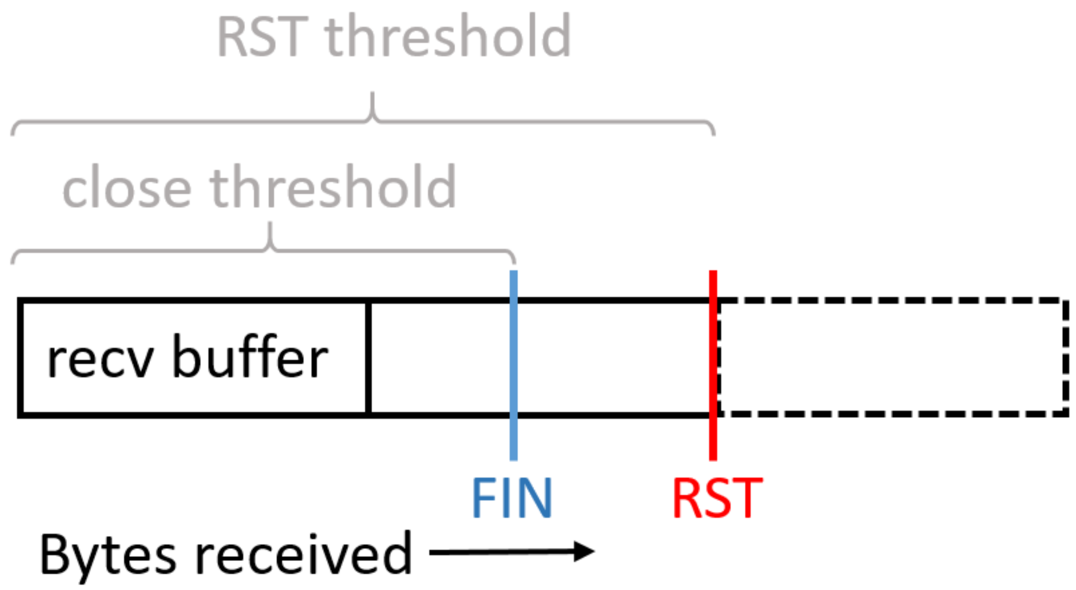


Figure 4.1: **TCP Threshold** — Many TCP server applications close the connection after receiving a certain threshold of bytes from the client (e.g. protocol headers or expected payload chunks) when they fail to parse them. Servers that close a connection after a threshold will send a FIN packet. However, if the application has not read (by calling `recv`) all of the data from the OS's connection buffer when it calls `close`, the server will send a RST instead. These thresholds can be learned remotely (via binary search using random/invalid data) and used as a way to identify or fingerprint server implementations.

	Connection Timeout	FIN Threshold	RST Threshold
obfs4	60–180s	8–16 KB	next mod 1448
Lampshade	90 / 135 s	256 bytes	257 bytes
Shadowsocks-python	configurable	50 bytes	-
Shadowsocks-outline	configurable	50 bytes	51 bytes
MTProto	-	-	-
OSSH	30 s	24 bytes	25 bytes

Table 4.2: **Proxy Timeouts and Thresholds** — We measured how long each probe-resistant proxy lets a connection stay open before it times out and closes it (Connection Timeout), and how many bytes we can send to a proxy before it immediately closes the connection with a FIN or RST. obfs4's RST Threshold is the next value after the FIN threshold that is divisible by 1448.

obfs4 handshake can be up to 8192 bytes in length, the server reads this many bytes before determining the client is invalid, and entering a `closeAfterDelay` function. This function either closes the connection after a random delay (30–90 seconds) or after the server has read an additional N bytes, for N chosen randomly between 0 and 8192 at server startup. However, each of these reads is done using a 1448-byte buffer. This means that obfs4 servers will send a FIN if they are sent $8192+N$ bytes, and send a RST if they are sent $8192+N-((8192+N) \mod 1448)+1448$ bytes, a behavior that appears to be unique to obfs4. Table 4.2 shows the timeouts and thresholds for the probe-resistant proxies we study.

Threshold Detector

We developed a tool to binary search for the close and RST threshold for a given endpoint. Our tool starts by connecting and sending 4,096 bytes of random data, and seeing if the connection is closed (FIN or RST) within 3 seconds. If it is, we make subsequent connections, halving the number of bytes we send until we reach a value that the server does not close the connection within a second. If even the original 4,096 byte probe does not result in a close, we reconnect and send twice the amount of data (up to 1 MB) until we find a close. Once we have identified an upper and lower bound for a given server, we binary search between these bounds to obtain the exact value of the close threshold. Once a close threshold is found, we reconfirm using two follow-up probes: one immediately below and one immediately above the found threshold. We use 3 different seeds to generate the random data in those confirmation probes. If the behavior is identical with all 3 seeds, we mark it as having a stable close threshold. Otherwise, we mark the endpoint as unstable and

Probe	Tap	ZMap
TLS	87.8%	0.90%
HTTP	64.6%	0.95%
STUN	52.5%	0.56%
Empty	8.4%	0.23%
S7	56.9%	0.66%
Modbus	51.4%	0.54%
DNS-AXFR	58.8%	0.67%
Any	**94.0%**	**1.16%**

Table 4.3: **Percent of Endpoints Responding with Data** — For both our ISP passive (Tap) and active (ZMap) datasets, we report the percent of endpoints that respond to each of our probes, as well as the percent that responded to at least one (Any).

ignore the threshold value.

4.2.5 Evaluation

To evaluate our competency at distinguishing proxies from other servers, we sent our probes to over 1.9 million endpoints contained in our "common servers" dataset, and observed the ways in which their responses differ from those of known proxies. As a reminder, this dataset contains over 500,000 endpoints observed at our passive ISP tap, and approximately 1.5 million endpoints collected using ZMap on random IPs and ports. We send 13 probes to each endpoint (our 7 probes from Section 4.2.3.1 and 6 probes with random data ranging from 23 bytes to over 17KB) and record if and when the server responds with data or closes the connection. If the server closes, we record if it used a FIN or RST. If the server does not respond or close the connection, we timeout after 300 seconds and mark the result as a TIMEOUT. In addition to sending probes, we also use our threshold detector on each endpoint, recording their close thresholds.

Table 4.3 shows each of our (non-random) probes, and the percent of endpoints in each of our two "common endpoints" datasets that respond to the probes with data. Since probe-resistant proxies never respond with data, we can immediately discard endpoints that reply to any of our probes as non-proxies. In our passive Tap dataset, this rules out 94% of hosts, leaving only 26,021 potential proxies. On the other hand, in our ZMap dataset, the overwhelming majority of hosts do not respond with data to any probes, allowing us to discard only 1.2% of endpoints based solely on response data.

We find a significant reason for this discrepancy is that ZMap identifies a large number of firewalls that employ common "chaff" strategies, where they respond to every SYN they receive on all ports and IPs in their particular subnet [163]. We observe over 42% of endpoints in our ZMap dataset behave identically by never sending data and never closing the connection (up to our 300 second timeout) to our probes[2] . To understand the diversity of responses, we clustered responses from endpoints by constructing **response sets** that captures the response type (FIN, RST, timeout), number of bytes they respond with (possibly 0), and the time of connection close or timeout (binned to integer seconds) for each non-random probe we send. If two endpoints have identical response sets (for instance they both respond to probe-A with a FIN after 3 seconds and a 10-byte response, and probe-B with a RST after 9 seconds with no data, etc), we say they are in the same response group. In this clustering, we ignore any actual content and only compare lengths if any response data is sent.

The most popular response group in our Tap dataset comprises only 3.0% of endpoints, and appears to be TLS servers (99.9% are on endpoints with port 443) in Cloudflare's network. These hosts respond to our TLS probe with a handshake alert, due to the lack of Server Name Indication (SNI) in our ClientHello probe. Figure 4.2 shows a CDF of unique response sets (sorted by popularity) for our ZMap and Tap dataset, and illustrates the larger diversity in our Tap dataset. The top 10 response sets comprise over 80% of endpoints in the ZMap dataset, but only 13% of endpoints collected at our Tap.

4.2.5.1 Designing a decision tree

We now turn to distinguishing proxies from the remaining set of common non-proxy endpoints that did not respond with any data.

A natural choice for distinguishing proxies from common hosts is to use machine learning to automate the synthesis of a classifier model. We evaluated using automatically-generated decision trees in Appendix 4.2.11, but did not find them to provide higher accuracy, and found they require more manual labor than the decision trees we build manually. In this section we instead focus on manually creating a decision tree that can distinguish proxies from common endpoints, as the resulting trees are easier to interpret and are

[2] excluding our random data probes

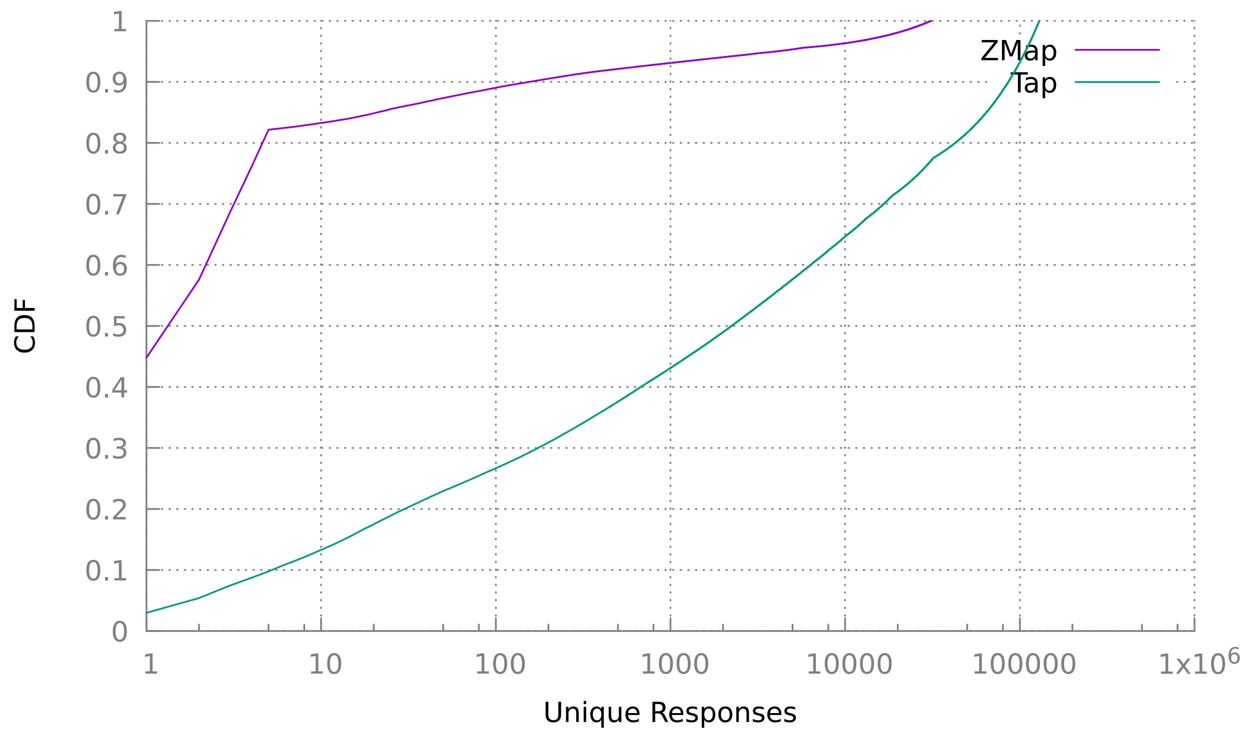


Figure 4.2: **Response Set CDF** — We group endpoints based on how they respond to our 7 non-random probes, capturing the number of bytes the endpoint replies with, how long it keeps the connection open (binned to seconds), and how it closed the connection. Over 42% of endpoints respond the same (timeout after 300 seconds) in our ZMap dataset, which we identify as a common firewall behavior. Despite being smaller, the Tap dataset is much more diverse (129k vs 31k unique response sets).

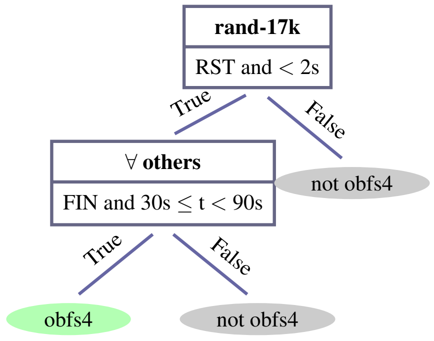


Figure 4.3: **obfs4 Decision Tree** — Each box represents a list of probes (top) and the expected response (close type (FIN, RST, TIMEOUT)) and the time bound. For instance, **rand-17k** represents a 17 KB random probe, which obfs4 servers respond to with an immediate RST due it exceeding obfs4's RST threshold. All other probes in our set are below the close thresholds, and cause obfs4 proxies to produce a FIN between 30 and 90 seconds. Responses that do not satisfy all of these criteria are labelled **not obfs4**.

as effective compared to the automatically-generated ones.

For each proxy protocol, we manually created a decision tree from analysis of the proxy's source code and observed behavior to our probes. Since none of the proxies send data responses for any of our probes, the first decision layer is to mark endpoints that respond with any data as non-proxies. As shown in Table 4.3, this eliminates 94% of endpoints in our Tap dataset, but only 1.2% of endpoints in our ZMap dataset.

The next layers of each decision tree are protocol specific, and we describe each in Figures 4.3-4.8. In each of these figures, each box lists at the top a set of probes, and at the bottom the expected response. Each expected response is in the form of a close type (FIN, RST, or TIMEOUT) and a time bound (in integer seconds). For example, Figure 4.3 shows the decision tree for obfs4, which first looks at the response to the **rand-17k** probe that sends 17 KB of random data to the endpoint. If the endpoint responds with a RST in under 2 seconds, the next layer in the tree is checked. Otherwise, the endpoint is labeled as not obfs4. In the next layer, every other probe's response is checked to be a FIN between 30 and 90 seconds. If they all are, the endpoint is labeled as obfs4, otherwise it is not.

We note that since all proxies have mutually exclusive decision trees, it is possible to compose a multi-class classifier by checking conditions for each proxy label in any order, and classifying the unmatched samples as not proxies.

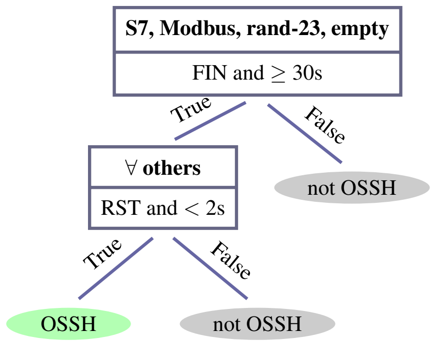


Figure 4.4: **OSSH Decision Tree** — OSSH's FIN threshold is 24 bytes, with a RST threshold of 25. Thus, any probes less than 24 bytes in length (S7, STUN, Modbus, and rand-23) cause OSSH servers to send a FIN after 30 seconds. All our other probes are beyond OSSH's RST threshold, causing it to send an immediate RST.

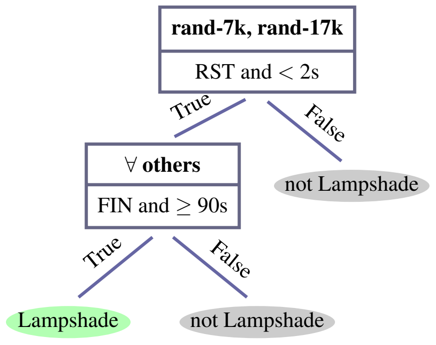


Figure 4.5: **Lampshade Decision Tree** — Lampshade's RST threshold is 257 bytes. Only two of our probes (our 7 KB and 17 KB random probes) exceed this, and cause Lampshade servers to RST immediately. Otherwise, Lampshade servers timeout after 90 seconds. Despite not having Lampshade-specific probes (meaning our data will likely over-find potential Lampshade servers), we do not see any servers that meet even this liberal criteria in our Tap dataset.

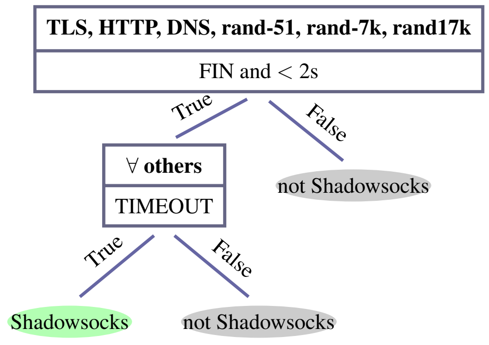


Figure 4.6: **Shadowsocks-Python Decision Tree** — Shadowsocks-python has a FIN threshold of 50 bytes, and no RST threshold (likely owing to the event driven socket buffering, which is unlikely to leave data in the socket between events). Thus, all probes larger than 50 bytes will cause Shadowsocks to immediately close with a FIN, while probes less than 50 bytes cause it to time out.

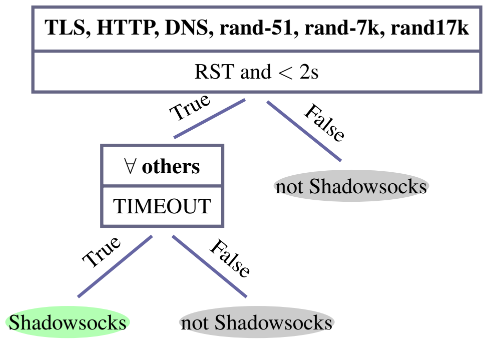


Figure 4.7: **Shadowsocks-Outline Decision Tree** — Shadowsocks-outline has a FIN threshold of 50 bytes, and a RST threshold of 51. Thus, all probes with size less of 50 bytes will cause it to timeout, while probes of size 51 and above would cause Shadowsocks-outline to immediately close with a RST. A 50 byte probe would cause the server to immediately close with FIN, but we did not use such a probe while collecting the data.

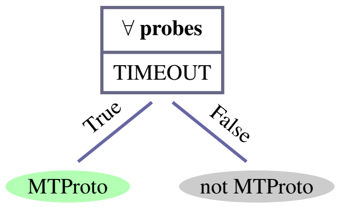


Figure 4.8: **MTProto Decision Tree** — MTProto does not appear to have a close threshold, and does not have any unique timeout (appears to leave the connection open forever). Thus, we label endpoints that time out for all our probes as MTProto.

4.2.5.2 Timeouts

We measure the distribution of the duration an endpoint will keep a connection open before closing it. Figure 4.9 shows the distribution of timeouts (binned to seconds) over our Tap dataset, when we send nothing (Empty) and when we send a single random byte (1 byte). The distribution shows clear modes at 10, 15, 20, 30, 60, and beyond 300 seconds[3] . The different probes have slightly different distributions, meaning that many endpoints have different timeouts based on the amount of data they receive. Still, we find that 71% of endpoints in the Tap dataset have the same timeout for the empty and 1-byte probes.

This distribution suggests that the popular strategy of using random timeouts is far from ideal: 82% of endpoints timeout at one of the top ten timeout values. For instance, timing out at a random value such as 74 seconds puts such a hypothetical proxy in a group with only 0.02% of endpoints, making it easier for censors to block without worry of blocking legitimate services. We note this analysis is over **all** endpoints in our Tap dataset, and many of those endpoints may have other distinguishing features, such as sending response data or having unique close thresholds. We further analyze timeout values that probe-resistant proxies could use in Section 4.2.6.

Our measurement tool marks a connection as TIMEOUT if it doesn't close or send data for over 300 seconds. However, it is possible that some servers timeout at values much higher than that, so we performed a follow-up experiment to determine if 300 seconds was a reasonable value to TIMEOUT, or if we were missing distinguishable features by not waiting longer. We made a follow-up connection to the 8500 endpoints that were marked as a 300+ second TIMEOUT in our previous probes (from both Tap and ZMap datasets), and waited up to 2.5 hours (9000 seconds) to see if they would close the connection after 300 seconds. Figure 4.10 shows our results, showing that most servers (97.5%) that previously would be marked as a TIMEOUT at 300 seconds would have also been marked as a TIMEOUT at 9000 seconds. Only an additional 2.5% would have possibly be labelled differently if we had waited longer, suggesting 300 seconds is a reasonable cutoff for a timeout.

This experiment also revealed a bug in our original probing tool, where a server may be marked as

[3] the upper limit of our measurements

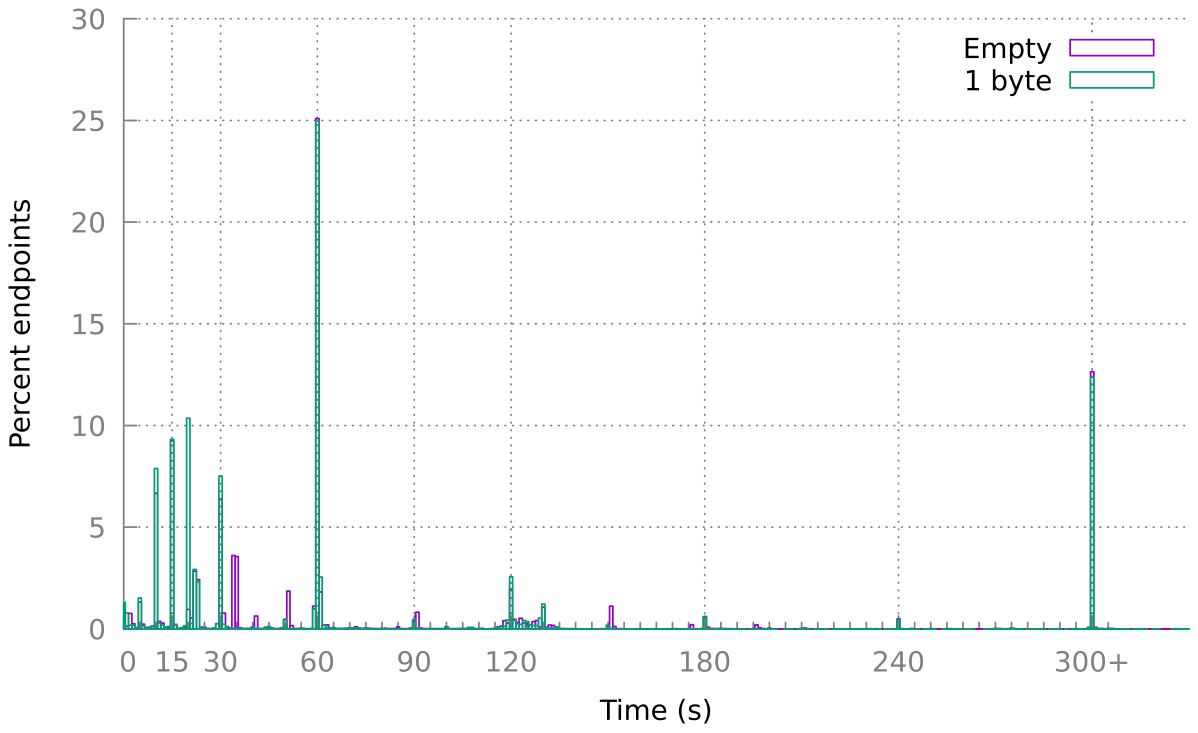


Figure 4.9: **Server Timeouts** — We measure how long servers in our dataset will allow a connection to stay open if we send it no data (Empty) or 1 byte. Unsurprisingly, we find servers time out at expected modes in multiples or fractions of minutes.

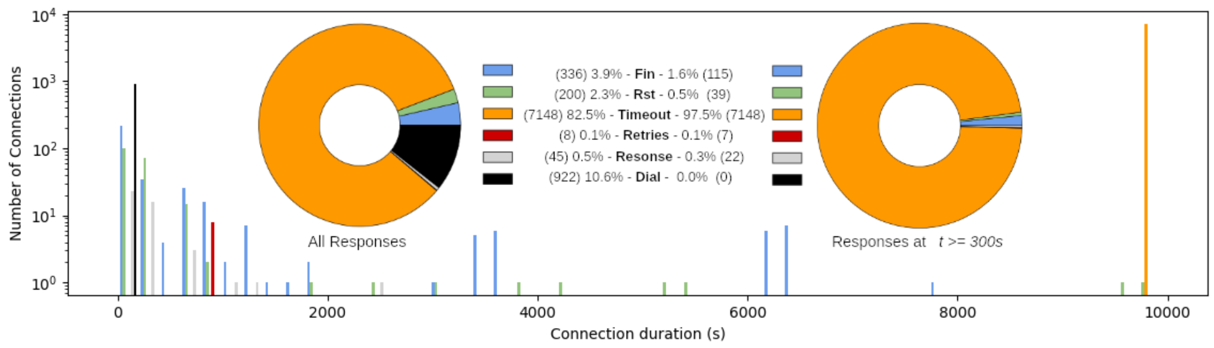


Figure 4.10: **Connection Timeouts** — For the subset of servers that never responded to any probe within 300 s we performed a follow-up scan allowing the connection to stay open for an extended period of time to identify a fair representative of an infinite timeout. The results after 300 s are dominated by the client timeout suggesting that this is a reasonable approximation of an unlimited timeout. The strategy employed by MTProto to never respond and wait for clients to timeout is well represented in common server behavior.

TIMEOUT if it never acknowledges (with a TCP ACK) the data we sent. In this case, we will continue to retransmit the data and our prober's kernel will close the connection before our 300 second timeout, but since no FIN or RST is received, we will mark the connection as a TIMEOUT at less than 300 seconds. We note this is non-standard TCP behavior for servers, and happens in about 1.3% of connections. We label these as **retries** in Figure 4.10. The bug is only triggered by the small fraction of hosts that (counter to the TCP specification) do not send ACKs for received data. We do not see this kind of behavior in any of the proxies we investigated, and do not believe this bug impacts our results.

4.2.5.3 Thresholds

In addition to timeouts, we measure the distribution of close thresholds (the number of bytes that an endpoint closes a connection after receiving) for endpoints in our Tap dataset. Figure 4.11 shows the histogram of close threshold values. The most popular threshold values are 11 and 5 bytes, which we identify is commonly used in TLS server implementations: 5 bytes corresponds to the size of the outer TLS record header, and 11 bytes corresponds to the minimum size of the outer plus inner handshake header. Many TLS implementations naturally read these headers, and close the connection if they fail to parse them. The next most common close threshold is beyond the 1MB limit of our probing tool, likely indicating a non-existent close threshold. We label the proxy server close thresholds on the graph, showing that most are unique, sharing their threshold with less than 0.05% of endpoints in our dataset. An exception is MTProto, which has no observable threshold, behavior shared by about 9% of endpoints in our Tap dataset.

We also investigate whether endpoints follow our expected understanding of FIN/RST thresholds: that servers will close with a FIN after receiving certain number of bytes, and potentially close with a RST after receiving a higher number of bytes. For the endpoints that we used our threshold detector on, we looked at the order of FIN, RST, and TIMEOUT responses as we increase the number of bytes sent to them. For example, if a server sends a FIN after 80 bytes, and a RST after 100, we mark it as "FIN before RST". Figure 4.12 shows the breakdown of FIN/RST orderings. We find that the vast majority of endpoints (85%) are observed sending either only a FIN or a FIN before a RST, which conforms with our expected understanding of the close threshold described in Figure 4.1. Endpoints that send only FIN (and never a RST) are caused either by

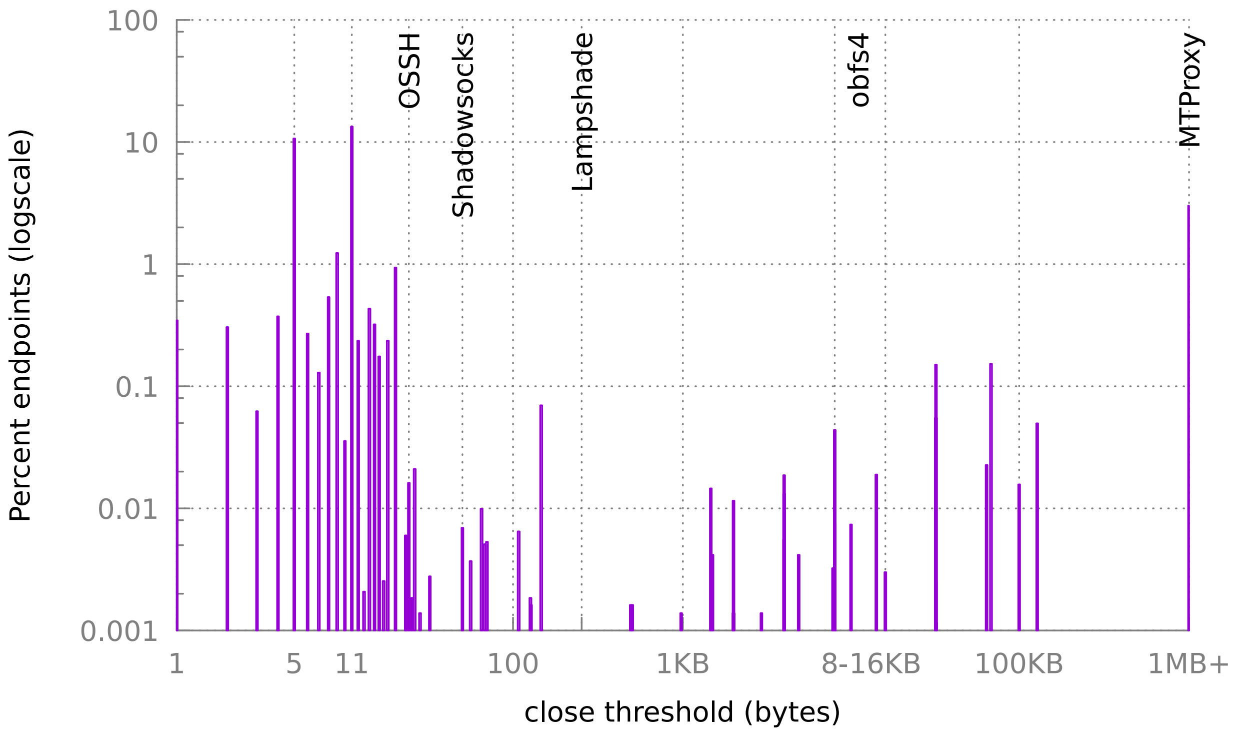


Figure 4.11: **Close Thresholds** — We measured the close threshold of over 400,000 server endpoints observed at our ISP. With the exception of MTProto, most proxy thresholds are relatively unique. We show the thresholds for OSSH (24 bytes), Shadowsocks (50 bytes), Lampshade (256 bytes), obfs4 (8–16 KB), and MTProto (above 1 MB).

	Endpoints w/ Threshold		Decision Tree Labeled	
Proxy	Tap	ZMap	Tap	ZMap
obfs4	355	65	2	0
Lampshade	2	1	0	1
Shadowsocks-Python	30	18	0	8
Shadowsocks-Outline	30	18	0	7
MTProto	13k	106k	3144	296
OSSH	70	5	8	0

Table 4.4: **Decision Tree Results** — We applied our manually-created decision trees to both the Tap (433k endpoints) and ZMap (1.5M endpoints) datasets. We expect the decision trees to label very few or no endpoints as proxies. Indeed, with the exception of MTProto, our decision tree finds very few or no proxies. In some instances, such as OSSH, 7 of the 8 endpoints found in our Tap dataset were confirmed to be actual OSSH servers by their developers. We also present the number of endpoints that have the same close threshold as the proxies we study, with the data showing that thresholds alone are not as discerning as timeouts for identifying proxies.

our detector not sending enough bytes to trigger a RST, or by the application always emptying the socket buffer on every read (common in event-driven applications that use `select` or `poll`). We do observe a minority (5.9%) of servers that only send a RST, and never send a FIN. Many of these servers appear to be previous versions of Microsoft Windows servers that use RST to close connections. A small fraction (0.4%) interleave RSTs and FINs, meaning there appears to be no single threshold for either. However, we find no endpoints that consistently close connections with a RST for lower thresholds and FIN for higher ones (RST before FIN).

4.2.5.4 Identified Proxies

Table 4.4 shows the results of applying our decision trees to both the Tap and ZMap datasets. Despite dozens to hundreds of endpoints having similar close thresholds as each of our proxies, our decision tree is able to cut down to only a handful of proxies that are possibly proxy servers (excepting MTProto). We note that our decision tree only uses the responses from our probes, and we use the close thresholds as additional evidence to help confirm potential proxies.

obfs4 – For obfs4, the 2 servers we identify from our Tap dataset both have a RST threshold of 10241 bytes with no FIN threshold, behavior that we observe to be inconsistent with obfs4 implementations. Both

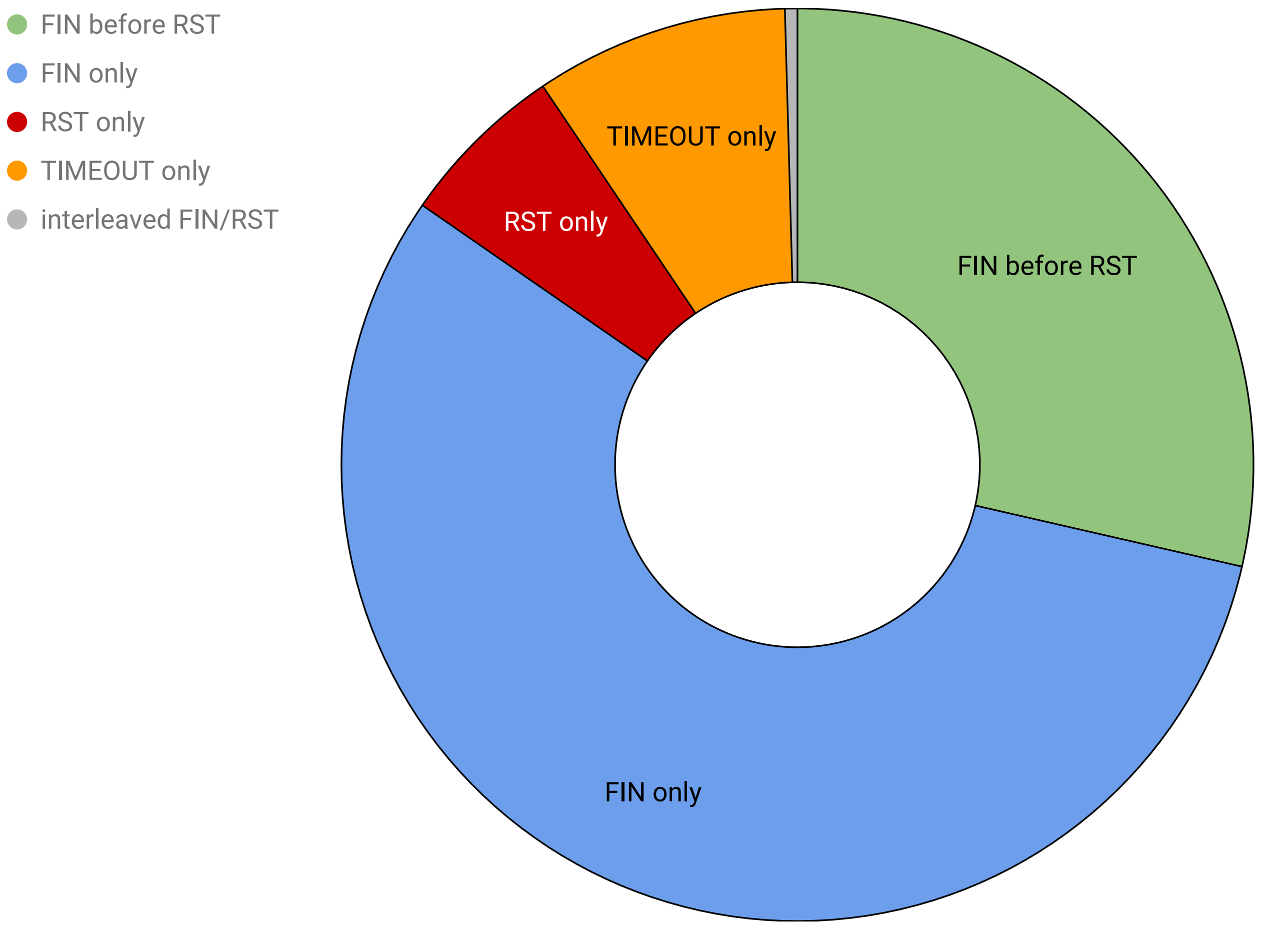


Figure 4.12: **Types of Close** — We measured the threshold behavior of each of over 400,000 server endpoints. As expected, most servers close the connection with a FIN, some of which would send RST if additional data is sent (FIN before RST). The majority of servers send only a FIN, meaning our test did not send enough data to elicit a RST or the server never sends a RST. Less commonly, we find (standards-non-conforming) servers that only close with a RST, or servers that interleave use of RST and FIN.

servers are in China, and one serves a TLS certificate valid for several subdomains of baofeng.com. We are unable to confirm whether these endpoints are running obfs4 servers, but conclude that it is unlikely given the RST threshold results and their location inside a censored country, where they are unlikely to be useful for censorship circumvention.

Lampshade – For Lampshade, only one endpoint was identified in our ZMap dataset. This endpoint did not have a stable close threshold, and is therefore not a Lampshade instance. This endpoint is running on a host that also serves a Traccar login page, an open source tool for interfacing with GPS trackers.

Shadowsocks – Our decision tree identifies 8 endpoints in our ZMap dataset as Shadowsocks-python, all of which have a 50 byte FIN threshold, suggestive of the proxy. We performed manual follow up scans, and found all but one of these endpoints also run SSH on the same host, with the same version banner. These hosts are scattered around the world in various hosting networks, though none in censored countries. We cannot conclude for certain that these are all Shadowsocks servers, but given the threshold results, and their similarity and network locations, we believe they likely are. If we extrapolate from our small ZMap scan to the rest of the Internet, we would estimate that there are on the order of 1 million Shadowsocks-python **endpoints** running worldwide. Upon further investigation of the identified Shadowsocks endpoints, 5 are in the same hosting provider (xTom) and each has a distinct set of 700 sequential TCP ports open that all exhibit identical behavior consistent with Shadowsocks. For example, one IP has TCP ports 30000–30699 open, all seemingly identical behavior. If 5 out of every 8 Shadowsocks-python servers had 700 ports open in this manner (and the others had only a single port), our 1 million Shadowsocks-python endpoints would extrapolate to about 2285 Shadowsocks servers (unique IPs) worldwide.

Our decision tree also identifies 7 endpoints in the ZMap dataset as Shadowsocks-outline. 6 of those endpoints are in Netropy IP blocks in South Korea, with the remaining in Cogent's network. By the time we performed manual analysis on these endpoints, they were no longer up, preventing follow-up analysis.

MTProto – Over 3,000 endpoints were classified as MTProto in our datasets, though likely few (if any) of these are truly MTProto servers. This over-count is due to the simple decision tree used to classify MTProto: many endpoints simply never timeout and do not have any close thresholds, making them difficult to distinguish from one another. These endpoints represent 0.56% and 0.02% of our Tap and ZMap datasets respectively. This suggests that MTProto's strategy of camouflage is effective at evading active probing, because these endpoints offer truly no response, even at the TCP level. We provide in-depth discussion of effective defense strategies in section 4.2.6.

OSSH – We classify 8 endpoints in our Tap dataset as OSSH. We followed up with Psiphon, a popular circumvention tool that commonly uses OSSH servers, and identified that 7 of these were their own servers, confirming they were indeed OSSH endpoints. The remaining was hosted in Linode's network on port 443, but we cannot confirm if it is running OSSH or an unrelated service.

Summary Our manually-crafted decision tree is generally effective at distinguishing proxy servers from common hosts in both our Tap and ZMap datasets (with the exception of MTProto). In some cases, the handful of endpoints that were classified as proxies turned out to be discovered proxies, which we confirmed through private conversation with their developers. In other cases, such as Shadowsocks, we have circumstantial evidence that supports the claim that these endpoints are proxies, but no definitive way to confirm. Despite our low false-positive rate (conservatively, less than 0.001% for all protocols beside MTProto), we note that the base rate of proxies as compared to common endpoints is an important consideration for would-be censors: even seemingly negligible false positive rates can be too high for censors to suffer [153]. We also observe that MTProto demonstrates an effective behavior that makes it more difficult to distinguish from a small but non-negligible fraction of non-proxy endpoints (0.56% and 0.02% of Tap and ZMap datasets), offering a potential defense to other proxies, which we further investigate in Section 4.2.6.

4.2.6 Defense Evaluation

Given the result that most of the probe-resistant proxies can be identified with a handful of probes, we now turn to discuss how to improve these protocols to protect them from such threats: How **should** probe-resistant proxies respond to best camouflage themselves in with the most servers?

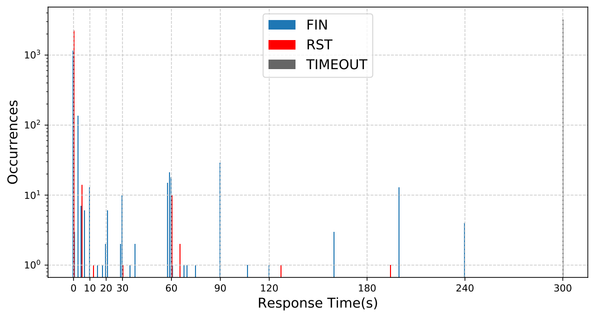
To answer this, we look in our datasets for the most **common responses** to our probes. If proxies respond the same way as thousands of other endpoints, they will better blend in with common hosts on the Internet, making them harder for censors to identify and block.

We note that proxies should not attempt to directly mimic common data-carrying responses to our probes. Sending any response data commits a proxy to a particular protocol, which introduces significant challenges in faithfully mimicking the protocol [62, 71]. Thus, despite TLS errors being the most common response to our probes, we rule out these and other endpoints that respond to our probes with data.

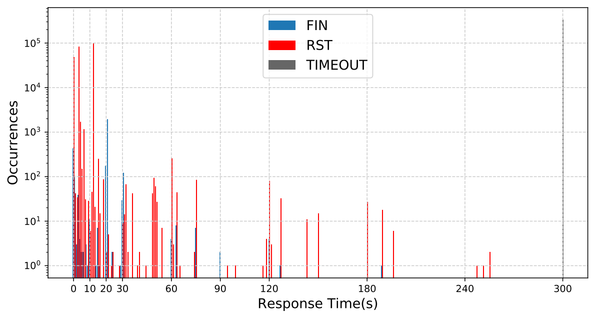
Ruling out the endpoints that respond with data eliminates 407k (94%) endpoints from our tap dataset, and nearly 9k (1.1%) endpoints from our ZMap dataset. However, many of the remaining servers still close the connection at different timeouts depending on the probe we send. For example, servers that have a close threshold will close the connection with FIN or RST at varying times, depending on the length of probe we send. It is possible that other probes beyond our own could elicit other timeouts or even data responses from these servers. We thus only consider servers that are **probe-indifferent**, in that they respond to all of our probes with the same response type (FIN or RST) at a similar time (within 500 milliseconds of their other responses, to allow for network jitter). We exclude the empty probe (where we do not send data) response times from our probe-indifferent endpoints as these servers are still waiting for data and might timeout at a different time.

Figure 4.13 shows the response type and timeout of the 6,956 **probe-indifferent** endpoints in our tap dataset (568,121 in ZMap). In both datasets, the overwhelmingly popular response type is timeout, which indicates the endpoint did not respond within the 5 minute limit for our scanner. As measured in Figure 4.10, these endpoints predominantly never timeout, and instead read from the connection forever without responding. Over 0.7% of our tap dataset endpoints (42% of ZMap) exhibit such "infinite timeout" behavior. Due to this relative ubiquity, **we recommend proxy developers implement unlimited timeouts for failed client handshakes**, keeping the connection open rather than closing it. This strategy is already employed by MTProto servers we probed, and we have made recommendations to other probe-resistant proxy developers to implement this as well.

However, not all proxies may be willing to keep unused or failed connections open forever. In addition, it may be beneficial for circumvention tools to employ multiple strategies, such as selecting between no timeouts and other popular finite timeouts on a per-server basis. For proxies that must timeout, a naive strategy would be to look at common (finite) response times in Figure 4.13, which shows the distribution of probe-indifferent timeouts (i.e. how long a server waited to timeout for all our probes). However, many



(a)



(b)

Figure 4.13: **Probe-Indifferent Server Timeouts** — We define an endpoint as **probe-indifferent** if it responds to all of our probes in the same way (i.e. with only one of FIN, RST, or TIMEOUT at (approximately) the same time). We compare the probe-indifferent timeouts for our Tap dataset (a) and our ZMap dataset (b). The most popular behavior, shared by both datasets, is to never respond to any of our probes, as shown by our 300+ second TIMEOUT in grey.

popular response times are shared by groups of servers that share **other** important characteristics that could be difficult for proxies to mimic. For example, many probe-indifferent endpoints responded with a FIN to all our probes after 90 seconds, but manual investigation reveals that all of these endpoints are in the same /16 and running on the same port (9933). In addition, each of the servers have the same additional port open (9443) that return identical HTTP 503 errors carrying the name of a Canadian video game developer. If probe-resistant proxies attempted to mimic these endpoints by only responding to all failed handshakes with a FIN after 90 seconds, an observant censor could distinguish them from the other common endpoints. Another example popular response is responding with a FIN 3 seconds after our probe, but we observe almost all of these hosts appear to be infrastructure associated with the Chandra X-ray Observatory [26].

One response exhibited by a heterogeneous mix of IP subnets and ports in both datasets is to close the connection with either a RST[4] or FIN at 0 seconds, meaning these servers closed the connection right away after our probes. While intuitive, applications must be careful to ensure they only send FINs or only send RSTs on invalid handshakes, regardless of client probe size. In addition, proxy applications must choose how long to wait if no data is sent when a connection is opened. We find that for endpoints in our tap dataset that send FINs right away to our data probes, the most common timeouts for empty probes (where we send no data) is to close the connection with a FIN after 120, 2, or 60 seconds. We caution that our manual analysis of these endpoints is not exhaustive: there may be other probes that allow censors to distinguish between proxies and common endpoints, and we recommend proxies choose no timeout over a specific finite one. However, if proxies must choose finite timeouts, these values may provide the best cover, provided the proxy also sends FINs right away for any data received.

4.2.7 Related Work

Several prior works have identified ways to passively identify proxy protocols, allowing censors to differentiate proxy traffic from other network traffic. For example, Wang et al. [153] use entropy tests to distinguish obfs4 flows (and other proxies) from common network traffic, observing that it is unusual for normal traffic to have high entropy, particularly early on in the connection. Since even encrypted protocols

[4] accomplished in Linux by setting the `SO_LINGER` socket option with a zero timeout

like SSH and TLS have low-entropy headers, the high-entropy bytes in obfs4 connections is an effective signal. Previously, Wiley [161] used Bayesian models to identify OSSH from other traffic. Finally, Shahbar et al. [138] use a classifier over several traffic features (e.g. packet size, timings, etc) to identify several proxies, including obfs3, a previous version from obfs4 that is not probe-resistant.

Our active probing attack complements these existing passive detectors for two reasons. First, because the base rate of "normal" (non-proxy) traffic is significantly higher than proxy traffic, even with low false positive rates, the majority of flows identified as proxies by a censor may actually be false positives [153]. Thus, our active approach could help censors **confirm** suspected proxies found using passive techniques. Second, proxies can thwart such passive analysis by adding random timing and/or padding to their traffic [22]. However, even with such defenses, existing proxies could still be vulnerable to active probing. Therefore, we argue that **both** passive and active attacks must be addressed, and focus on the active attack in this paper.

One of the main motivations for probe-resistant protocols to use randomized streams is from Houmansadr et al. [71] ("The Parrot is Dead"). This work argued that mimicry of existing protocols such as Skype or HTTP—employed by proxies at the time—is difficult to do correctly, as even small implementation subtleties can reveal differences between true clients and proxies that attempt to mimic them. The authors reveal several active and passive attacks to identify SkypeMorph [106], StegoTorus [158], and CensorSpoofer [154] from protocols (e.g. Skype, HTTP) they attempt to use or mimic.

Finally, there are circumvention tools that do not require endpoints to remain hidden from censors. For example, meek (domain fronting) [54] and TapDance (Refraction Networking) [59, 164] both have users connect to "decoy" sites that are not complicit in the circumvention system, and use network infrastructure (i.e. load balancers or ISP taps) to act as proxies that redirect traffic to their intended destination. Even if a censor discovers the decoy sites (which are public), they cannot block them without also blocking legitimate access to those sites as well. Flash proxy [108], and more recently Snowflake [36], create short-lived proxies in web browsers of users that visit particular websites. These websites serve JavaScript or WebRTC-based proxies that run in the visitor's browser, and can transit traffic for censored users for as long as the visitor remains on the page. These short-lived proxies can still be blocked by censors, but doing so reliably is difficult due to their ephemeral nature.

4.2.8 Discussion

4.2.8.1 Responsible disclosure

We shared our findings with developers of the probe-resistant proxies we studied, who acknowledged and in most instances made changes to address the issue. Specifically, we reached out to developers of Psiphon for OSSH (who pushed a fix on May 13, 2019), obfs4 (fixed June 21, 2019, version 0.0.11), Shadowsocks Outline (fixed September 4, 2019, version 1.0.7), and Lantern's Lampshade (fixed October 31, 2019). Psiphon's fix for OSSH is to read forever after an initial handshake failure, while the others read until data is not sent for a certain period of time, after which the connection is closed. All of these fixes remove the close threshold behavior from these probe-resistant proxies, though may still be observable due to their timeout behavior.

4.2.8.2 Future Work

Looking forward we believe that there are several avenues for extending this work to strengthen both the attack and defenses.

First, enriching the set of probes that we use could help to elicit responses from more non-proxy endpoints, ultimately improving our attack. To discover or even automatically create new probes, we could extract data from live connections in our ISP tap. Watching what data is sent in connections could allow us to infer the protocols being used, allowing us to synthesize additional probes for those protocols. Doing so requires overcoming privacy challenges, ensuring that private information is not inadvertently collected and then sent in probes to other endpoints.

As another improvement to active attacks, future work could investigate the other ports that are open on a suspected proxy host. For instance, many hosts in our ZMap dataset responded as open on all TCP ports when scanned using Nmap, a common firewall behavior that is intended to thwart active scanning. However, our detector could be extended to collect and use this data as additional information in identifying proxies.

4.2.8.3 Long-term defenses

Section 4.2.6 details and evaluates immediate small changes that existing probe-resistant proxies can make to help address our immediate attacks. However, solving this problem fully in the long term may require rethinking the overall design of probe-resistant proxies.

Wang et al. [153] categorize circumvention techniques into three categories: Randomizing, Protocol Mimicry and Tunnelling.

Randomizing transports attempt to hide the application-layer fingerprints and essentially have "no fingerprint" by sending messages of randomized size that are encrypted to be indistinguishable from random bytes. All probe-resistant transports we cover in this paper fall into this category. These transports detect probing by testing the client's knowledge of a secret, and do not respond if it fails. As we show, not responding to any probes is rare and a fingerprint itself, that probing censors could use to block such transports.

Both Protocol Mimicry and Tunnelling approaches attempt to make traffic look like it belongs to a certain protocol or an application, but with an important distinction: Mimicry involves implementing a look-alike version of the protocol, while Tunnelling leverages an existing popular implementation of a protocol or application and tunnels circumvention traffic over it. Prior work [71] has shown that Protocol Mimicry is difficult, due to the complexity of features in popular implementations. Protocol Mimicry transports like SkypeMorph [106] attempt to respond to censor probes with realistic common responses, but minor differences with a target protocol or popular implementation can yield unique fingerprints that censors can use to block [64, 71]. Thus, Protocol Mimicry transports are difficult to use in probe-resistant proxies.

On the other hand, Tunnelling protocols like DeltaShaper [9], Facet [94], and CovertCast [104] tunnel circumvention traffic over existing implementations and services. Censors that probe these servers receive responses from legitimate implementations, making them difficult to distinguish from benign (non-proxy) services. There have been several Tunnelling protocols proposed in the literature [9, 94, 104] and deployed in practice [54] demonstrating their strong potential. Tunnelling transports can defend against active probing by looking like legitimate services, even evading censors that whitelist popular protocols [7]. While recent work has demonstrated the feasibility of detecting Tunnelling transports using machine learning [10], to the best of our knowledge, these techniques have yet to be employed by censors, possibly due to the challenging combination of false positive rates of the detection algorithms and base rates of legitimate traffic [153].

4.2.9 Acknowledgements

We wish to thank the many people that helped make this work possible, especially the University of Colorado IT Security and Network Operations for providing us access to the network tap used in this paper, and J. Alex Halderman for providing ZMap data. We also thank the many proxy developers we discussed this paper with and for providing proxies to test against, including Ox Cart at Lantern, Michael Goldberger and Rod Hynes at Psiphon, and Vinicius Fortuna and Ben Schwartz from Google Jigsaw (Outline). We are also grateful to Prasanth Prahladan for his initial discussion and participation on this work, and we thank David Fifield for his valuable comments, feedback, and suggestions on the paper in several drafts.

4.2.10 Conclusion

Probe resistance is necessary for modern proxy implementations to avoid being blocked by censors. In this work, we demonstrate that existing probe-resistant strategies are generally insufficient to stop censors from identifying proxies via active probes. We identify several low-level choices that proxy developers make that leave them vulnerable to active probing attacks. In particular, proxy servers reveal developer choices about when connections are closed in terms of timing or bytes read, allowing censors to fingerprint and differentiate them from other non-proxy servers.

We evaluate the effectiveness of identifying proxies by probing endpoints collected from both passive tap observations and active ZMap scans, and find that our attacks are able to identify most proxies with negligible false positive rates that make these attacks practical for a censor to use today. Leveraging our datasets, we make recommendations to existing circumvention projects to defend against these potential attacks.

4.2.11 Automated Proxy Classification

While we used manually-crafted decision trees to distinguish proxies from common (non-proxy) endpoints, we also created automatically-generated decision trees. In this section, we present the challenges in using machine learning in this context, and compare the results to our manual approach.

We start by filtering our Tap and ZMap datasets to exclude endpoint samples that respond with data to our probes, as they are trivial to classify as non-proxy results, even without knowing any details of probe-resistant proxy protocols. This leaves 25k samples from our Tap dataset (776k from ZMap) that we use for training and testing our automated classifier.

Even after removing these trivially classified samples, our datasets remain extremely unbalanced as we have only a handful of positive proxy samples (compared to tens or hundreds of thousands of non-proxy samples). To address this imbalance, we synthesize proxy samples based on our understanding from manual inspection of their source code. To simulate network measurement, we add a random 20–500 milliseconds of latency to the timeouts specified by the proxy's server timeout code. We emphasize that while necessary to balance our datasets, understanding proxy behavior at this level of detail is already sufficient to create the manual decision trees.

We must be careful about the data imbalance in our dataset, as we synthesize the samples. If we synthesize too many, proxies will be a large cluster in an otherwise heterogeneous population. Otherwise, if we synthesize too few, the tree will overfit to the small sample of proxies and not generalize. We chose to generate the amount of proxy samples equal to 1% of the amount of not-proxy samples. As a result, for each proxy label we generate a total of 258 samples for Tap dataset, and 7767 samples for ZMap dataset. This helps to balance the dataset while still conveying that proxies are relatively uncommon on the Internet.

We then trained and tested an automated multi-class decision tree on our ZMap and Tap datasets, including in each the synthetic samples we generated to balance the datasets. We also build a manual multi-class classifier based on the conditions from Figures 4.3-4.8 for each proxy label, and classify any unmatched samples as not proxies. Since all proxies have mutually exclusive trees, we can check them in any order. The resulting accuracy for our automated decision tree is shown in Table 4.5. To evaluate our

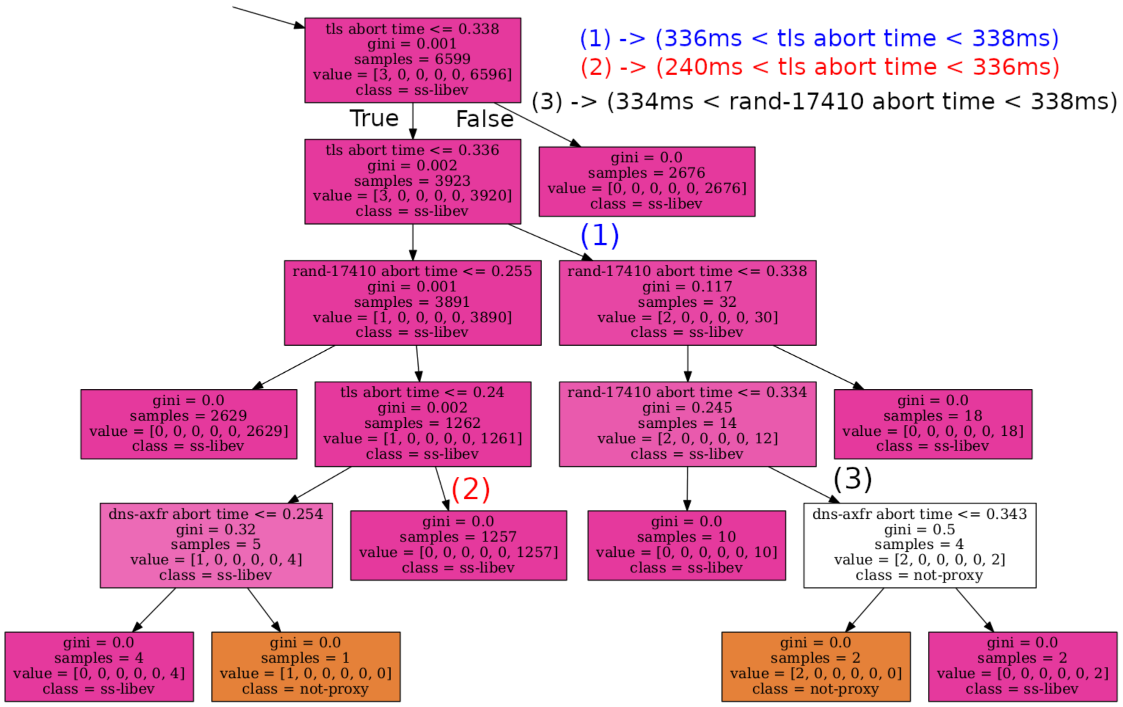

(1) -> (336ms < tls abort time < 338ms)
(2) -> (240ms < tls abort time < 336ms)
(3) -> (334ms < rand-17410 abort time < 338ms)
tls abort time <= 0.338 gini = 0.001 samples = 6599 value = [3, 0, 0, 0, 0, 6596] class = ss-libev
True
False
tls abort time <= 0.336 gini = 0.002 samples = 3923 value = [3, 0, 0, 0, 0, 3920] class = ss-libev
gini = 0.0 samples = 2676 value = [0, 0, 0, 0, 0, 2676] class = ss-libev
(1)
rand-17410 abort time <= 0.255 gini = 0.001 samples = 3891 value = [1, 0, 0, 0, 0, 3890] class = ss-libev
rand-17410 abort time <= 0.338 gini = 0.117 samples = 32 value = [2, 0, 0, 0, 0, 30] class = ss-libev
gini = 0.0 samples = 2629 value = [0, 0, 0, 0, 0, 2629] class = ss-libev
tls abort time <= 0.24 gini = 0.002 samples = 1262 value = [1, 0, 0, 0, 0, 1261] class = ss-libev
rand-17410 abort time <= 0.334 gini = 0.245 samples = 14 value = [2, 0, 0, 0, 0, 12] class = ss-libev
gini = 0.0 samples = 18 value = [0, 0, 0, 0, 0, 18] class = ss-libev
(2)
(3)
dns-axfr abort time <= 0.254 gini = 0.32 samples = 5 value = [1, 0, 0, 0, 0, 4] class = ss-libev
gini = 0.0 samples = 1257 value = [0, 0, 0, 0, 0, 1257] class = ss-libev
gini = 0.0 samples = 10 value = [0, 0, 0, 0, 0, 10] class = ss-libev
dns-axfr abort time <= 0.343 gini = 0.5 samples = 4 value = [2, 0, 0, 0, 0, 2] class = not-proxy
gini = 0.0 samples = 4 value = [0, 0, 0, 0, 0, 4] class = ss-libev
gini = 0.0 samples = 1 value = [1, 0, 0, 0, 0, 0] class = not-proxy
gini = 0.0 samples = 2 value = [2, 0, 0, 0, 0, 0] class = not-proxy
gini = 0.0 samples = 2 value = [0, 0, 0, 0, 0, 2] class = ss-libev

Figure 4.14: **Overfit Subtree** — Our automated decision tree (trained on our ZMap dataset with synthetic samples) shows evidence of overfitting. At the root node of this subtree, there are 1171 Shadowsocks and 4 non-proxy samples left to classify. Rather than deciding on something inherent to the Shadowsocks protocol, the classifier divides samples based on response latency differences extrinsic to the TLS probe (TLS abort time). Parts of the tree divide samples into endpoints that responded to our TLS probe between 336 and 338 milliseconds, and to our rand-17410 probe between 334 and 338 milliseconds. We confirmed none of these times are intrinsic to the Shadowsocks implementations.

	Evaluated on	
Trained on	ZMap	Tap
ZMap	0.99959	0.88180
Tap	0.99017	0.98910
manual	0.99962	0.88386

Table 4.5: **Accuracy of Decision Trees** — We evaluated accuracy of our manual and automated decision trees, trained on our ZMap and Tap datasets (including synthetic samples). We excluded endpoints that respond with data to any of our probes, yielding 25k samples from our Tap dataset and 776k from the ZMap dataset. We used 5-fold cross-validation to train and test the automated decision trees when training and evaluating on the same dataset. The majority of inaccuracies for both automated and manual decision trees stem from misclassifications of MTProto.

automated decision tree on datasets it has not seen before, we train on one dataset (Tap or ZMap), and test on the other (ZMap or Tap). We also use 5-fold cross-validation when we train and test using subsets of the same dataset, partitioning the set into distinct subsets for training and testing.

There are 3,233 (12.5%) non-proxy endpoints in our Tap learning dataset that never close the connection[5] . This behavior is shared by MTProto, and the decision whether or not to classify those endpoints as MTProto affects accuracy the most for the Tap datasets. The manual decision tree and automated classifier learned on ZMap data both classify these endpoints as MTProto, while the automated decision tree learned on Tap data classified them as not-proxies. The high number of MTProto-like samples in the non-proxy Tap dataset and the Tap-trained tree's decision to classify them as non-proxies explains the higher accuracy of the decision tree that was trained and 5-fold cross-validated on Tap data. To provide a full summary of both correct and incorrect predictions that our manual and automated decision trees made, we include their Confusion Matrices in Table 4.6. Nonetheless, in all other cases, our manual decision tree slightly out-performs the automated decision tree. We present our automated multi-class decision tree trained on our Tap (and synthetic data) dataset in Figure 4.15.

In order to evaluate relative importance of each individual feature, we compute the Mutual Information [85, 130] between our features and target labels, and list this in Figure 4.7, located in Appendix. This metric does not depend on the classifier, and is only a function of features and their labels. We find that abort time (the time it takes an endpoint to respond to a particular probe) is the most important feature for all of our

[5] or do so after our 300 second probing utility timeout

Predicted as	Actual label					
	not-proxy	lampshade	mtproto	obfs4	ossh	ss-python
not-proxy	776421	0	0	0	0	0
Lampshade	1	7767	0	0	0	0
MTProto	296	0	7767	0	0	0
obfs4	0	0	0	7767	0	0
OSSH	0	0	0	0	7767	0
ss-python	8	0	0	0	0	7767

(a) manual, tested on ZMap dataset

Predicted as	Actual label					
	not-proxy	Lampshade	MTProto	obfs4	OSSH	ss-python
not-proxy	22721	0	0	0	0	0
Lampshade	0	258	0	0	0	0
MTProto	3145	0	258	0	0	0
obfs4	2	0	0	258	0	0
OSSH	8	0	0	0	258	0
ss-python	0	0	0	0	0	258

(b) manual, tested on Tap dataset

Predicted as	Actual label					
	not-proxy	Lampshade	MTProto	obfs4	OSSH	ss-python
not-proxy	22665	0	0	0	0	0
Lampshade	2	258	0	0	0	0
MTProto	3182	0	258	0	0	0
obfs4	19	0	0	258	0	0
OSSH	8	0	0	0	258	0
ss-python	0	0	0	0	0	258

(c) automated, trained on ZMap, tested on Tap

Predicted as	Actual label					
	not-proxy	Lampshade	MTProto	obfs4	OSSH	ss-python
not-proxy	776663	8	7767	24	122	31
Lampshade	11	7759	0	0	0	0
MTProto	0	0	0	0	0	0
obfs4	25	0	0	7743	0	0
OSSH	6	0	0	0	7645	0
ss-python	21	0	0	0	0	7736

(d) automated, trained on Tap, tested on ZMap

Table 4.6: **Confusion Matrices** — Confusion matrices for our manual and automatically-generated decision trees for both our Tap and ZMap datasets. We see good performance overall for identifying proxies, though note all classifiers struggle to distinguish MTProto, due to the ubiquity of endpoints that have no connection timeout.

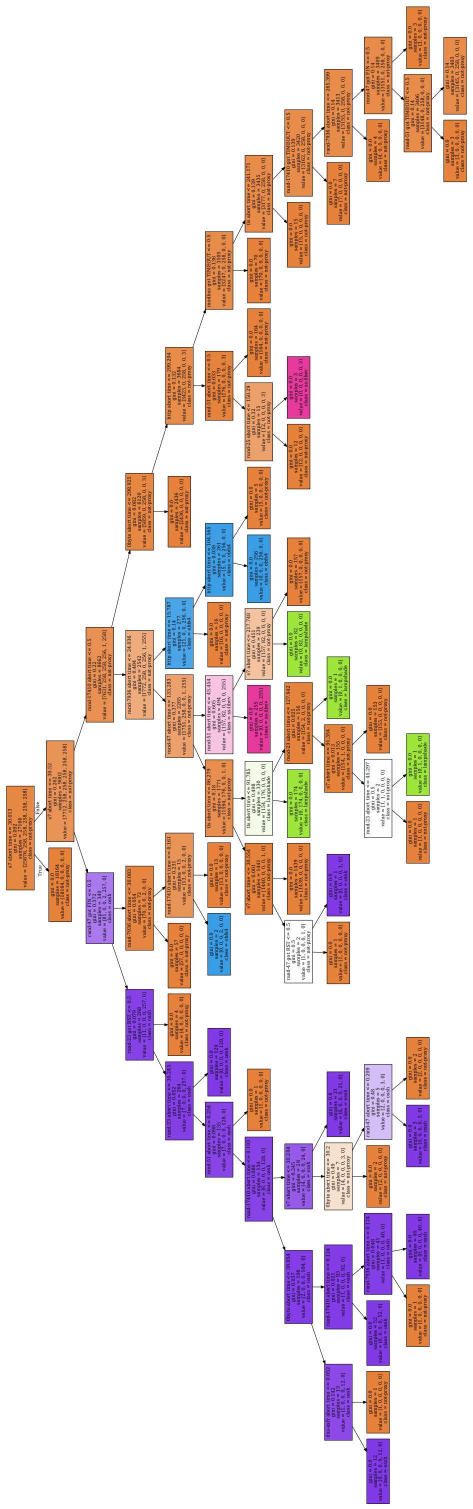


Figure 4.15: **Automated Decision Tree** — We trained a multi-class decision tree classifier on the 25k endpoints in our Tap dataset (and 1k synthetically-generated proxy samples). Each node contains an array of the number of samples yet to be classified at that point in the tree: [not proxy, lampshade, mtproto, obfs4, ossh, Shadowsocks-python]. Nodes are labeled with the current classification, and colored according to how confident a decision could be made at that point.

probes (as opposed to feature indicating whether the probe got FIN, RST or timed out), with S7, rand-23 and Modbus abort times being most important in our Tap dataset, and rand-51, HTTP, DNS-AXFR abort times being more important in the ZMap dataset.

We conclude that automated decision trees may be a viable way to allow proxy developers to quickly test if their servers have responses that stand out, but are no more accurate than our manually created decision trees, while requiring no less manual labor. We note our manually created trees have the advantage that they were built using only domain knowledge of the specific proxies; any updates to proxies can be directly encoded. On the other hand, our "automated" decision trees will need to be provided both updated domain knowledge (for synthetic samples), and also retrained if even unrelated non-proxy traffic changes; the "automated" decision tree also needs to be retrained if only one of the proxies change behavior.

Tap dataset	Feature	ZMap dataset
0.1761	S7 abort time	0.2681
0.1725	rand-23 abort time	0.2704
0.1696	Modbus abort time	0.2711
0.1419	0byte abort time	0.2386
0.1355	rand-47 abort time	0.2558
0.1340	rand-25 abort time	0.2545
0.1320	STUN abort time	0.2556
0.1285	rand-51 abort time	0.2831
0.1279	HTTP abort time	0.2835
0.1268	dns-axfr abort time	0.2821
0.1190	TLS abort time	0.2740
0.0868	rand-7936 abort time	0.2591
0.0582	rand-17410 abort time	0.2387
0.0399	rand-47 got TIMEOUT	0.0339
0.0397	rand-25 got TIMEOUT	0.0339
0.0397	rand-23 got TIMEOUT	0.0339
0.0392	S7 got TIMEOUT	0.0340
0.0372	STUN got TIMEOUT	0.0340
0.0362	TLS got FIN	0.1059
0.0359	Modbus got TIMEOUT	0.0339
0.0334	S7 got FIN	0.1057
0.0332	rand-51 got FIN	0.1057
0.0332	rand-17410 got RST	0.0341
0.0332	rand-23 got FIN	0.1057
0.0332	HTTP got FIN	0.0645
0.0328	dns-axfr got FIN	0.1055
0.0326	rand-47 got FIN	0.0742
0.0326	rand-25 got FIN	0.0742
0.0324	STUN got FIN	0.0742
0.0324	rand-7936 got RST	0.0323
0.0324	0byte got TIMEOUT	0.0341
0.0321	0byte got FIN	0.1041
0.0321	rand-7936 got FIN	0.0819
0.0320	Modbus got FIN	0.1057
0.0279	TLS got RST	0.0297
0.0268	HTTP got RST	0.0264
0.0266	rand-51 got RST	0.0298
0.0266	rand-47 got RST	0.0298
0.0266	rand-25 got RST	0.0298
0.0263	rand-17410 got FIN	0.0446
0.0262	dns-axfr got RST	0.0298
0.0259	STUN got RST	0.0297
0.0242	HTTP got TIMEOUT	0.0353
0.0242	dns-axfr got TIMEOUT	0.0351
0.0241	rand-51 got TIMEOUT	0.0351
0.0240	rand-17410 got TIMEOUT	0.0352
0.0240	rand-7936 got TIMEOUT	0.0352
0.0238	TLS got TIMEOUT	0.0354
0.0210	S7 got RST	0.0270
0.0206	rand-23 got RST	0.0271
0.0079	Modbus got RST	0.0270
0.0053	0byte got RST	0.0265

Table 4.7: **Mutual Information of Features** — Mutual Information is a measure of the mutual dependence between two variables. In this case, Mutual Information measures the dependence between features and labels, effectively quantifying the information gained from any specific feature. Abort time (the time it takes an endpoint to respond to a particular probe) is the most important feature for all of our probes, with distinguishing probes like S7 and Modbus filtering out many embedded devices.

4.3 HTTPT: A Probe-Resistant Proxy

4.3.1 Introduction

Internet censors strive to detect and block circumvention proxies. Many censors have adopted sophisticated proxy discovery techniques, such as **active probing** [47, 75, 159], where the censor connects to suspected proxy servers and sends probes designed to distinguish them from non-proxies. In response to this attack, developers designed and built **probe-resistant** proxies, such as ScrambleSuit [162], obfs4 [147], and Shadowsocks [137] which require clients to prove knowledge of a secret key (obtained out of band) before the proxy will respond to them.

While probe-resistant proxies are harder for censors to identify, recent work demonstrates there are still ways censors could detect probe-resistant proxies [61]. For instance, not responding to common protocols like HTTP or TLS is unusual behavior for servers. Recently, China's GFW has been observed using similar active probing techniques to detect and block Shadowsocks [6].

In this work, we propose HTTPT, an alternative proxy architecture that is designed to defend against advanced active probing. While previous probe-resistant designs avoid looking like any protocol, we argue this approach is fundamentally limited, as this behavior is unusual for servers on the Internet [61]. Instead, HTTPT uses HTTPS, a ubiquitous protocol with many diverse implementations, providing a heterogeneous set of behaviors to blend in with. HTTPT can be deployed behind already-existing Web Servers, making it impossible for censors to access and identify the proxy without knowing the secret necessary to use it.

4.3.1.1 Benefits

HTTPT has several benefits over existing designs:

Replay attack protection Existing probe-resistant proxies are vulnerable to replay attacks, where the censor re-sends observed client messages. Some proxies, such as Shadowsocks-libev [166], implement a cache to prevent replays of previous connections. However, China's active probing thwarts this defense by permuting replays [6], and the behavior of not responding at all to replays may be unusual. In contrast, HTTPT is immune to replay attacks due to its reliance on TLS, which includes bidirectional nonces in the handshake.

Using existing web servers HTTPT leverages existing web servers, making it more difficult for censors to single out or identify. Hiding a proxy behind an authentic web server, such as Apache or nginx, obviates the need to mimic TLS or any other server fingerprints [62] that could allow censors to block it.

Overhead After the initial TLS handshake, HTTPT's overhead is minimal. To avoid the overhead associated with sending HTTP-compatible encodings, HTTPT instead uses WebSockets between client and server. This allows the client and server to send binary data over the web server, obviating the need for costly HTTP-safe encodings such as MIME or base64.

Using a popular protocol Existing probe-resistant proxies rely on randomized protocols that try to blend in with generic Internet traffic and make it hard for censors to identify passively. However, prior work has shown that these protocols might still be detectable using entropy analysis and machine learning [10, 153]. In HTTPT, we rely on the popular TLS protocol to tunnel data, making it more difficult for censors to block outright (because TLS is popular), and hard to perform fingerprinting attacks due to the heterogeneity of the protocol.

4.3.2 Background

The arms race between censors and circumvention tools has led to new techniques from both sides. In this section, we provide background on the active probing attacks and defenses employed recently.

Active Probing To detect and block proxies, censors often use **active probing** attacks, where they connect to suspected proxy servers and attempt to communicate using known circumvention protocols. If a server responds to these probes using the protocol (e.g. they offer proxy service to the censor), the censor learns the server is in fact a proxy, and can block it. The Great Firewall of China (GFW) have used this technique for nearly a decade to find and block Tor bridges and other proxies [47, 75, 159].

Censors typically find suspected proxies by traffic analysis or Internet scanning [42]. For instance, the GFW has been observed to send probes to TLS (TCP port 443) servers when a client first connects to them [159]. These probing techniques include sending random data and attempting to perform a Tor handshake. If the server responds like a proxy, for instance by responding with the Tor protocol, the endpoint is blocked by the censor.

Probe-resistant proxies To resist active probing attacks, circumvention tools have developed **probe-resistant** proxy protocols, such as ScrambleSuit [162], obfs4 [147], Shadowsocks [137], and Lampshade [117]. These protocols require clients to prove knowledge of a shared secret before the server will respond. This shared secret is distributed among the clients through private channels, such as Tor's BridgeDB [148] or email, and without the knowledge of the secret, censors are unable to get responses to their probes. As these proxies effectively remain silent to such probes, it is difficult for censors to confirm if they are proxies.

Detecting probe-resistant proxies However, there are still ways censors can identify probe-resistant proxies. In 2020, Frolov et al. showed how not responding to any probes is uncommon behavior online: over 94% of servers responded with data to at least one popular protocol [61]. Furthermore, circumvention servers would often have unique timeouts or data limits before they closed connections, allowing censors to identify and even fingerprint probe-resistant proxies that never respond to the censor. For example, Lampshade reads 256 bytes from the client, and closes the connection immediately if it does not demonstrate knowledge of the secret (e.g. a censor sends random data). However, if less than 256 bytes is read, the server will wait for 90 seconds, and then timeout and close the connection.

This gives censors a new strategy for identifying probe-resistant proxies: send several probes for popular protocols and random data, and identify the servers whose responses are consistent with responses of a given proxy (e.g. close immediately for probes over 256 bytes or close after 90 seconds otherwise for Lampshade).

Another potential way censors can identify probe-resistant proxies is with replay attacks, where they replay a previous legitimate client's initial message to the server. Since this message proves knowledge of the secret, the server will respond, alerting the censor it is a proxy.

Recently, the GFW has been observed using replay attacks and the aforementioned timeout fingerprinting to identify and block Shadowsocks servers [6]. When a client connects to a server and sends a Shadowsocks-sized initial message, the GFW will send several follow-up probes, including replays of the client's message, and random-data probes of various sizes up to and exceeding the 50-byte data limit of

Shadowsocks.

Probe-resistant proxies could try to prevent replay attacks using server randoms or challenges, where the client proves knowledge of the shared secret by hashing it with server-provided (random) data. However, this would require the server send data before the client has authenticated, which could be used by censors to identify those proxies.

In contrast, web servers provide a natural defense against replays due to the security properties of TLS. TLS defends against client replay attacks by including the server random in the key agreement, ensuring each connection uses unique cryptographic keys.

TLS also has a large number of popular implementations, providing a plethora of fingerprints to blend into [62]. While previous work has shown the difficulty in mimicking existing protocols for circumvention [71], many of these protocols only had a single used implementation, making it difficult for circumvention tools to copy perfectly. In contrast, our technique leverages **existing** TLS implementations and efficiently tunnels through them, avoiding the difficult task of mimicry altogether. While prior work has tunneled circumvention traffic through other protocols, including chat applications, Skype, or video streaming [9, 94, 104, 106], these protocols are generally less popular than TLS and used in a narrow set of applications. This allows censors to use traffic analysis to find proxy use [64]. In contrast, the wide range of applications used by TLS makes it more difficult (though not impossible) to perform this kind of analysis.

4.3.3 Design

At a high level, HTTPT functions as a TLS server that also has secret proxy functionality. The proxy function can only be accessed by clients that know a secret key, distributed out of band. If a censor attempts to probe an HTTPT server, they will receive the TLS server's benign response, making it difficult to identify the server as a proxy. We first describe the challenges in crafting benign responses in Section 4.3.3.1 and several solutions in Section 4.3.3.2. Finally, we describe how clients can access the proxy functionality in Section 4.3.3.3.

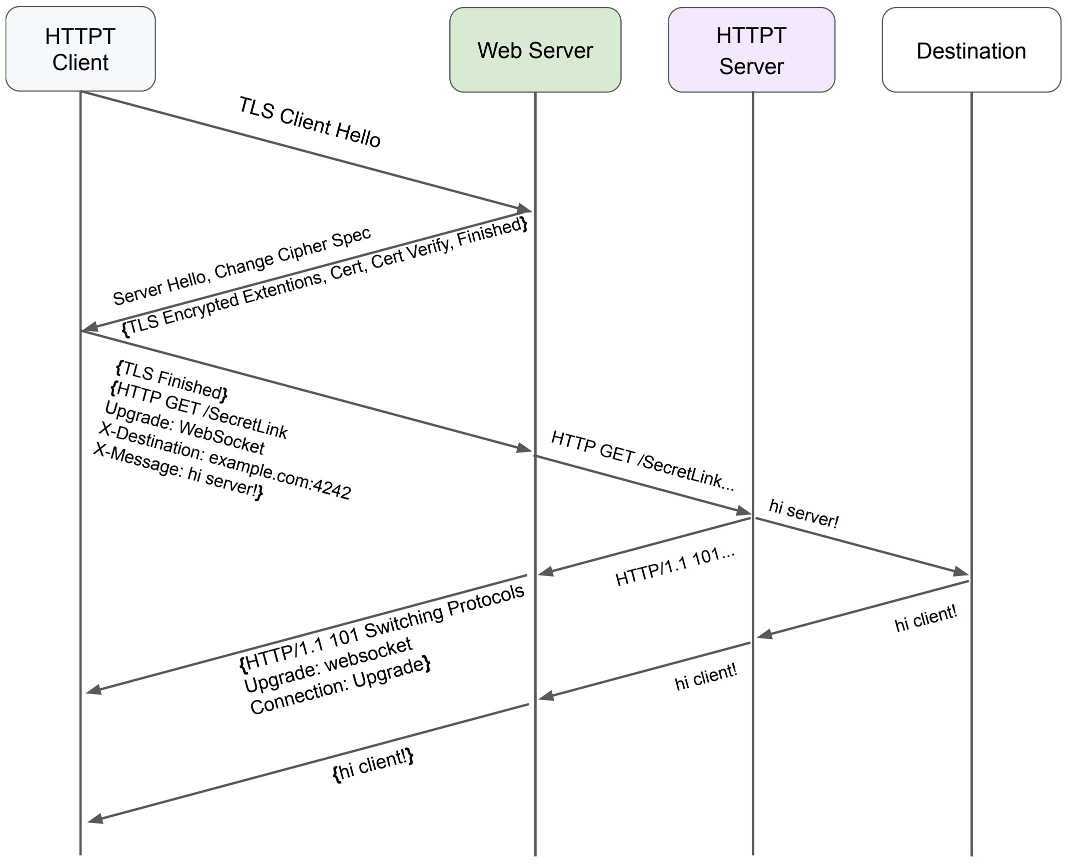


Figure 4.16: **HTTPT Handshake** — HTTPT minimizes Time To First Byte by passing application data together with the first HTTP request. Following initial handshake, the traffic passes through the proxy with relatively small overhead, associated with encrypting all traffic between HTTPT client and the web server using TLS. Curly brackets denote messages in encrypted TLS Records. Overhead per TLS Record amounts to only 5 bytes of the TLS header frame, and the message authentication code, which is 16 bytes for most common MACs, such as Galois MAC and Poly1305.

4.3.3.1 Benign Response Challenges

When probed by a censor, the HTTPT server must respond in a normal-seeming way that does not identify it as a proxy. There are several features of TLS servers that might be used to identify HTTPT servers.

TLS implementation TLS servers (and clients) can be **fingerprinted** based on identifying features they send in the TLS handshake. For instance, the cipher suites or extensions they support or use may vary by implementation, and those differences have been used by censors previously to identify proxies [49, 62]. Thus, in HTTPT we must be careful to avoid having identifying characteristics in the TLS layer. On the server side we address this by using existing popular TLS implementations and servers, obviating the need to mimic them. Our client implementation currently uses uTLS [62] library to mimic popular TLS fingerprints; alternatively, we may use a popular browser to establish TLS connections.

TLS certificate A related feature is the certificate the server sends. TLS certificates contain a public key, one or more domain names, a CA signature, and other extensions. To avoid detection, the HTTPT server must respond with a realistic certificate. This is challenging because real certificates often contain legitimate domain names and are usually signed by trusted certificate authorities, preventing us from forging arbitrary CA-signed certificates for domains we do not control. Self-signed certificates could be used instead, allowing us to construct arbitrary values. However, self-signed certificates may be easier for censors to block, since websites, that are popular and therefore their blocking may cause collateral damage, use CA-signed certificates.

TLS Content Once a TLS connection is established and the client (or censor) sends a request, the HTTPT server must respond with benign data, such as an HTTP response with an HTML payload. The contents of this payload must be carefully chosen so that censors cannot identify HTTPT servers based on their responses. Ideally, this content appears to (or actually does) provide real-value to legitimate users, so that censors have a harder time blocking the service. For instance, while a blank page may be common to non-HTTPT servers, it does not carry significant collateral damage that would discourage a censor from blocking it.

4.3.3.2 Benign Responses

We propose several approaches that allow HTTPT to make the TLS certificate and content appear innocuous to probing censors. If HTTPT servers are able to utilize multiple approaches, censors must find ways to identify and block **all** of these variants to prevent its use. If any of these schemes are successful at evading detection, HTTPT will still be able to provide circumvention value to users.

Existing servers One option is to place HTTPT proxies at already existing HTTPS servers run by volunteers. These servers already have valid certificates and content that can be returned to a probing censor, providing legitimate camouflage that the HTTPT proxy hides behind. Because HTTPT proxies don't change the behavior of the underlying website, the censor will not be able to determine based on certificates or content if the site is a proxy or not.

Existing servers provide an ideal deployment for HTTPT, particularly if they are popular. We discuss strategies for recruiting existing sites to operate HTTPT proxies in Section 4.3.5.1.

Mask sites

Alternatively, HTTPT can adopt an idea from Conjure [60] and mimic existing servers not controlled by the HTTPT proxy admin. In this configuration, HTTPT will transparently proxy all traffic to a dedicated "mask site", which is an external TLS server. When a censor makes a connection to the HTTPT server, their packets are relayed to the mask site, and the responses are sent back to the censor. This allows an HTTPT proxy admin to have their server mimic any existing site online, in both certificate and content. We describe how the HTTPT proxy can detect the secret from the client in Section 4.3.3.3. Censors unable to prove knowledge of the secret will continue to have their traffic transparently relayed to the mask site, making it appear as if the HTTPT server is in fact the mask site.

Common error pages

For content, HTTPT servers could serve a benign error page and respond with 4xx or 5xx HTTP status code. Censys [40] scans of IPv4 HTTPS [23] servers demonstrate that these status codes are common online: as of June 2020 over 21% of the servers probed for `/` responded with `400 Bad Request`, 11.19% responded with `403 Forbidden`, 8.62% with `404 Not Found`, and 2.91% with `401 Unauthorized`.

Status Line	Hosts	
200 OK	19,887,860	48.78%
400 Bad Request	8,594,330	21.08%
403 Forbidden	4,560,297	11.19%
404 Not Found	3,515,282	8.62%
401 Unauthorized	1,187,145	2.91%
503 Service Unavailable	413,175	1.01%
Other	2,611,261	6.4 %

Table 4.8: **Common Status Lines** — Most common Status Lines of HTTP responses received during an internet-wide scan of IPv4 HTTPS servers on port 443 performed by Censys. Only half of the hosts scanned have responded with `200 OK`, demonstrating that a proxy does not need to serve a website with original content to blend in. Data fetched on June 8th, 2020 from `https://censys.io/ipv4/report?field=443.https.get.status_line.raw`

Copying Content Content could also be copied directly from existing websites, providing a defense similar to mask sites. Unlike mask sites, hosting the content directly avoids delays in TCP/TLS packets that would be present when transparently proxying at the packet level. However, copying content has several downsides. First, it may violate copyright law, making HTTPT servers a target from domestic legal threats as well as from censors. Second, it still requires servers to provide a plausible certificate (and potentially a domain) for this content, which may be difficult to pair with already-existing content.

Restricted Access Finally, we can avoid having to send content altogether by restricting access to it. For instance, requiring an HTTP authorization, or a login page, and returning generic errors or denial pages for all attempted passwords. Such prompts are common online, causing censors to hesitate to block such sites, even without being able to see the content.

These restrictions could also be implemented at the TLS level: clients that send an incorrect server name indication (SNI) could trigger a handshake alert—behavior shared with approximately 3% of endpoints online [61]. However, censors could still replay the correct SNI value from legitimate HTTPT clients, as it is sent in the clear in the ClientHello message. TLS 1.3 offers an encrypted SNI extension [124, 126] that would address this, but it has yet to see wide use, and there have already been mixed reports of censors blocking it [25, 63].

4.3.3.3 Proving knowledge of the secret

Legitimate clients must prove knowledge of a secret shared out-of-band to use the proxy. However, they must do so in a way such that incorrect guesses do not reveal to the client that the guess was even checked. For instance, if the server responded with an "invalid secret" response, a censor could use this to identify and block HTTPT proxies via probing.

HTTPT provides two ways for clients to prove knowledge of the secret.

Secret URL First, HTTPT clients can include the shared secret in the URL of the initial HTTP request as an authentication mechanism. The web server is configured to reverse proxy all requests visiting the secret phrase to the HTTPT server. This allows the web server to respond with its normal error page (e.g. 404 Not Found) for incorrect guesses from a censor, indistinguishable from a non-proxy web server. This method works for existing sites, where the HTTPT admin can easily configure the site.

In-band TLS When using mask sites, the HTTPT server is not able to directly observe the URL that the client visits. Instead, we adopt a technique from Conjure [60] to covertly detect legitimate users. During connection establishment, the HTTPT server proxies all traffic transparently to the mask site, allowing the HTTPT client to complete a normal TLS handshake with the mask site. This means the HTTPT server appears to be sending certificates as the mask site, though does not have control over the private key and thus cannot decrypt or inject data into this connection.

After the TLS handshake completes, the HTTPT client switches to a new Master Secret, derived from the HTTPT shared secret combined with the client and server random values in the mask site TLS connection to prevent replay attacks. The HTTPT server also derives this new master secret, and if it is able to decrypt the client's application data, it knows the client is authentic (i.e. knows the shared secret). Otherwise, the HTTPT server continues forwarding packets to the mask site.

4.3.4 Evaluation

4.3.4.1 Implementation

We implemented a prototype HTTPT client and server in Golang. Our prototype is relatively simple: our client and server are written in 163 and 158 lines respectively. We confirmed our prototype is compatible with several popular web servers, including Apache 2.4.43, nginx 1.16.1, and Caddy 2.1.0. We currently support existing sites configured in any of these servers, which allows us to support all of the schemes in Section 4.3.3.2 except mask sites.

Figure 4.17 gives an overview of the components in the HTTPT system. Our implementation focuses on minimizing **framing overhead** between the web server and the HTTPT server. HTTPT clients make a normal TLS connection with the web server, and send an HTTP request with the secret URL and a WebSocket upgrade header. The web server (e.g. Caddy) is configured to reverse proxy this request to the HTTPT server WebSocket application based on the URL and the HTTPT server confirms the WebSocket upgrade. At this point, the web server simply forwards all application traffic transparently between the client and the HTTPT server. While normally WebSockets contain a small framing overhead, we found that this was not necessary for the web servers we tested, allowing us to have no framing between HTTPT client and server after the initial WebSocket upgrade.

This means any bytes the client sends will be encapsulated in TLS, decrypted by the web server, and forwarded directly on to the HTTPT server. Over this transport, HTTPT could support any proxying protocol, such as SOCKS. We choose to implement HTTP proxying, allowing clients to connect to and transparently communicate with an arbitrary endpoint over the HTTPT tunnel.

4.3.4.2 Performance

We evaluate the performance of HTTPT by comparing it with Shadowsocks. Shadowsocks is designed to be lightweight: it does not add padding and eliminates round trips in the initial connection, which provides exceptional performance, but with no forward secrecy.

We launched Shadowsocks-libev 3.1.3 and HTTPT clients in a VM in Karnataka, India, a Shadowsocks and HTTPT proxy server in Oregon, USA, and had the client connect to a "covert" destination web server in Virginia, USA via its respective proxy. The round-trip latency between the proxy client and server VMs is around 230ms, and the latency between the proxy server and the destination server is 80 ms.

We perform 100 measurements of Time To First Byte for Shadowsocks, and for two configurations of HTTPT, one using TLS 1.2, and another TLS 1.3, and show results in Figure 4.18. Shadowsocks provides faster latency by having fewer round trips in its handshake: compared to Shadowsocks, TLS 1.2 requires 2 additional RTT, and TLS 1.3 requires 1 additional RTT. We note that it is possible to take advantage of TLS 1.3 Zero Round Trip Resumption to match Shadowsocks' TTFB, however, this feature sacrifices forward secrecy, and may be used by censors as a distinguisher due to its infrequency on the Internet, or allow them to execute replay attacks.

To evaluate bandwidth, we perform 25 measurements of the time it takes to fetch a 100-megabyte file from the covert destination through each proxy. As shown in Figure 4.19, the median time to fetch the file for Shadowsocks was 24.65 seconds, and the median time for HTTPT was 25.15 seconds (2% overhead), a difference explained by the extra round trips involved for HTTPT.

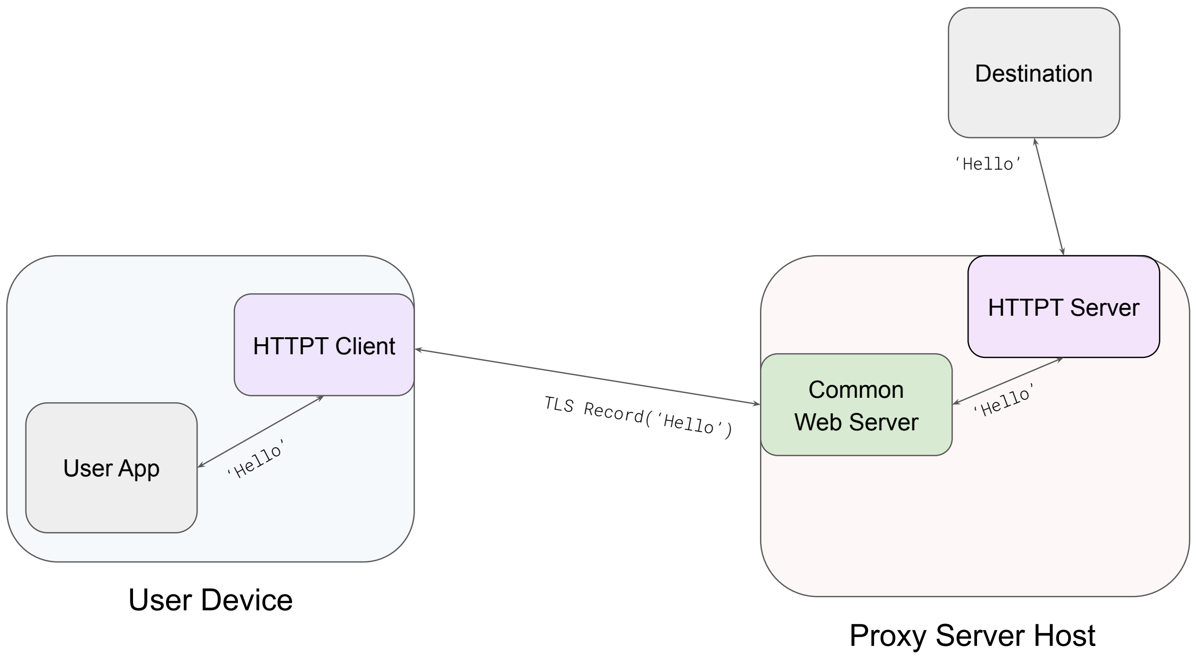


Figure 4.17: **HTTPT Overall Diagram** — In addition to client and server applications, present in most proxy designs, HTTPT adds an extra component: a Web Server used to deter active probing by providing plausible responses to the censors' probes.

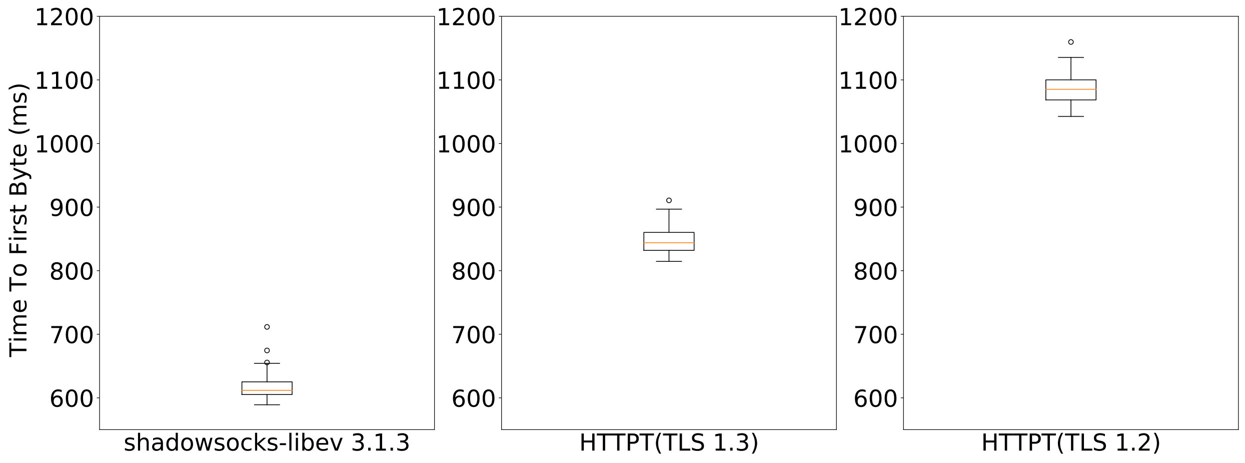


Figure 4.18: **Time To First Byte** — Having performed 100 measurements, we found that the median time to the first byte for Shadowsocks was 612 ms, 844 ms for HTTPT (TLS 1.3), and 1085 ms for HTTPT (TLS 1.2).

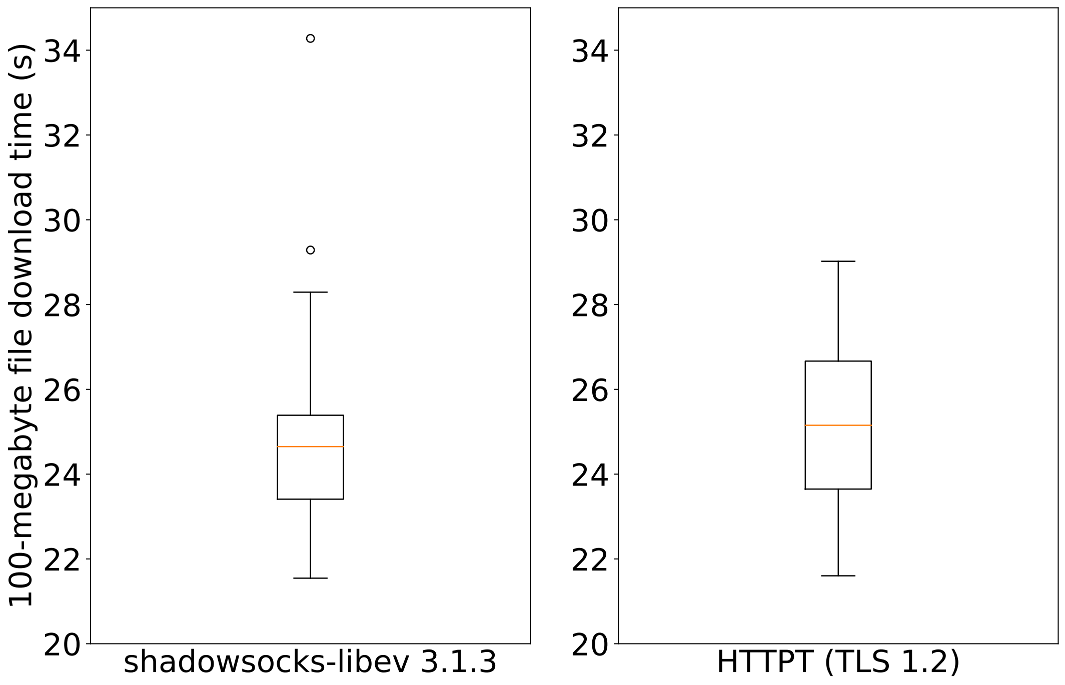


Figure 4.19: **Bandwidth Test** — For a bandwidth test, we download a 100-megabyte file 25 times. The resulting median time for Shadowsocks is 24.65 seconds, which is comparable to the HTTPT median time of 25.15 seconds.

Figures 4.18 and 4.19 demonstrate that HTTPT performance is comparable to that of Shadowsocks, and we argue that a single extra round trip of connection establishment time is an acceptable price to pay for forward secrecy and increased probe resistance.

4.3.5 Discussion

4.3.5.1 Deployment Incentives

In order for HTTPT proxies to be effective, they must be deployed to a wide range of heterogeneous servers. If the servers all share some fingerprint (e.g. follow a template or have certain features in their domain names), censors may be able to easily identify and block these proxies. We envision two likely modes of deployment of HTTPT.

First, circumvention tool developers could spin up their own proxies. This would likely involve using a combination of error-page or restricted access-type servers, as well as leveraging mask sites. Deployers in this model should be careful to deploy to multiple cloud providers to avoid identifying clusters of proxies based on IP or AS.

Second, volunteers could be encouraged to run HTTPT proxies on their existing sites. Websites that wish to support censored users could enable these proxies. We note it may be difficult to recruit volunteers, as ideally, they do not advertise they are supporting this effort. Nonetheless, sites that wish to support these efforts could deploy HTTPT proxies and have a direct impact. Additionally, if a large enough cohort of high-profile websites supports HTTPT proxies, then advertising support could still be possible, assuming censors are unwilling to collectively block these sites.

This second type of deployment would require coordination from a central party, such as a circumvention tool, that keeps track of what sites and secret URLs can be used as HTTPT proxies, and provides easy-to-install configurations to volunteers. Websites could also choose whether they are comfortable exiting proxy traffic directly from the web server (which increases user privacy), or if they want to relay such traffic to the central party (which limits abuse and liability concerns for the website).

4.3.5.2 Future Work

In the future, we plan to implement other benign response strategies (Section 4.3.3.2) such as mask sites, which will allow HTTPT to be used in an even wider range of scenarios.

In addition, there are several implementation features that we believe would benefit HTTPT. These include HTTP/2 [11], which could allow multiplexing of multiple covert connections over a single stream, or fine-grained padding control.

We also plan to adopt Turbo Tunnel [52], an inner reliability layer for circumvention protocols. This layer makes it easier to multiplex sessions over multiple overt connections in the event they are disconnected or disrupted.

4.3.5.3 Limitations

HTTPT is not designed to defend the proxy distribution system against enumeration attacks, where censors could discover IPs and secrets of proxies in bulk. Solutions such as Salmon [38] or Hyphae [96] could be used to defend against such attacks.

HTTPT also does not protect against website fingerprinting attacks. While prior research has shown machine learning classification may allow future censors to passively distinguish proxy use from other website access [67, 116], researchers have not yet observed these techniques used by censors in practice. This is likely due to the base-rate of non-proxy web traffic making it difficult for censors to justify using methods with even small (but non-zero) false positive detection rates [153]. Therefore, we believe that website fingerprinting is not an immediately likely attack against HTTPT, though it may pose one in the future. In any case, it should be possible to adapt existing fingerprinting defenses to our design [18, 22].

4.3.5.4 Related Work

The HTTPS protocol has been previously proposed in censorship circumvention technologies, including Domain Fronting [54] and Refraction Networking [59, 60, 72, 82, 164, 165]. However, these technologies are intended to protect open proxies against enumeration attacks, and have deployment requirements that constrain deployment. In contrast, HTTPT can be deployed at any web server and does not require ISP or CDN cooperation.

TLS has also been used in VPN proxies, such as OpenVPN [114]. However, these proxies do not address any of the benign response challenges that HTTPT solves, and are thus easily actively probed by censors.

4.3.6 Conclusion

Censors have recently adapted new methods to detect probe-resistant proxies [6, 61]. In this paper, we present HTTPT, which provides a defense against recent such active probe attacks employed by censors. By leveraging the existing popular TLS protocol, HTTPT avoids the bulk of attacks that existing probe-resistant proxies face. We address several new challenges that HTTPT introduces, and evaluate our prototype, finding that it is comparable to existing popular proxies used today. We plan to release our code as open source[6] .

While HTTPT is not the final word in the censorship arms race, we believe it presents a unique new challenge for censors to effectively detect at scale, making it well-suited to protecting circumvention tools from discovery.

[6] `https://github.com/sergeyfrolov/httpt`

Chapter 5

Conclusion

This dissertation makes several contributions to the state of circumvention art across three categories of attacks: Enumeration, TLS Fingerprinting, and Active Probing. During my research, I learned that censors tend to focus on simple practical attacks; therefore, it is more beneficial to look into such attacks and countermeasures against them. Proposing and analyzing practical attacks assists us in developing and testing defenses against them and helps us correctly prioritize our efforts, so we focus on the countermeasures against threats that are more likely to prove a real issue. Proposing practical defenses, that may be implemented in censorship circumvention tools, and utilized by censored Internet users, allows us to bring direct change to the state of anti-censorship in the real world.

I provided analytical evaluation of practicality of proposed attacks and countermeasures in relevant chapters. Another good indicator of practicality of the proposed attacks is a subsequent change in the behavior of those tools, while practicality of of the proposed countermeasures is their adoption in tools that help real users circumvent censorship. Here I document the changes to the tools, resulting from the proposed attacks, and the real-world adoption of the countermeasures:

Enumeration Attacks

- *Countermeasures.* We partnered with Psiphon to deploy TapDance and Conjure and deliver them to a large number of users. By the end of 2019, TapDance had been used [151] to serve upwards of 33,000 unique users per month.

TLS Fingerprinting

- *Attack.* The study found that the TLS fingerprints used by Psiphon, Snowflake, Lantern, and Signal are unique and, therefore, at the risk of blocking. It was also found that meek used a fingerprint that only accounted for 0.50% of traffic on our campus network, which is neither unique nor very rare; however, combined with other signals, meek's fingerprint could be used to reliably increase censors' certainty that a suspected connection is using meek. As a result, Psiphon, Lantern, and meek changed their TLS fingerprints.
- *Countermeasure.* A specialized library called uTLS has been built to help censorship circumvention tool developers defend against the attack. TapDance, Psiphon, Lantern, and meek eventually adopted this library.

Active Probing

- *Attack.* The newly-introduced active probing attack was able to reliably fingerprint the behavior of obfs4, Lampshade, OSSH, Shadowsocks-Outline, and Shadowsocks-Python. Developers of all the transports mentioned above, except Shadowsocks-Python, changed their transports' behavior as a result of my report.
- *Countermeasures.* Developers of affected transports changed their transports according to the suggested countermeasure to continue to read data from the client and not immediately close the connection after authentication failure. In a very recent paper published in August 2020, which was included in section 4.3 of the dissertation, I evaluate the use of web servers as a practical long-term defense, but it has not yet been adopted by the censorship circumvention tools.

As argued in section 4.3, HTTPS proxying is one of the most promising directions for future work. However, it may require additional protections against fingerprinting. We already made the first step of providing a countermeasure against TLS fingerprinting in section 3.2, but some issues remain unaddressed. Censors may be able to use packet sizes and timing to fingerprint a specific website, and determine that the content being served does not match the traces generated by the censorship circumvention tools; or, rather than fingerprint individual websites, detect weaker signals, indicative of circumvention tunnels in general and not natural HTTPS traffic.

Another important area of future work that would allow censorship circumvention tools to be used by wider audiences is usability. Work in this direction includes traditional User Experience research and development, as well as features like easy migration of a blocked proxy to another IP address with automated creation of a new VM and shutdown of the blocked one. Easy migration is available to users of Outline VPN [80], which provides options in its GUI to shut down the old proxy VM and to easily deploy a new one; however, it would be beneficial to create a generic framework, usable by a wide range of anti-censorship tools. It is also possible for a censorship circumvention client to detect that the server has been blocked, then issue a migration request and start using a new proxy without any involvement of the user, temporarily communicating over channels that are difficult to block, but may be higher in price or overhead, such as Domain Fronting or Refraction Networking.

Empirical measurements of real-world data allowed us to increase the efficiency of both attacks and defenses. I had access to a copy of traffic on the University of Colorado Boulder campus network, and used it for the benefit of circumvention; I also collected empirical data by scanning the Internet. For example, during the TLS Fingerprinting study, I collected anonymized TLS ClientHello messages from the University of Colorado Boulder campus network to measure the popularity of various implementations used in the real world and determine which censorship circumvention tools have rare fingerprints, and which fingerprints they should use instead to avoid the risk of blocking. When working on Active Probing attacks, we used both our campus tap and scanning of the Internet to find IP addresses of endpoints to be scanned, and then actively probed those endpoints. The use of real-world data will remain important in the future for the purpose of blending in with general traffic on the Internet. Real-world data can also assist censorship circumvention developers by providing visibility into censors' strategies as they evolve.

Bibliography

[1] Sadia Afroz and David Fifield. Timeline of Tor censorship. `http://www1.icsi.berkeley.edu/~sadia/tor_timeline.pdf`.

[2] Alexa Internet, Inc. Alexa top 500 global sites, 2019. (Visited on 02/15/2019). `https://www.alexa.com/topsites`.

[3] Nadhem J AlFardan, Daniel J Bernstein, Kenneth G Paterson, Bertram Poettering, and Jacob CN Schuldt. On the security of RC4 in TLS. In **USENIX Security Symposium**, pages 305–320, 2013.

[4] Bernhard Amann, Matthias Vallentin, Seth Hall, and Robin Sommer. Extracting certificates from live traffic: a near real-time SSL notary service. **Technical Report TR-12-014**, 2012.

[5] Blake Anderson, Subharthi Paul, and David McGrew. Deciphering Malware's use of TLS (without Decryption). **Journal of Computer Virology and Hacking Techniques**:1–17, 2016.

[6] Anonymous, David Fifield, and Amir Houmansadr. How China Detects and Blocks Shadowsocks. `https://gfw.report/blog/gfw_shadowsocks`.

[7] Simurgh Aryan, Homa Aryan, and J. Alex Halderman. Internet Censorship in Iran: A First Look. In **3rd USENIX Workshop on Free and Open Communications on the Internet (FOCI 13)**, Washington, D.C. USENIX Association, August 2013.

[8] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: breaking TLS with SSLv2. In **25th USENIX Security Symposium**, August 2016.

[9] Diogo Barradas, Nuno Santos, and Luís Rodrigues. DeltaShaper: enabling unobservable censorship-resistant TCP tunneling over videoconferencing streams. **Proceedings on Privacy Enhancing Technologies**, 2017(4):5–22, 2017.

[10] Diogo Barradas, Nuno Santos, and Luís Rodrigues. Effective detection of multimedia protocol tunneling using machine learning. In **27th USENIX Security Symposium**. USENIX Association, 2018.

[11] Mike Belshe, Martin Thomson, and Roberto Peon. Hypertext Transfer Protocol Version 2 (HTTP/2), 2015.

[12] David Benjamin. Applying GREASE to TLS Extensibility. `https://tools.ietf.org/html/draft-davidben-tls-grease-01`, September 2016.

[13] Daniel J Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In **Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security**, pages 967–980. ACM, 2013.

[14] Alexander Bersenev. Async MTProto proxy for Telegram in Python. `https://github.com/alexbers/mtprotoproxy`.

[15] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: taming the composite state machines of TLS. In **Security and Privacy (SP), 2015 IEEE Symposium on**, pages 535–552. IEEE, 2015.

[16] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-) security of 64-bit block ciphers: collision attacks on HTTP over TLS and OpenVPN. In **Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security**, pages 456–467. ACM, 2016.

[17] Cecylia Bocovich and Ian Goldberg. Secure asymmetry and deployability for decoy routing systems. **Proceedings on Privacy Enhancing Technologies**, 2018(3):43–62, 2018.

[18] Cecylia Bocovich and Ian Goldberg. Slitheen: perfectly imitated decoy routing through traffic replacement. In **Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security**, pages 1702–1714. ACM, 2016.

[19] Lee Brotherston. Stealthier Attacks and Smarter Defending With TLS Fingerprinting, DerbyCon, 2015.

[20] Chad Brubaker, Amir Houmansadr, and Vitaly Shmatikov. CloudTransport: using cloud storage for censorship-resistant networking. In **The** 14^{th} **Privacy Enhancing Technologies Symposium (PETS)**, 2014.

[21] Emil Burzo. No ability to use a supported (but not enabled) cipher suite. `https://github.com/square/okhttp/issues/2698`, 2016.

[22] Xiang Cai, Rishab Nithyanand, and Rob Johnson. CS-BuFLO: a congestion sensitive website fingerprinting defense. In **Proceedings of the 13th Workshop on Privacy in the Electronic Society**, pages 121–130. ACM, 2014.

[23] Censys. Most common Status Lines responses of an internet-wide scan of IPv4 HTTPS servers on port 443. `https://censys.io/ipv4/report?field=443.https.get.status_line.raw`, June 2020.

[24] Jacopo Cesareo, Josh Karlin, Michael Shapira, and Jennifer Rexford. Optimizing the placement of implicit proxies, June 2012.

[25] Zimo Chai, Amirhossein Ghafari, and Amir Houmansadr. On the Importance of Encrypted-SNI (ESNI) to Censorship Circumvention. In **9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19)**, 2019.

[26] Chandra X-ray Observatory. `http://cxc.harvard.edu`, October 2019.

[27] clowwindy. Shadowsocks (python). `https://github.com/shadowsocks/shadowsocks`, June 2019.

[28] Sara Dickinson, Daniel Kahn Gillmor, and K Tirumaleswar Reddy. Usage Profiles for DNS over TLS and DNS over DTLS. `https://rfc-editor.org/rfc/rfc8310.txt`, March 2018. DOI: `10.17487/RFC8310`.

[29] Roger Dingledine. Obfsproxy: the next step in the censorship arms race. `https://blog.torproject.org/obfsproxy-next-step-censorship-arms-race`, 2012.

[30] Roger Dingledine. Research problems: ten ways to discover tor bridges. `https://blog.torproject.org/research-problems-ten-ways-discover-tor-bridges`, October 2011.

[31] Roger Dingledine. Strategies for getting more bridge addresses. `https://blog.torproject.org/strategies-getting-more-bridge-addresses`, 2011.

[32] Roger Dingledine and Jacob Appelbaum. How governments have tried to block Tor. `https://svn-archive.torproject.org/svn/projects/presentations/slides-28c3.pdf`, 2011.

[33] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In **13th USENIX Security Symposium**, pages 303–320, August 2004.

[34] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[35] Chromium Developer Documentation. SPDY: an experimental protocol for a faster web. `https://www.chromium.org/spdy/spdy-whitepaper`, 2009.

[36] Tor documentation. Snowflake: pluggable transport that proxies traffic through temporary proxies using webrtc. `https://trac.torproject.org/projects/tor/wiki/doc/Snowflake`, 2018.

[37] Tor documentation. Tor: Pluggable Transports. `https://www.torproject.org/docs/pluggable-transports.html`.

[38] Frederick Douglas, Rorshach, Weiyang Pan, and Matthew Caesar. Salmon: robust proxy distribution for censorship circumvention. **PoPETs**, 2016(4):4–20, 2016.

[39] Arun Dunna, Ciarán O'Brien, and Phillipa Gill. Analyzing China's blocking of unpublished Tor bridges. In **Free and Open Communications on the Internet**. 8th USENIX Workshop on Free and Open Communications on the Internet (FOCI 18), 2018.

[40] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J Alex Halderman. A search engine backed by Internet-wide scanning. In **Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security**, pages 542–553. ACM, 2015.

[41] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J Alex Halderman, and Vern Paxson. The security impact of HTTPS interception. In **Proc. Network and Distributed System Security Symposium (NDSS)**, 2017.

[42] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast Internet-wide scanning and its security applications. In **22nd USENIX Security Symposium**, pages 605–620, August 2013.

[43] Kevin P. Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In **Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security**, pages 61–72. ACM, 2013.

[44] Frank Ehlis. How to disable SSL ciphers in Google Chrome. `http://fehlis.blogspot.com/2013/12/how-to-disable-ssl-ciphers-in-google.html`, 2013.

[45] Daniel Ellard, Alden Jackson, Christine Jones, Victoria Manfredi, W. Timothy Strayer, Bishal Thapa, and Megan Van Welie. Rebound: decoy routing on asymmetric routes via error messages. In **40th IEEE Conference on Local Computer Networks**, LCN, pages 91–99, October 2015.

[46] R. Ensafi, P. Winter, M. Abdullah, and J. Crandall. Analyzing the Great Firewall of China over space and time. In **Proceedings on Privacy Enhancing Technologies**, PETS, pages 61–76, 2015.

[47] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. Examining how the Great Firewall discovers hidden circumvention servers. In **15th ACM Internet Measurement Conference**, IMC, pages 445–458, October 2015.

[48] David Fifield. Anticipating a world of encrypted SNI: risks, opportunities, how to win big. https://groups.google.com/d/msg/traffic-obf/UyaLc9jPNmY/ovNImK5HEQAJ, August 2018.

[49] David Fifield. Cyberoam firewall blocks meek by TLS signature. https://groups.google.com/forum/#!topic/traffic-obf/BpFSCVgi5rs/, 2016.

[50] David Fifield. Fortiguard firewall blocks meek by TLS signature. https://groups.google.com/forum/#!topic/traffic-obf/fwAN-WWz2Bk, 2016.

[51] David Fifield. **Threat modeling and circumvention of Internet censorship**. PhD thesis, University of California, Berkeley, December 2017.

[52] David Fifield. Turbo Tunnel. Designing circumvention protocols for speed, flexibility, and robustness. https://www.bamsoftware.com/sec/turbotunnel.html, 2019.

[53] David Fifield, Nate Hardison, Jonathan Ellithorpe, Emily Stark, Roger Dingledine, Phil Porras, and Dan Boneh. Evading censorship with browser-based proxies. In **Privacy Enhancing Technologies Symposium**, pages 239–258. Springer, 2012.

[54] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-resistant communication through domain fronting. **Proceedings on Privacy Enhancing Technologies**, 2015(2):46–64, 2015.

[55] David Fifield and Lynn Tsai. Censors' delay in blocking circumvention proxies. In **6th USENIX Workshop on Free and Open Communications on the Internet (FOCI 16)**, Austin, TX. USENIX Association, 2016.

[56] Pawel Foremski, David Plonka, and Arthur Berger. Entropy/IP: Uncovering structure in IPv6 addresses. In **Proceedings of the 2016 Internet Measurement Conference**, pages 167–181. ACM, 2016.

[57] Freedom House. Freedom on the Net 2018: the rise of digital authoritarianism, 2018. https://freedomhouse.org/sites/default/files/FOTN_2018_FinalBooklet_11_1_2018.pdf.

[58] FreedomPrevails. High Performance NodeJS MTProto Proxy. `https://github.com/FreedomPrevails/JSMTProxy`.

[59] Sergey Frolov, Fred Douglas, Will Scott, Allison McDonald, Benjamin VanderSloot, Rod Hynes, Adam Kruger, Michalis Kallitsis, David Robinson, Nikita Borisov, J. Alex Halderman, and Eric Wustrow. An ISP-scale deployment of TapDance. **Free and Open Communications on the Internet (FOCI)**:49, 2017.

[60] Sergey Frolov, Jack Wampler, Sze Chuen Tan, J Alex Halderman, Nikita Borisov, and Eric Wustrow. Conjure: summoning proxies from unused address space. In **Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security**, pages 2215–2229, 2019.

[61] Sergey Frolov, Jack Wampler, and Eric Wustrow. Detecting Probe-resistant Proxies. In **Proc. Network and Distributed System Security Symposium (NDSS)**, 2020.

[62] Sergey Frolov and Eric Wustrow. The use of TLS in censorship circumvention. In **Proc. Network and Distributed System Security Symposium (NDSS)**, 2019.

[63] Sergiu Gatlan. South Korea is censoring the Internet by snooping on SNI traffic. `https://www.bleepingcomputer.com/news/security/south-korea-is-censoring-the-internet-by-snooping-on-sni-traffic/`, 2019.

[64] John Geddes, Max Schuchard, and Nicholas Hopper. Cover your ACKs: Pitfalls of covert channel censorship circumvention. In **Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security**, pages 361–372. ACM, 2013.

[65] John Gilmore. Cracking DES: secrets of encryption research. **Wiretap Politics & Chip Design**, 272, 1998.

[66] Devashish Gosain, Anshika Agarwal, Sambuddho Chakravarty, and H. B. Acharya. The devil's in the details: placing decoy routers in the Internet. In **Proceedings of the 33rd Annual Computer Security Applications Conference**, ACSAC 2017, pages 577–589, Orlando, FL, USA. ACM, 2017. ISBN: 978-1-4503-5345-8.

[67] Jamie Hayes and George Danezis. k-fingerprinting: A Robust Scalable Website Fingerprinting Technique. In **25th USENIX Security Symposium (USENIX Security 16)**, pages 1187–1203, Austin, TX. USENIX Association, August 2016. ISBN: 978-1-931971-32-4.

[68] Paul E. Hoffman and Patrick McManus. DNS Queries over HTTPS (DoH). `https://rfc-editor.org/rfc/rfc8484.txt`, October 2018. DOI: `10.17487/RFC8484`.

[69] Ralph Holz, Johanna Amann, Olivier Mehani, Matthias Wachs, and Mohamed Ali Kâafar. TLS in the wild: an Internet-wide analysis of TLS-based protocols for electronic communication. **CoRR**, abs/1511.00341, 2015. arXiv: `1511.00341`.

[70] Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. The SSL landscape: a thorough analysis of the X.509 PKI using active and passive measurements. In **Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference**, pages 427–444. ACM, 2011.

[71] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: observing unobservable network communications. In **Security and Privacy (SP), 2013 IEEE Symposium on**, pages 65–79. IEEE, 2013.

[72] Amir Houmansadr, Giang T. K. Nguyen, Matthew Caesar, and Nikita Borisov. Cirripede: circumvention infrastructure using router redirection with plausible deniability. In **18th ACM Conference on Computer and Communications Security**, CCS, pages 187–200, October 2011.

[73] Amir Houmansadr, Edmund L. Wong, and Vitaly Shmatikov. No direction home: the true cost of routing around decoys. In **21st Annual Network and Distributed System Security Symposium, NDSS 2014**. The Internet Society, 2014.

[74] Amir Houmansadr, Wenxuan Zhou, Matthew Caesar, and Nikita Borisov. SWEET: Serving the Web by Exploiting Email Tunnels. **IEEE/ACM Transactions on Networking**, 25(3), January 2017.

[75] How the Great Firewall of China is blocking Tor. In **Presented as part of the 2nd USENIX Workshop on Free and Open Communications on the Internet**, Bellevue, WA. USENIX, 2012.

[76] Lin Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. Analyzing forged SSL certificates in the wild. In **Security and privacy (sp), 2014 ieee symposium on**, pages 83–97. IEEE, 2014.

[77] Qing Huang, Deliang Chang, and Zhou Li. A comprehensive study of DNS-over-HTTPS downgrade attack. In **10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20)**. USENIX Association, August 2020.

[78] ToorCon Inc. crack.sh — the world's fastest DES cracker. `https://crack.sh/`.

[79] Jigsaw Operations LLC. Intra protects you from DNS manipulation. `https://play.google.com/store/apps/details?id=app.intra`, November 2018.

[80] Jigsaw Operations LLC. Outline VPN. `https://www.getoutline.org/en/home`.

[81] George Kadianakis. GFW probes based on Tor's SSL cipher list. `https://gitlab.torproject.org/legacy/trac/-/issues/4744`, 2011.

[82] Josh Karlin, Daniel Ellard, Alden W. Jackson, Christine E. Jones, Greg Lauer, David P. Mankins, and W. Timothy Strayer. Decoy routing: toward unblockable Internet communication. In **1st USENIX Workshop on Free and Open Communications on the Internet**, FOCI, August 2011.

[83] Sheharbano Khattak, Tariq Elahi, Laurent Simon, Colleen M Swanson, Steven J Murdoch, and Ian Goldberg. SoK: Making sense of censorship resistance systems. **Proceedings on Privacy Enhancing Technologies**, 2016(4):37–61, 2016.

[84] Erik Kline and Ben Schwartz. DNS over TLS support in Android P Developer Preview. `https://security.googleblog.com/2018/04/dns-over-tls-support-in-android-p.html`, April 2018.

[85] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. Estimating mutual information. **Physical review E**, 69(6):066138, 2004.

[86] Redis Labs. Redis is open source, in-memory data structure store, 2019. `https://redis.io/`.

[87] SSL Labs. HTTP client fingerprinting using SSL handshake analysis. `https://www.ssllabs.com/projects/client-fingerprinting/`, 2009.

[88] The Go Programming Language. Cgo. `https://golang.org/cmd/cgo/`, 2018.

[89] Lantern. `https://getlantern.org/`.

[90] Homin K Lee, Tal Malkin, and Erich Nahum. Cryptographic strength of SSL/TLS servers: current and recent practices. In **Proceedings of the 7th ACM SIGCOMM conference on Internet measurement**, pages 83–92. ACM, 2007.

[91] Bruce Leidl. Obfuscated OpenSSH. `https://github.com/brl/obfuscated-openssh/blob/master/README.obfuscation`, 2009.

[92] Andrew Lewman. Update on Internet censorship in Iran. `https://blog.torproject.org/update-internet-censorship-iran`, 2011.

[93] Fangfan Li, Abbas Razaghpanah, Arash Molavi Kakhki, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. Lib• erate,(n): a library for exposing (traffic-classification) rules and avoiding them efficiently. In **Proceedings of the 2017 Internet Measurement Conference**, pages 128–141. ACM, 2017.

[94] Shuai Li, Mike Schliep, and Nick Hopper. Facet: streaming over videoconferencing for censorship circumvention. In **Proceedings of the 13th Workshop on Privacy in the Electronic Society**, pages 163–172. ACM, 2014.

[95] Patrick Lincoln, Ian Mason, Phillip A Porras, Vinod Yegneswaran, Zachary Weinberg, Jeroen Massar, William Allen Simpson, Paul Vixie, and Dan Boneh. Bootstrapping communications into an anti-censorship system. In **2nd USENIX Workshop on Free and Open Communications on the Internet**. USENIX, 2012.

[96] Isis Agora Lovecruft and Henry de Valence. HYPHAE: Social Secret Sharing. Technical report, Tech. rep. 2017-04-21. Apr. 2017.

[97] Gordon Fyodor Lyon. **Nmap network scanning: The official Nmap project guide to network discovery and security scanning**. Insecure, 2009.

[98] Colm MacCarthaigh. Enhanced domain protections for Amazon CloudFront requests. https://aws.amazon.com/blogs/security/enhanced-domain-protections-for-amazon-cloudfront-requests/, 2018.

[99] Victoria Manfredi and Pi Songkuntham. Multiflow: cross-connection decoy routing using TLS 1.3 session resumption. In **8th USENIX Workshop on Free and Open Communications on the Internet (FOCI 18)**. USENIX Association, 2018.

[100] Bill Marczak, Nicholas Weaver, Jakub Dalek, Roya Ensafi, David Fifield, Sarah McKune, Arn Rey, John Scott-Railton, Ron Deibert, and Vern Paxson. An analysis of China's "Great Cannon". **FOCI. USENIX**:37, 2015.

[101] Moxie Marlinspike. Amazon threatens to suspend Signal's AWS account over censorship circumvention. https://signal.org/blog/looking-back-on-the-front/, 2018.

[102] Moxie Marlinspike. Doodles, stickers, and censorship circumvention for Signal Android. https://signal.org/blog/doodles-stickers-censorship/, 2016.

[103] Damon McCoy, Jose Andre Morales, and Kirill Levchenko. Proximax: measurement-driven proxy dissemination (short paper). In **Proceedings of the 15th International Conference on Financial Cryptography and Data Security**, FC'11, pages 260–267, Berlin, Heidelberg. Springer-Verlag, 2012. ISBN: 978-3-642-27575-3.

[104] Richard McPherson, Amir Houmansadr, and Vitaly Shmatikov. CovertCast: using live streaming to evade internet censorship. **Proceedings on Privacy Enhancing Technologies**, 2016(3):212–225, 2016.

[105] Paul V. Mockapetris. Domain names - implementation and specification. RFC 1035, IETF, November 1987.

[106] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. SkypeMorph: protocol obfuscation for Tor bridges. In **Proceedings of the 2012 ACM conference on Computer and communications security**, pages 97–108. ACM, 2012.

[107] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback. **Security Advisory**, 2014.

[108] Alex Moshchuk, Steven D Gribble, and Henry M Levy. Flashproxy: transparently enabling rich web content via remote execution. In **Proceedings of the 6th international conference on Mobile systems, applications, and services**, pages 81–93. ACM, 2008.

[109] MTProto mobile protocol: detailed description. `https://core.telegram.org/mtproto/description`.

[110] Milad Nasr and Amir Houmansadr. Game of decoys: Optimal decoy routing through game theory. In **Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security**, pages 1727–1738, 2016.

[111] Milad Nasr, Hadi Zolfaghari, and Amir Houmansadr. The waterfall of liberty: decoy routing circumvention that resists routing attacks. In **Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security**, pages 2037–2052. ACM, 2017.

[112] Refraction Networking. uTLS—fork of the Go standard TLS library, providing low-level access to the ClientHello for mimicry purposes. 2019. `https://github.com/refraction-networking/utls/`.

[113] ntop. PF_RING: High-speed packet capture, filtering and analysis. `http://www.ntop.org/products/pf_ring`.

[114] OpenVPN: VPN Software Solutions & Services For Business. `https://openvpn.net`.

[115] Jitendra Pahdye and Sally Floyd. On inferring TCP behavior. **ACM SIGCOMM Computer Communication Review**, 31(4):287–298, 2001.

[116] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. Website fingerprinting at Internet scale. In **NDSS**, 2016.

[117] Lantern Project. Lampshade: a transport between Lantern clients and proxies. `https://godoc.org/github.com/getlantern/lampshade`.

[118] ZMap project. ZGrab: application layer scanner that operates with zmap. `https://github.com/zmap/zgrab`.

[119] Psiphon. `https://psiphon.ca`.

[120] Psiphon Inc. Psiphon tunnel core. `https://github.com/Psiphon-Labs/psiphon-tunnel-core`, 2019.

[121] Reethika Ramesh, Ram Sundara Raman, Matthew Bernhard, Victor Ongkowijaya, Leonid Evdokimov, Anne Edmundson, Steven Sprecher, Muhammad Ikram, and Roya Ensafi. Decentralized control: a case study of Russia. In **Network and Distributed Systems Security (NDSS) Symposium 2020,** 2019.

[122] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. Studying TLS usage in Android apps, 2017.

[123] Refraction Networking: Internet freedom in the network's core. `https://refraction.network/`.

[124] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. `https://rfc-editor.org/rfc/rfc8446.txt`, August 2018. DOI: `10.17487/RFC8446`.

[125] Eric Rescorla. Transport Layer Security (TLS) parameters. `https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml`, 2018.

[126] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. TLS Encrypted Client Hello. Internet-Draft draft-ietf-tls-esni-07, Internet Engineering Task Force, June 2020. 31 pages. Work in Progress.

[127] Ivan Ristić. HTTP client fingerprinting using SSL handshake analysis. `https://blog.ivanristic.com/2009/06/http-client-fingerprinting-using-ssl-handshake-analysis.html`, 2009.

[128] Ivan Ristić. sslhaf: Passive SSL client fingerprinting using handshake analysis. `https://github.com/ssllabs/sslhaf`, 2009.

[129] David Robinson, Harlan Yu, and Anne An. Collateral freedom: a snapshot of chinese Internet users circumventing censorship. **Open Internet Tools Project Report**, 2013.

[130] Brian C Ross. Mutual information between discrete and continuous data sets. **PloS one**, 9(2):e87357, 2014.

[131] Runa Sandvik. Ethiopia introduces deep packet inspection. `https://blog.torproject.org/ethiopia-introduces-deep-packet-inspection`, 2012.

[132] Runa Sandvik. Kazakhstan uses DPI to block Tor. `https://trac.torproject.org/projects/tor/ticket/6140`, 2012.

[133] Runa Sandvik. UAE uses DPI to block Tor. `https://trac.torproject.org/projects/tor/ticket/6246`, 2012.

[134] Max Schuchard, John Geddes, Christopher Thompson, and Nicholas Hopper. Routing around decoys. In **2012 ACM Conference on Computer and Communications Security (CCS)**, CCS '12, pages 85–96, Raleigh, North Carolina, USA. ACM, 2012. ISBN: 978-1-4503-1651-4. DOI: `10.1145/2382196.2382209`.

[135] Will Scott, Thomas Anderson, Tadayoshi Kohno, and Arvind Krishnamurthy. Satellite: joint analysis of CDNs and network-level interference. In **2016 USENIX Annual Technical Conference**, ATC, pages 195–208, 2016.

[136] Sergey Arkhipov. MTProto proxy for Telegram in Golang. `https://github.com/9seconds/mtg`.

[137] Shadowsocks: a secure SOCKS5 proxy. `https://shadowsocks.org/assets/whitepaper.pdf`, January 2019.

[138] Khalid Shahbar and A Nur Zincir-Heywood. An analysis of Tor pluggable transports under adversarial conditions. In **2017 IEEE Symposium Series on Computational Intelligence (SSCI)**, pages 1–7. IEEE, 2017.

[139] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. Deep fingerprinting: undermining website fingerprinting defenses with deep learning. In **Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security**, pages 1928–1943. ACM, 2018.

[140] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen K Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. MD5 considered harmful today, creating a rogue CA certificate. In **25th Annual Chaos Communication Congress**, number EPFL-CONF-164547, 2008.

[141] statcounter. Browser version market share worldwide. `http://gs.statcounter.com/chart.php?device=Desktop&device_hidden=desktop&multi-device=true&statType_hidden=browser_version&region_hidden=ww&granularity=monthly&statType=Browser%20Version&region=Worldwide&fromInt=201504&toInt=201604&fromMonthYear=2015-04&toMonthYear=2016-04&csv=1`, 2016.

[142] Let's Encrypt Stats. Percentage of Web Pages Loaded by Firefox Using HTTPS. `https://letsencrypt.org/stats/#percent-pageloads`, 2018.

[143] Marc Stevens. Fast collision attack on MD5. **IACR Cryptology ePrint Archive**, 2006:104, 2006.

[144] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In **Annual International Cryptology Conference**, pages 570–596. Springer, 2017.

[145] Open Whisper Systems. Signal - private messenger. `https://signal.org/`.

[146] Telegram: a new era of messaging. `https://telegram.org/`.

[147] The Tor Project. obfs4 (the obfourscator) specification. `https://gitweb.torproject.org/pluggable-transports/obfs4.git/tree/doc/obfs4-spec.txt`.

[148] Tor Project. BridgeDB. `https://bridges.torproject.org/`.

[149] Michael Carl Tschantz, Sadia Afroz, Vern Paxson, et al. SoK: Towards grounding censorship circumvention in empiricism. In **Security and Privacy (SP), 2016 IEEE Symposium on**, pages 914–933. IEEE, 2016.

[150] uProxy. `https://www.uproxy.org/`.

[151] Benjamin VanderSloot, Sergey Frolov, Jack Wampler, Sze Chuen Tan, Irv Simpson, Michalis Kallitsis, J Alex Halderman, Nikita Borisov, and Eric Wustrow. Running refraction networking for real. **Proceedings on Privacy Enhancing Technologies**, 1:15, 2020.

[152] Mathy Vanhoef and Frank Piessens. All your biases belong to us: breaking RC4 in WPA-TKIP and TLS. In **USENIX Security Symposium**, pages 97–112, 2015.

[153] Liang Wang, Kevin P. Dyer, Aditya Akella, Thomas Ristenpart, and Thomas Shrimpton. Seeing through network-protocol obfuscation. In **Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security**, pages 57–69. ACM, 2015.

[154] Qiyan Wang, Xun Gong, Giang TK Nguyen, Amir Houmansadr, and Nikita Borisov. Censorspoofer: asymmetric communication using IP spoofing for censorship-resistant web browsing. In **Proceedings of the 2012 ACM conference on Computer and communications security**, pages 121–132. ACM, 2012.

[155] Qiyan Wang, Zi Lin, Nikita Borisov, and Nicholas Hopper. rBridge: user reputation based Tor bridge distribution with privacy preservation. In **20th Annual Network and Distributed System Security Symposium, NDSS 2013**. The Internet Society, 2013.

[156] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective attacks and provable defenses for website fingerprinting. In **23rd USENIX Security Symposium**, pages 143–157, 2014.

[157] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In **Annual international cryptology conference**, pages 17–36. Springer, 2005.

[158] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. StegoTorus: a camouflage proxy for the tor anonymity system. In **Proceedings of the 2012 ACM conference on Computer and communications security**, pages 109–120. ACM, 2012.

[159] Tim Wilde. Great firewall Tor probing circa 09 DEC 2011. `https://gist.github.com/da3c7a9af01d74cd7de7`, 2011.

[160] Tim Wilde. Knock knock knockin' on bridges' doors. Tor Blog, 2012. `https://blog.torproject.org/blog/knock-knock-knockin-bridges-doors`.

[161] Brandon Wiley. Blocking-resistant protocol classification using Bayesian model selection. Technical report, University of Texas at Austin, 2011.

[162] Philipp Winter, Tobias Pulls, and Juergen Fuss. Scramblesuit: a polymorphic network protocol to circumvent censorship. In **Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society**, pages 213–224. ACM, 2013.

[163] Ryan Wright and Adam Wick. CyberChaff: confounding and detecting adversaries. `https://galois.com/project/cyberchaff/`, 2019.

[164] Eric Wustrow, Colleen M. Swanson, and J. Alex Halderman. TapDance: end-to-middle anticensorship without flow blocking. In **23rd USENIX Security Symposium**, pages 159–174, August 2014.

[165] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: anticensorship in the network infrastructure. In **20th USENIX Security Symposium**, August 2011.

[166] Linus Yang, Max Lv, and Clow Windy. Shadowsocks-libev – libev port of shadowsocks. `https://github.com/shadowsocks/shadowsocks-libev`, 2014.

[167] ZeroMQ Distributed Messaging. `http://zeromq.org/`.

[168] Tao Zhu, David Phipps, Adam Pridgen, Jedidiah R. Crandall, and Dan S. Wallach. The velocity of censorship: high-fidelity detection of microblog post deletions. In **22nd USENIX Security Symposium (USENIX Security 13)**, pages 227–240, Washington, D.C. USENIX Association, August 2013. ISBN: 978-1-931971-03-4.