Recipes for Resistance: A Censorship Circumvention Cookbook

by

Cecylia Bocovich

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2018

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Vern Paxson
Supervisor	Ian Goldberg
Internal Members	Urs Hengartner
	Bernard Wong
Internal-external Member	Jennifer R. Whitson

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

The entirety of this thesis was researched and written under the supervision of Ian Goldberg, who additionally provided guidance and input in the development of Slitheen, proposed the gossip protocol for asymmetric decoy routing deployments, and contributed modifications to Firefox and NSS in our implementation of Slitheen's overt user simulator. The experimental setup used for evaluation and testing in Chapter 3 and Chapter 5 were designed and implemented with the help of Lori Paniak. The content in Chapter 4 was developed with the help of discussions with Anna Lorimer, who also coined the name Slifox, and helped with setting up the Docker container test environment and experimental setup.

Abstract

The increasing centralization of Internet infrastructure and web services, along with advancements in the application of machine learning techniques to analyze and classify network traffic, have enabled the growth and proliferation of Internet censorship. While the Internet filtering infrastructure of censoring authorities improves, cracks and weaknesses in the censorship systems deployed by the state allow Internet users to appropriate existing network protocols in order to circumvent censorship attempts. The relationship between censors and censorship resistors is often likened to a cat-and-mouse game in which resistors struggle to find new gaps in nation-state firewalls through which they can access content freely, while censors are devoted to discovering and closing these gaps as quickly as possible.

The life cycle of censorship resistance tools typically begins with their creation, but often ends very quickly as the tools are discovered and blocked by censors whose ability to identify anomalous network traffic continues to grow. In this thesis, we provide several recipes to create censorship resistance systems that disguise user traffic, despite a censor's complete knowledge of how the system works. We describe how to properly appropriate protocols, maximize censorship-resistant bandwidth, and deploy censorship resistance systems that can stand the test of time.

Acknowledgements

I would firstly like to thank Ian Goldberg, my supervisor. It was his graduate class on privacy enhancing technologies that first captured my interest and set me on the path of fighting for Internet freedom. I am forever grateful for his enthusiasm, guidance, and dedication to pulling people up the ladder.

This thesis would not have been possible without the lightning response times, hard work, and system administration skills of Lori Paniak. I would also like to thank the machines: this work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo. I am grateful to my thesis committee: Urs Hengartner, Bernard Wong, Jennifer R. Whitson, and Vern Paxson for offering their time, effort, and perspectives on this thesis.

The CrySP graduate lab and all of the experiences and friendships made there are the most treasured possessions I have carried with me throughout the last four years. I will continue to look back on the lab as the most joyful and meaningful part of my PhD. I have loved the discussions that surpass the technical details of our research, the adventures, and the continued drive to challenge everything, empower others, and make something new that pervade the lab. The network of love and support that we created together is the most beautiful and radical experience that I have been a part of.

Thank you also to friends far away that continue to bring happiness to my life. Our meetings are brief but no less wonderful as a result. To Drew, for reminding me how to be kid when I need it the most. To my friends at UnMonastery and Zagori who showed me how we can build something new together. To everyone I met upon arriving at Waterloo who have since dispersed but continue to bring joy into each other's lives.

I am incredibly grateful to my family: Mary Jane, Mike, Joanne, Carolyne, and Nick, for their love and emotional support. They have supported me endlessly in every endeavour that I have begun and prioritize my personal happiness over any other type of accomplishment. Thank you to Charmaine, Farrell, Sol, and Sabrina for the encouragement and support. Finally, thank you to Adriel for always being full of sunshine.

Ideally, this thesis acknowledgement would be a comprehensive and profound thank you to everyone I love in a way that highlights the impact they've had on my life during the last four years. In reality what I have written above falls far short of that intent, and instead the only way I hope to express my gratitude is in the adventures we will continue to have together in the years to come.

Dedication

This thesis is dedicated to the cracks and weaknesses in oppressive structures that allow resistance to grow. Let's make more.

Table of Contents

List of Tables

List of Figures

RECIPES FOR RESISTANCE

BY
CECYLIA BOCOVICH

A CENSORSHIP CIRCUMVENTION COOKBOOK

Chapter 1

Introduction

In recent years, Internet censorship has become an increasing world-wide concern. A 2016 Freedom House report [KTS+16] showed a steady decline in Internet freedom for the six consecutive years leading up to the study. They reported that in 2016, roughly two-thirds of Internet users dealt with government censorship. This censorship aims to cut off access from websites that support political opposition, marginalized communities, and the criticism or satirization of those in power. Furthermore, journalists and users of social media that disseminate, or merely read, content that a censoring nation deems contrary have faced personal dangers such as arrest or increased scrutiny [dP17]. A recent crackdown on the suspected usage of privacy enhancing technologies that are meant to protect such users in Turkey led to the arrest of approximately 75,000 people [Bow17].

Ever since its inception, Internet protocols and applications have been designed with poor security and privacy properties, a trend that has continued to the present day to the detriment of many Internet users. That is not to say that the marginalized and oppressed have not found refuge in various sites and services that the Internet has to offer. Despite an overwhelming set of odds stacked against them, people have found ways to communicate, share, and organize online. However, they do so at great risk to themselves. The ever-increasing centralization of infrastructure, the disconnect between marginalized groups and the developers of Internet services, and the increased cooperation of popular sites with state censors and the police have resulted in a decline of Internet freedom and the safety of individuals that use it in ways that differ from the ideals of the powers that be.

The Internet was never designed by or for marginalized or oppressed groups. What is left to those that wish to resist is to find ways to hack the existing infrastructure to provide a sliver of empowerment and protections to those that need it most. This is an uphill battle, against

adversaries with more resources, who are in a better position to benefit from future advances in technology.

The history of Internet freedom tools shows a repeating theme in which systems to circumvent censorship are created and then cease to be useful as the technology of censors and their censorship infrastructure improves, surpassing previous assumptions of their inability to identify and filter traffic. The cost and effort required to design, build, and deploy censorship resistance systems draws from a much smaller pot of resources than the censor's does for responding to their existence. Similarly, the effort required to detect and analyze changes in Internet censorship practices by state censors dwarfs the ability of censors to discover the usage of effective new censorship resistance tools.

We are in need of tools for circumventing censorship that remain effective and protect their users without making assumptions about censors' ability to analyze network traffic. These tools should continue to be usable for years after their deployment, despite complete knowledge of their existence, implementation, and deployment by an adversarial censor.

Thesis statement: *Whereas existing systems that aim to hide censorship-resistant traffic from a censor decline in usefulness over time as assumptions limiting the censor's ability to detect circumvention tools collapse with improvements to traffic analysis technologies, we can design and deploy usable Internet freedom tools that stand the test of time, despite open knowledge of their operation and use, and despite technological improvements that enhance the traffic analysis abilities of the censor.*

This thesis is a collection of recipes for building systems for censorship resistance that will stand the test of time in the presence of an adversary that continues to benefit disproportionately from the Leviathan that drives the evolution of technology and the Internet, accompanied by a scientific analysis of experiences in realizing them. The intent is to provide ideas and a basis for the design of censorship resistance systems that will spur the creation of usable, secure, and empowering technology that will enable people to oppose the authorities that aim to stem or manipulate communication.

1.1 Contributions

We provide three generalizable recipes in this thesis that support our thesis statement by describing steps that guide the design and deployment of censorship circumvention tools. These recipes satisfy the properties we outline in our statement: they perfectly hide many features of censorship-resistant traffic by design, their design and deployment are effective even if the censor has complete knowledge of both, and they do not rely on current assumptions of the analytical

inability of censors to process large amounts of data. These recipes are accompanied by a description of what we have made with them, but are meant to be generally applicable, iterated on, modified, and applied further.

We begin with an introduction to Internet filtering, the type of Internet censorship that is the target of our recipes and the subject of analysis in this thesis. We define and discuss measurement studies on the current known filtering techniques used by censors to block access to a subset of the Internet and define our threat model. We demonstrate the cat-and-mouse relationship between censors and censorship resistors, enumerate the features of Internet traffic that censors can see and use to decide which traffic to block, and describe the current techniques used to evade this form of censorship.

Our first recipe in Chapter 3 outlines a method for tunnelling censorship-resistant traffic through appropriated protocols and connections that remain unblocked by the censor. This recipe shows how to make the single use of a protocol indistinguishable from regular, non-resistance traffic. In Chapter 4, we provide a recipe for improving the bandwidth of censorship resistance systems and hiding a single individual's use of the system by simulating compliance with the censor's filtering policies. Our final recipe in Chapter 5 provides insight on various deployment strategies and steps for the successful deployment of censorship circumvention tools. We conclude this thesis in Chapter 6 with a summary of our contributions, a discussion of future work, and a defence of our thesis statement.

Throughout the entirety of this thesis, we demonstrate and scientifically evaluate our recipes through the analysis of a system for censorship circumvention called Slitheen[1] [BG16, BG18]. Our implementation of Slitheen is available online[2] and under continued development.

[1]The name of our system is a reference to a family of Doctor Who aliens that disguise themselves by fitting perfectly inside of their victims.

[2]https://crysp.uwaterloo.ca/software/slitheen/

Chapter 2

Internet filtering

There are many different types of Internet censorship, each with its own set of threat models, warranting a different set of solutions. In this thesis, we focus on circumventing **Internet filtering**, a form of Internet censorship in which a censor, typically a state actor, analyzes network traffic in an attempt to *selectively* block access to sites and services. In this chapter, we define Internet filtering and summarize recent efforts to measure and document how Internet filtering happens at the network level. We then describe the relationship between changes in Internet filtering practices and the development of systems to circumvent them, motivating the thesis of this document. Finally, we formally define our threat model and the censor that will be our adversary for the recipes we present in the subsequent chapters.

2.1 Definitions and measurements

The centralized nature of current networks and the ability of state actors to manipulate key points in Internet infrastructure has enabled the wide-spread practice of filtering access to content on the Internet. The analysis of network traffic and the blocking of selected connections can occur at several different points in the network. There have been many studies on the censorship practices of high-profile countries known to block access to politically sensitive topics.

In this section, we provide a summary of selected studies that show basic Internet filtering techniques such as IP address blocking [AWR14, EKAC14] and DNS manipulation [Nab13, AAH13, PJL+17], the use of which has been measured in at least 58 countries [PJL+17]. We then take a closer look at advanced filtering techniques used in China [WL12, CMW06], whose

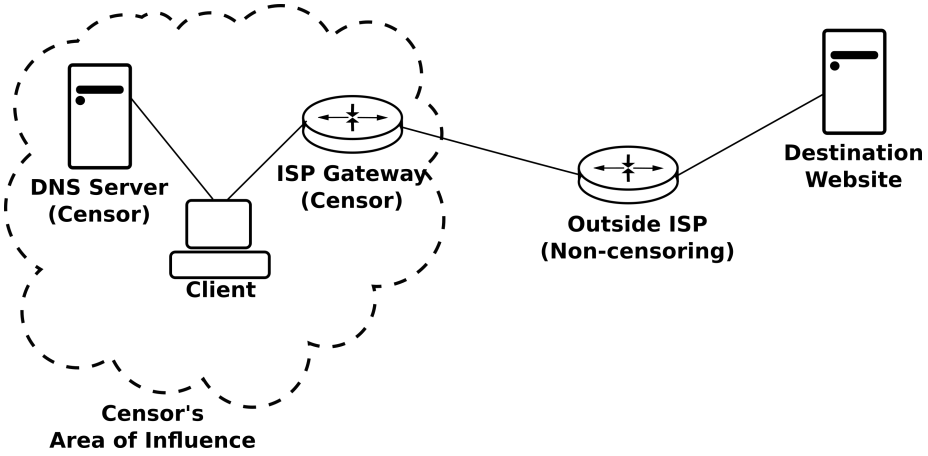


Figure 2.1: A generic view of Internet infrastructure. To access a website, a client must first resolve the hostname to an IP address by contacting a DNS sever, which is often under the control of the censor. The client then makes a connection to the destination IP, their traffic passing through censor-controlled routers before leaving the censor's area of influence.

Great Firewall is only a small piece of their widespread Internet censorship efforts to perform the ongoing suppression of political discourse.

We define the **area of influence** of a censor to be the geographical region(s) over which the censor is able to observe or control network traffic, similar to the "sphere of influence" defined by Elahi et al. [EDH+16]. The majority of countries that practice Internet filtering do so by maintaining a **blacklist** of sites to block. This allows censors to enumerate and block access to high-profile sites and services, while keeping the rest of the Internet free for use by those that reside in their area of influence. A more restrictive whitelist of allowed sites or protocols is much less common and typically surrounds real-world events of intense political interest to the censor, such as the whitelisting of allowed protocols leading up to the 2013 presidential election in Iran [AAH13]. In extreme cases, censors have also been known to conduct complete Internet blackouts, a trend that has been increasingly common and typically occurs during political protests or elections [Xyn18].

2.1.1 Basic filtering techniques

We present a generic view of network infrastructure in Figure 2.1. Web browsing connections made by users inside the censor's area of influence pass through censor-controlled infrastructure on their path to the destination site. For a typical web browsing session, a user first performs

a domain name lookup to find the IP address of the destination site at a Domain Name System (DNS) server controlled by the censor. The user's connection to the destination IP address then passes through routers controlled by the censor before exiting the censor's area of influence.

The blockage of destinations by IP address occurs at routers, typically gateway routers [And12], that are subject to the control of Internet Service Providers. Upon discovering a connection to an IP address on a censor's blacklist, the censor node will inject spoofed TCP RST packets to both the client and server machines, terminating the connection from the client [CMW06, WCQ+17].

In two case studies conducted in Turkey and Russia in 2014, Anderson et al. [AWR14] measured the direct blocking of IP addresses to prevent access to websites containing political content. In Turkey, the authors detected the blocking of IP addresses associated with the hostname twitter.com. In a similar time frame, Russia was reported blocking access to the LiveJournal accounts of opposition leaders and a Ukrainian blog documenting events in Crimea.

Access to Tor [DMS04], a system for anonymous web browsing commonly used for censorship circumvention, is a frequent target for IP blocking attempts [TAAP16]. In a 2014 study using a remote technique to detect blocking at the TCP and IP layers, Ensafi et al. [EKAC14] reported the blocking of Tor relays, Tor directory authorities, which maintain lists of the IP addresses of Tor relays, as well as known Tor bridges, which provide unlisted entry points to the Tor network.

While IP blocking is a basic technique to terminate connections to forbidden sites or services, the blocking or manipulation of DNS queries is a widespread and effective technique to stop connections even before they are issued from the client. When users load a website, they typically first retrieve the IP address of the site's host from a DNS server. These servers are distributed around the world, but requests within a censor's area of influence can be easily directed by a censor to their own state-owned servers. For example, Anderson et al.'s case study of Turkey documented the blockage of Google's DNS servers, 8.8.8.8 and 8.8.4.4, which were being used to circumvent Turkey's own DNS manipulation attempts.

The return of erroneous DNS responses for URLs that correspond to prohibited content is known as **DNS manipulation**. This practice has been documented in individual studies from within countries such as Iran [AAH13], Pakistan [Nab13], and China [Ano14], as well as in up to 58 countries through the use of external, global measurements [PJL+17]. Censored responses either timeout, redirect the user to an internal censorship page (often in the 10.0.0.0/8 range), or return a valid yet incorrect IP address in place of the correct response.

The Great Firewall (GFW) of China has long used keyword filtering techniques to block access to sensitive content. This is done in the upstream direction of the connection (i.e., requests

for content), and in the downstream direction (i.e., results that violate the censor's content policy) [CMW06]. The GFW analyzes plaintext wherever possible, such as in the HTTP requests and responses of unencrypted web browsing sessions.

With the increased use of the Transport Layer Security protocol (TLS) to form end-to-end encrypted connections between clients and servers, censors no longer have access to the plaintext web traffic of an increasingly large percentage of Internet connections. For the remainder of this thesis, we focus on techniques used to block encrypted connections; although many states have implemented Internet filtering techniques that selectively block plaintext HTTP requests, the growing adoption of TLS has reduced the efficacy of this technique. Furthermore, the anti-censorship tools we discuss all function only in the presence of unblocked TLS connections.

2.1.2 Advanced filtering techniques

The development of more advanced measures to filter access to Internet content has been motivated by the deployment and use of censorship circumvention tools [Wil12, WL12, TAAP16]. We will discuss this relationship in greater detail in the next section, and summarize the techniques themselves and evidence of their use here. Advanced filtering techniques can therefore be described as attacks on the usage of censorship circumvention tools and comprise both active and passive methods. Active attacks such as active probing attempt to discover the existence of new endpoints (IP addresses and domains) that are in active use for censorship resistance. Passive attacks use deep packet inspection (DPI) techniques to discover the identifying characteristics of censorship circumvention systems in network traffic, eliminating the need to maintain comprehensive blacklists of emerging entry points to censorship resistance networks.

An early study by Wilde in 2011 discovered the use of DPI techniques to automatically detect and block access to entry points into the Tor anonymity network [Wil12]. They found that the GFW was inspecting the ClientHello message of the TLS handshake, sent by the client to the entry point, for a cipher suite list that was unique to Tor.

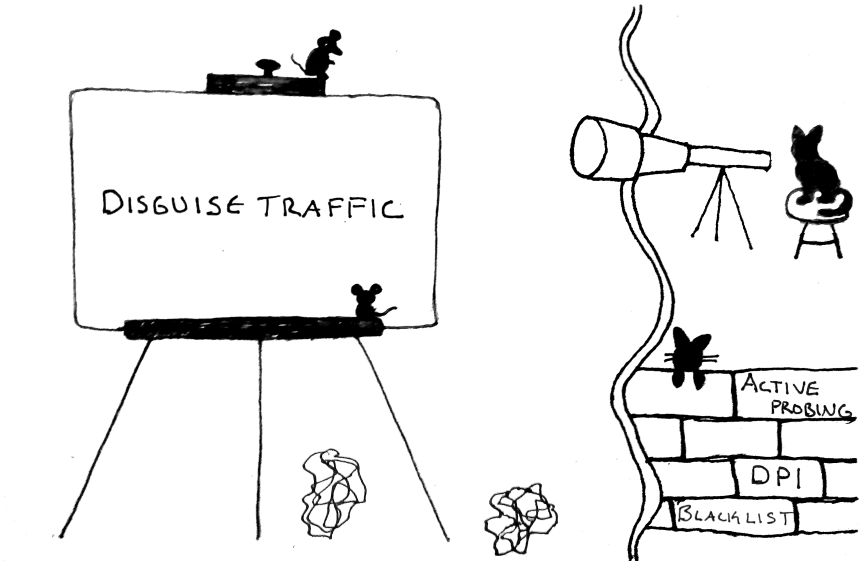
Winter and Lindskog conducted a follow-up study in 2012 that documented the discovery and blockage of entry points to the Tor network [WL12] by China. They gained several important insights into the application of fingerprinting to traffic originating from Chinese clients. An important point, relevant to our analysis of the capabilities of censors, is that the fingerprinting of Tor traffic occurred only on connections to destinations outside of China. This drastically cuts down on the amount of network traffic analysis necessary, increasing the ability of censors to more comprehensively analyze outgoing traffic.

A more recent study on the Great Firewall's fingerprinting and blocking efforts in 2015 by Ensafi et al. [EFW+15] showed that in addition to the ciphersuite list of ClientHello messages,

Chinese censors were also able to automatically detect the usage of obfs2 [Tor15a], an early traffic obfuscation tool, by looking for the telltale decryption key that precedes an encrypted connection between the client and destination.

Active probing [WL12] is a method used by the Great Firewall of China to discover new, emerging entry points to Tor. When a connection is made by a client from within the firewall's influence with suspicious characteristics as determined by the above DPI methods, attempts to connect to the Tor network are automatically sent from various IP addresses within China. In the event of a successful connection to Tor, the destination IP address is categorized as a bridge relay and added to the censor's blacklist.

While the original documentation of active probing in 2011 by Wilde reported probing in 15-minute intervals [Wil12], Ensafi et al. discovered that by 2015 over half of all suspicious connections were probed within one second of a client connecting to the Tor network, and that the probe attempts were to connect not only to Tor, but several other known censorship circumvention systems at the time [EFW+15].



2.2 The censorship cat-and-mouse game

Changes in censorship practices, techniques, and infrastructure are often tied to major political events in the censor's area of influence [Nab13, AAH13, AWR14]. However, changes in censor-

ship also follow the advances or adoption of censorship resistance systems. In the presence of censorship, developers design new systems to evade this censorship. These systems are adopted by users and subsequently discovered by government censors, who then improve their systems to detect and block the new tools. In this way, the relationship between censors and the developers and users of censorship resistance tools can be characterized as a cat-and-mouse game.

The relationship between censors and censorship resistance systems is reminiscent of similar problems in cryptography, and benefits from the application of Kerckhoffs' principle [Ker83]. Originally accompanied by five other design principles for military ciphers, what is today known as Kerckhoffs' principle states (in the English translation) that the system:

> "should not require secrecy, and it should not be a problem if it falls into enemy hands" [Pet97].

Perhaps ironically, the original intended beneficiaries of Kerckhoffs' advice are our enemies in the cat-and-mouse game—the states that use technology for manipulation and oppression. Our enemies have perhaps a larger advantage and more direct control over us than Kerckhoffs' did, making his principle even more useful to us as it is even more likely that our systems will fall into their hands. To put this in context with the design of systems for censorship resistance, we must assume that knowledge of censorship resistance systems will eventually fall into the hands of censors. Our tools should continue to evade censorship in an undetectable manner even if censors are aware of their existence, the details of how they work, and are looking for them.

The history of circumventing Internet filtering is littered with examples of censorship resistance tools that have worked for a brief amount of time, but were eventually discovered, analyzed, and blocked. Many systems are designed to be secure in the event of their discovery, but rely on assumptions of the computational, network, and data analysis limitations of censors. While these assumptions are true at the time of the system's inception, they cease to be true as technology advances and benefits those in power disproportionately to those in resistance to it. This relationship between the tactics used by state censors and the development of tools to bypass Internet filtering is strongly exemplified by the GFW and the efforts of the Tor Project to bypass it, as described by Tschantz et al. [TAAP16].

Tor [DMS04] was originally proposed as a system for anonymous web browsing and is widely used for censorship circumvention. Tor extends the simple proxy model by routing the user's traffic through a circuit of three proxies, or relays. The additional hops guarantee web-browsing anonymity for the client, even in the event that the censor has compromised one of the relays. However, Tor itself does not mask a client's participation in the system. Clients select relays from a publicly available list, one that is also available to a censoring authority. Censors

have been known to block access to Tor by simply blacklisting connections to known Tor entry, or guard, relays [Lew09].

In response to the blocking of Tor guards, the Tor Project has begun to gradually and selectively release the location of secret or hidden entry relays, called bridges [Tor15b]. A client may use these relays to continue circuit construction with publicly listed relays. As Tor bridges are not included in public Tor directories, they are much more difficult for a censor to track down and block. However, censors such as the Great Firewall of China have employed other techniques to identify Tor traffic in the event that the client is using a hidden entry to Tor.

Tor traffic has several distinguishing characteristics necessary for providing anonymity, but allow a censor to distinguish Tor traffic from regular traffic to an unknown IP address. In 2012, Winter and Lindskog experimentally confirmed that the GFW could identify the use of Tor bridges with DPI boxes due to the unique ciphersuite list sent by Tor clients in the TLS ClientHello message [WL12]. Furthermore, Tor traffic is distinguishable in the fact that all packets entering and leaving the Tor network are padded to 512-byte cells.

Thus, the current battleground between censors and censorship resistors in the Internet filtering context is the disguise and discovery of network traffic patterns. When users access content, their network traffic produces identifiable features. We divide these features into the following categories:

1. Plaintext Information: Protocol messages and data that have not been encrypted are visible to anyone on the network path. This includes metadata such as the source and destination IP addresses of the Internet connections, as well as application-layer messages that establish an encrypted communication session.

2. Metadata: In this context, metadata is information about plaintext or encrypted data other than the data itself. This includes timing information such as when the data is being sent or the time that elapses between requests from the user and responses from the server. It also includes how the data is split into network packets that are transmitted between the user and the visited site, and linkable destinations between the same client and multiple sites. Note that, contrary to other contexts, we do not consider the source and destination IP addresses to be metadata in our scenario, unless the real source and destination of the network traffic is not reflected as plaintext in protocol messages.

Censors can and have used both plaintext information and metadata in attempts to identify and block censorship resistance traffic. To draw from the examples given above, censors have analyzed the plaintext IP addresses of packets to block access to known proxies or Tor relays, and the plaintext TLS messages to look for unique ciphersuite negotiation messages that identify

Tor. Packet size and timing metadata has recently been the suspected cause of the blocking of obfs4, a censorship circumvention technique used by Tor [Fif18].

Current attempts at designing new censorship resistance systems aim to disguise both plaintext information and metadata traffic patterns that can be used to identify censorship resistance tools. Tor provides a variety of **pluggable transports** to users, that each take a slightly different approach to hiding the identifying features of network traffic in connections to the Tor network. These transport protocols are designed to be swapped into use in different scenarios, as determined by the region of the user and the censorship techniques employed by the censor.

The different approaches to hiding traffic patterns can be split into three main categories: obfuscation, mimicry, and appropriation. Each approach boasts examples of currently successful censorship circumvention tools. However, they also suffer from challenges and setbacks that make them vulnerable to the cat-and-mouse game and the disproportionate ability of the censor to make use of technological advances or exercise state control.

Traffic obfuscation approaches this problem by masking the defining characteristics of network traffic through randomization, with the intent of preventing censors from identifying it as any known protocol or tool [WPF13, Din12]. The success of this technique is grounded in the assumption that censors are unwilling to block traffic that they are unable to definitively classify as censorship resistance or contrary to their governance, as that would possibly lead to an increase in public unrest [EDH+16]. However, past precedent indicates that during decisive political events, censors may be willing to take the risk. For example, Aryan et al. recorded the blocking of undefined Internet protocols by the government of Iran during the 2013 presidential elections [AAH13].

Mimicry aims to make connections indistinguishable from popular unblocked content or services, forcing censors to make a difficult decision: to either continue to expand their list of blocked sites to include popular services (thereby risking public unrest), or surrender their position. Many pluggable transports shape traffic or encapsulate it in messages that closely resemble protocols such as HTTP [DCS15], Skype [MMLDG12], or HTML [WWY+12]. The ultimatum presented to the censor rests entirely on the ability of these systems to mimic allowed sites and services more closely than the censor's ability to exploit minor differences. Houmansadr et al. [HBS13] argue that the maintenance of near-perfect mimicry is extremely difficult; as advances in computing allow censors to classify large amounts of traffic more accurately, censorship resistors will see themselves on the losing side of this reactive battle.

Appropriation uses real connections to allowed sites and services to tunnel censorship resistance traffic. This is distinct from mimicry; instead of manipulating traffic patterns to make it appear as the user is accessing an allowed site or server, the user is in fact accessing it — the difference from regular use is that the encrypted data sent between the user and the site

is censorship resistance traffic as opposed to the data or service the site normally offers. Appropriation is often more difficult to deploy as many tools that use this technique require the collaboration of the sites or services [FLH+15], or the owners of significant network infrastructure [WWGH11, HNCB11, KEJ+11]. Additionally, although appropriation hides identifying plaintext features of traffic patterns, it may not hide metadata by default. Geddes et al. discovered several systems that use appropriation to tunnel censorship resistance traffic and failed to account for differences in the underlying traffic that produced features identifiable to a potential censor [GSH13].

Our recipes make use of appropriation, and we will discuss how to perform protocol appropriation correctly and safely in greater detail in the subsequent chapters. Our focus on appropriation as opposed to the other techniques ties back in with our adherence to Kerckhoffs' principle and desire to make censorship resistance systems secure even in the event of their discovery, turning the cat-and-mouse game around to benefit the resistor. We see in appropriation the ability to definitively disguise both plaintext and metadata traffic patterns in the presence of a whitelisting censor, which mimicry and obfuscation are unable to accomplish. In our recipes, we remove any assumptions about the technical abilities of the censors and show that regardless of a censor's ability to process and analyze data our censorship resistance traffic remains indistinguishable from regular traffic to allowed sites and services.

2.3 Threat model

These recipes are meant to empower users that reside inside the area of influence of a state censor with the ability and intent to filter access to the Internet. Our recipes are designed to be effective against censors that maintain a whitelist of allowed sites or content, although they will be just as effective in the much more common case of censors that use a blacklist to enumerate sites or content that should be blocked. We do, however, assume that the censor is allowing connections from all users to at least *some* sites or services that are hosted outside of the censor's area of influence, and that this practice is quite common.

The area of influence in this model includes regions over which the censor may not have complete control, but can convince or coerce the governing body in that region to collude in the blocking or identification of censorship resistance traffic. In addition to allowing access to some outside sites or services, we also assume that user traffic to these servers pass through Internet infrastructure that exists outside the censor's area of influence. We do not provide recipes for the event of a complete Internet blackout, in which users are unable to connect to any Internet resources outside of the censor's area.

In addition to allowing users to access blocked content, we also aim to prevent the censor from identifying the use of censorship circumvention tools by anyone in their area of influence. We assume that the censor has the ability and intent to not only analyze large amounts of traffic to discover anomalous behaviour, but is also willing to single out individual targets or suspects to monitor their traffic more closely in the attempt to block censorship resistance traffic or obtain evidence of its use.

We assume that censors are aware of the details of current and future censorship circumvention tools. This includes the cryptographic and networking techniques used to encrypt or disguise traffic, as well as the deployment of any infrastructure needed for users to access the systems (unless otherwise explicitly stated). The grounds for this assumption are well founded, as they are in keeping with Kerckhoffs' principle. States have a disproportionate ability to perform reconnaissance on the existence or development of new tools, compared to our ability to analyze censorship infrastructure.

We make a very small set of assumptions that bound the abilities of our adversary, though unlike limits on traffic analysis and machine learning capabilities, these assumptions are likely to continue to hold in the future. Namely, we assume that the censor cannot break state-of-the-art public-key cryptographic protocols. We also assume that users can make end-to-end encrypted connections with sites or services outside of the censor's area of influence. For the purpose of scoping this work, we also forgo an analysis of possible side-channel attacks and out-of-band targeted attacks such as in-person surveillance, malware distribution, etc.

Despite the strength of our defined adversary, there remain opportunities to resist. In the following recipes, we will show how to leverage the allowance of whitelisted connections to sites and services outside the censor's area of influence, prevent censors from flagging the users of censorship circumvention systems as suspicious, and explain how to deploy our tools in existing Internet infrastructure in the presence of censors with strong reconnaissance abilities.

Chapter 3

Recipe #1: Protocol appropriation

Internet protocols have been developed over the years to transmit information between devices on the Internet. Many are too useful for censors to block, making them ideal candidates for **appropriation**, or use outside of their original intended purpose, for censorship circumvention. The goal of appropriation is to use a common protocol to transmit covert traffic in a censorship-free way such that an observing censor is unable to read the covert traffic, or even detect the deviation from the protocol's original, intended usage.

Protocols that lend themselves well to appropriation for censorship-resistant communication are those that are used very frequently (e.g., Domain Name Service (DNS) [AFK16] or Transport Layer Security (TLS) [WWGH11, HNCB11, KEJ+11, WSH14, EJM+15, BG16, NZH17]) and those that normally transmit a large amount of information (e.g., HyperText Transfer Protocol (HTTP) [EJM+15, BG16, NZH17] or Voice-over-IP [BSR17, HBS13]). However, done incorrectly, protocol appropriation can leave identifying characteristics that are detectable by an observant censoring authority [GSH13].

In this chapter, we begin by describing in more detail the advantages of protocol appropriation, followed by a description of previous attempts and their shortcomings.[1] We will then lay out a generally applicable method for the correct appropriation of protocols, which does not fall victim to the identifying characteristics of those analyzed previously. Finally, we propose and evaluate a novel censorship circumvention system, called Slitheen, that successfully appropriates both TLS and HTTP in this manner.

[1] This chapter contains text from our CCS 2016 paper [BG16].

3.1 Previous attempts at appropriation

Voice-over-IP (VoIP) is a natural choice for appropriation due to the high bandwidth and low latency necessary to transmit its usual traffic: video or voice calls between distant users. FreeWave is a censorship circumvention tool that tunnels censorship-resistant traffic through VoIP connections between the client and dedicated FreeWave VoIP IDs [HRBS13]. Houmansadr et al. implement FreeWave using Skype, a popular VoIP service. The client's censorship circumvention traffic is modulated into audio signals and sent over the VoIP connection to the FreeWave server, which then decodes the traffic and proxies between the client and the previously blocked destination site.

A challenge with this type of appropriation is the easily identifiable difference between the traffic patterns of normal VoIP calls and those used for web browsing, even with the employed audio modulation. Despite a statistical analysis showing that the audio modulation of FreeWave is similar to that of a normal Skype call [HRBS13], Geddes et al. show that their analysis of packet sizes, lengths, and timings was not enough and that when looked for specifically, FreeWave produces patterns that allow a sufficiently capable censor to reliably fingerprint its use [GSH13].

Although VoIP is a popular protocol, capable of providing a high-bandwidth channel for censorship-resistant traffic, the use of variable bit rate (VBR) encoding makes hiding the features of the underlying tunnelled traffic difficult. In contrast, DNS is an extremely low-bandwidth protocol whose requests are difficult to fingerprint. The advantage of DNS is that it is extremely commonly used; users make multiple DNS requests every time their browser loads a website. Due to the low-bandwidth nature of systems that appropriate DNS, they are typically useful in conjunction with other censorship resistance systems that require censorship-resistant **bootstrapping**. Bootstrapping is an initial setup step required by some censorship resistance systems before they can be used. A few protocol appropriation systems require users to register or preshare keys with another endpoint of the system [HNCB11, NZH17, EJM+15]. These registration and out-of-band communication protocols require a very small amount of bandwidth, making them the ideal use case for low-bandwidth protocol appropriation systems.

Akbar et al. propose a system called DNS-sly that encodes small amounts of censorship-resistant traffic in the responses of a cooperative DNS server [AFK16]. This technique assumes the cooperating DNS server has not been compromised by the censor and itself requires out-of-band bootstrapping with the client before use. Clients encode upstream traffic in their choice of domains to resolve. Downstream traffic in DNS-sly is encoded by the subset and order of the IP addresses returned for a user query. In addition to the difficulties of bootstrapping this system, DNS-sly traffic also deviates from usual patterns of requests and responses, which may be used to fingerprint its use.

Domain fronting [FLH+15] is a widely deployed technique to hide the true destination of a client's traffic through the appropriation of HTTPS connections to cloud services. Unlike existing pluggable transports, domain-fronted traffic appears to the censor to be heading to a legitimate, allowed website. The actual, covert destination (typically a proxy running on the same cloud service as the allowed website) is hidden in the `Host:` header field of the HTTP header to the allowed site. Destination information appears in three different places in an HTTPS request: the IP address, the TLS Server Name Indication (SNI) extension, and the Host header of the HTTP request. While the first two are viewable by a censor or any router between the client and the destination, the HTTP request is encrypted with all other application data after the completion of the TLS handshake. Domain fronting is the practice of specifying one domain, usually an edge server for a cloud service, in the IP address and SNI fields, while setting the encrypted HTTP host header to a different domain. Recent proposals to encrypt the SNI field [HR18], made possible with the move from TLSv1.2 to TLSv1.3, would allow for a similar practice, where the IP address indicates an edge service and the SNI indicates a covert domain.

The pluggable transport that uses this technique to connect to Tor is called meek. To make a connection to Tor using meek, the client establishes a TLS connection with an edge server of an uncensored service that allows domain fronting. Several content distribution networks (CDNs) and large websites have in the past allowed domain fronting for web applications that subscribe to their services. The proxy only has to subscribe and pay for bandwidth (shown to be between 0.10 and 0.20 USD per GB [FLH+15]) in order for their service to be accessible from an overt edge server. After establishing a connection to the overt destination, the client can issue HTTP requests to the meek proxy. Packets are redirected to the proxy by the overt destination according to the Host field of the HTTP header. The client's Tor traffic is then tunneled over HTTP to the proxy and sent to a Tor guard. In this way, the proxy to Tor "hides" behind the cloud service. To block all traffic to the meek proxy, the censor would have to block all traffic to the front service, causing collateral damage.

In a recent evaluation of meek, Fifield reports that the cost of running meek grew to just over 26,000 USD in 2016 and has cost a grand total of over 50,000 USD from the time of its deployment to measurements taken in early 2017 [Fif17]. These measurements were taken from deployments on Google App Engine, Amazon CloudFront, and Microsoft Azure. Furthermore, meek does not hide the patterns of underlying traffic, instead relying on the large amount of traffic processed by these services to hide accesses to censored content. Although it remains to be seen in the wild, the application of website fingerprinting techniques that use the size and timing of responses to classify accesses to specific webpages [ZH16] could be applied to meek traffic.

SWEET (Serving the Web by Exploiting Email Tunnels) [HZCB17] appropriates email as an essential service to relay content between the user and a blocked website. It was originally

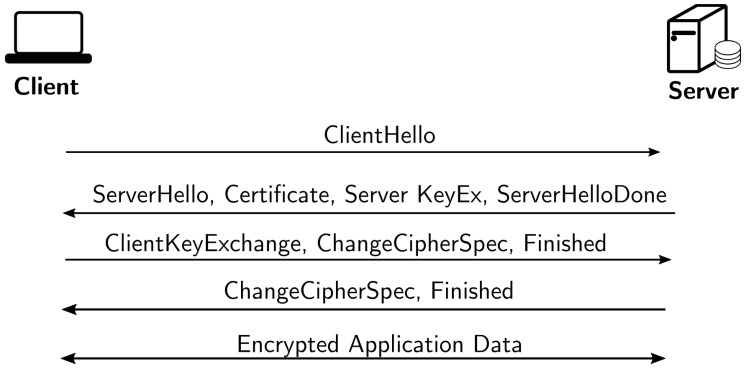
proposed and submitted to ArXiv.org in 2012, but recently published in 2017. Clients send and receive censorship-resistant traffic by sending and receiving emails to and from a dedicated SWEET server. The fact that the client is communicating with the SWEET server is hidden by their encrypted connection to their email provider. As with meek, SWEET does nothing to hide the volume or frequency of emails between the client and their provider and thus exposes the censorship-resistant traffic to website fingerprinting techniques.

3.1.1 Decoy routing

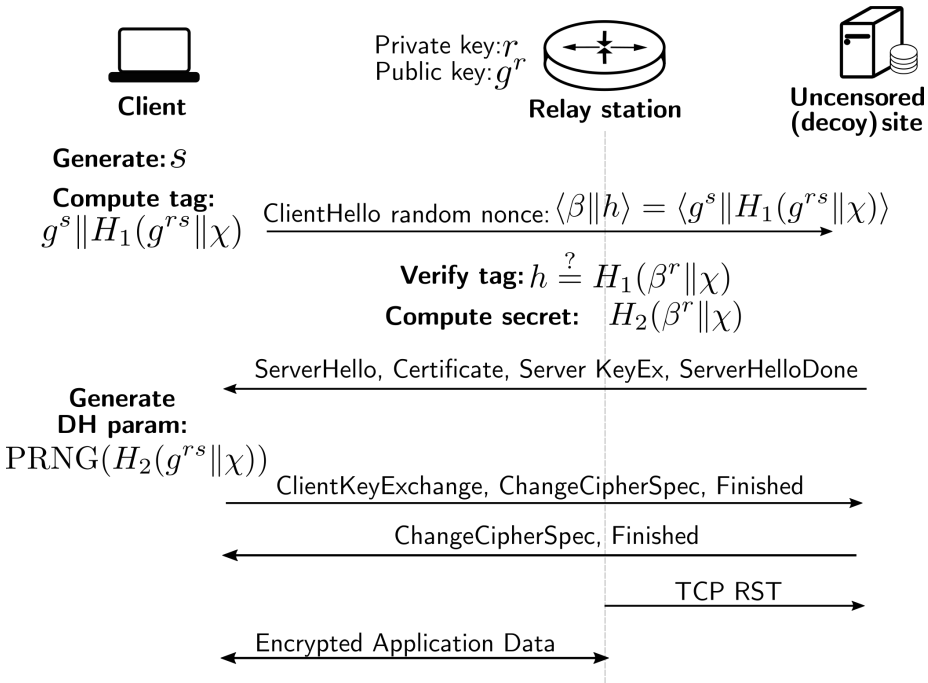
Decoy routing [WWGH11, HNCB11, KEJ+11, WSH14, EJM+15, BG16, NZH17] is a general technique for circumventing Internet filtering that uses protocol appropriation. Decoy routing systems have two main characteristics: 1) they appropriate the Transport-Layer Security (TLS) protocol to fool censors into believing the user is making a secure connection to an unblocked website, and 2) the infrastructure for decoy routing systems is deployed at Autonomous Systems (ASes) in the middle of the network, rather than at endpoints, a technique also known as End-to-Middle (E2M) proxying. In this chapter, we will focus on the first of these characteristics, the means by which decoy routing systems appropriate TLS, and in some cases HTTP, to deliver censorship-resistant traffic and hide the usage of the system. We will discuss the second characteristic, E2M proxying, in a later recipe for deploying censorship resistance systems in Chapter 5.

The first generation of decoy routers surfaced in 2011, proposed by three independent research groups. Telex [WWGH11], Cirripede [HNCB11], and Curveball [KEJ+11] all use the same basic technique in which the client steganographically expresses a desire to access blocked content covertly by tagging the setup messages in a seemingly benign connection to an overt destination. This destination is one that has not been blocked by the censor and resides outside of the censor's area of influence. For these systems, we refer to the overt destination also as a **decoy** destination. The steganographic tags are recognized by a **relay station** on the path between the client and the decoy site, but are provably unidentifiable to a censoring ISP without the relay station's private key. After the tag has been recognized, the relay station facilitates the flow of information between the client and the blocked website via a man-in-the-middle proxy, maintaining the illusion that the user is accessing the decoy page while they are instead covertly receiving content from the blocked site.

Telex tags TCP flows for censorship resistance by making slight modifications to the TLS handshake. We give an overview of the Telex tagging procedure in Figure 3.1, along with an overview of TLS for reference. TLS is a protocol used to establish a shared secret between a client and a server, and to send encrypted data back and forth between the two parties us-



(a) Original, unmodified TLS handshake. These messages are a subset of the total allowed messages and variations of TLS, as defined in RFC 5246 [DR08].



(b) Modified TLS session for tagged flows using the Telex tagging procedure. We define the context string χ as `server_ip`$\|\rho$, where $\rho =$ `ClientHello random`$[0..3]$. After the TLS handshake is complete, the relay station severs the connection to the decoy site and proxies information between the client and a covert destination.

Figure 3.1: A comparison of the original TLS handshake and the modifications made by Telex to appropriate it for use in censorship resistance.

ing that shared secret. In a regular use of TLS, the client initiates the handshake by sending a ClientHello message. This message includes, among other values, a random nonce that seeds the computation of the TLS master secret. The hello message also includes suggestions for various methods and parameters that negotiate the encryption and authentication methods used. The server then responds with its own ServerHello message, supplying an additional random nonce, and completing the negotiations on ciphersuites, parameters, and extensions. The server then sends a Certificate that can be verified by the client and a ServerKeyExchange message containing the server's public key values for negotiating a shared secret. After the client has sent their ClientKeyExchange message, both sides can compute the shared master secret. They both send a ChangeCipherSpec message, indicating that all subsequent messages will be encrypted, and complete the handshake with encrypted Finished messages that must be verified for correctness by both sides before encrypted application data is sent. There are many variations and extensions to TLS, as defined in RFC 5246 [DR08].

In Telex, the tag is placed in the random nonce of the ClientHello message that initiates a TLS handshake with the decoy site. From this tag and the station's private key, the station can compute the client's Diffie-Hellman (DH) exponent, allowing it to compute the session's TLS master secret and man-in-the-middle the connection between the client and the overt destination. Upon the receipt, decryption, and verification of both TLS Finished messages, the station severs the connection to the overt destination and assumes its role as a proxy, preserving the server-side TCP and TLS state. The client may then connect to a censored website through the relay station; the traffic between the client and the censored site appears to the censor as encrypted traffic to and from the decoy destination.

Cirripede takes a different approach by inserting a tag in the Initial Sequence Numbers (ISNs) of TCP SYN packets to register the client with the relay station over the course of 12 TCP connections. Once registered, the client initiates a TLS connection with a decoy site, now routed by the relay station through a *service proxy*, and sends an initial HTTP GET request. After the request goes through, the service proxy terminates the connection on behalf of the client, and begins to impersonate the overt destination. The proxy generates a new TLS session key, computed from the registration tag and station private key, and issues a ChangeCipherSpec message and Finished message to the client. Once the client responds with a valid Finished message, the service proxy begins to relay traffic between the client and the censored site until a predetermined time interval has passed.

In Curveball, the client and the relay station share a predetermined secret, obtained through the use of an out-of-band channel. This secret is used to generate a tag recognizable by the station and is inserted in the ClientHello message of a TLS handshake to the overt destination, similar to Telex. Upon receipt of this tagged message, the relay station observes the completion of the TLS handshake with the overt destination, and assumes the role of the server, sending the client a

Hello message in the form of a TLS record encrypted with the client-station shared secret. Once the client responds with a similar Hello message, the station begins to proxy traffic between the client and the censored site.

TapDance [WSH14] is a second-generation decoy routing system designed to be more deployable than first-generation systems. TapDance flows are tagged in an initial, incomplete HTTP GET request to the decoy site, after the negotiated TLS handshake. The TapDance station recovers the tag (steganographically embedded in the ciphertext of an ignored GET request header), and uses the corresponding secret to encrypt a confirmation, mimicking an HTTP response from the overt site. The client then sends the station upstream requests for a censored site, making sure to never signify the completion of the initial GET request to the overt site. As such, the overt site will continue to receive upstream data from the client without complaint. The station fulfills the client's requests for censored content, issuing encrypted responses on behalf of the overt site. As the only downstream data from the server are the original TLS handshake messages (which are not needed by the TapDance station), and the TCP ACK messages following extra data from the incomplete header, the station does not need to witness this downstream traffic, making the scheme more deployable in realistic network conditions where packets going to the decoy site may take a different path through the Internet than returning packets.

Rebound [EJM+15] is another second-generation decoy routing system that provides an asymmetric solution and defends against an active adversary capable of injecting or modifying packets. Clients tag Rebound flows in a manner similar to Telex, by inserting a tag in the ClientHello message of a TLS handshake, enabling the Rebound station to compute the master secret of the TLS connection between the client and the overt site. They achieve an asymmetric version of the tagging procedure by leaking the values of the server random nonce and the ciphersuite to the Rebound station by embedding them in the ciphertext of initial HTTP GET requests from the client to the overt site. Once the Rebound station is able to man-in-the-middle the TLS connection, the client begins issuing requests for censored content embedded in invalid HTTP GET requests to the overt site. The relay station fetches the censored content and stores it in a queue. When the next invalid GET request is received by the Rebound station, the station replaces the URL field of the request with content from the censored site. This information is forwarded to the overt site, which "rebounds" the encrypted content, inside an HTTP error response.

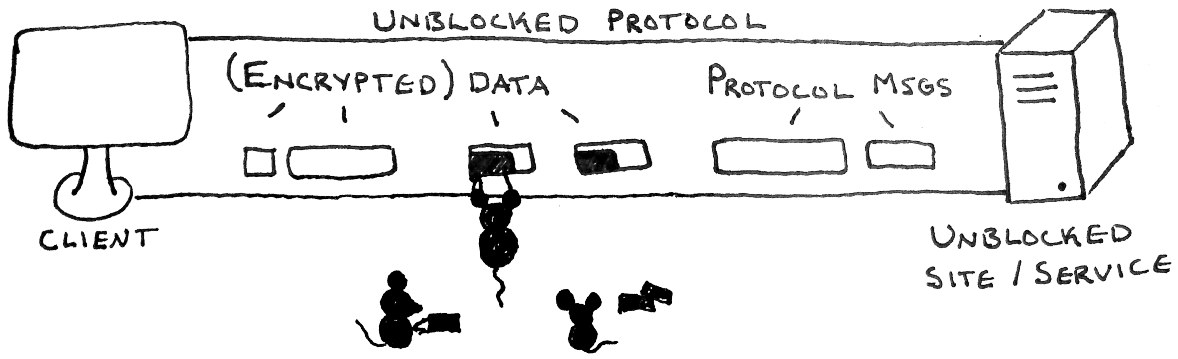
Rebound maintains the connection between the client and the overt site. The TCP state of the overt site will report the TCP sequence and acknowledgement numbers expected by the censor. Furthermore, by rebounding content off of the overt site in the form of error messages, Rebound delivers downstream data from the proxy to the client even if the underlying network routes are asymmetric. There are two major barriers to the adoption of Rebound as a censorship resistance technique. The first barrier is that a client must send upstream data in an equal amount to the

downstream data they receive to avoid mismatched TCP sequence numbers upstream, alerting the censor of a decoy session. In typical Internet usage, the ratio between upstream data sent and downstream data received by the client is very low; this trend is reflected in the bandwidth provided by most ISPs. The second barrier to adoption is the flood of bad HTTP GET requests that the client sends to the overt site. The frequency and size of these requests are reminiscent of HTTP flooding, a class of Denial of Service attacks, and will likely be blocked by the overt site.

One advantage of decoy routing is that, if used for censorship-resistant web browsing, the appropriated and tunnelled protocol are the same and thus the expected distribution of traffic patterns are similar, making it more difficult to identify its use. However, the decoy destination for each connection is visible to the censor and can be reliably expected to exhibit traffic patterns that are distinct from other destinations. These distinctions can be identified using website fingerprinting. All first- and second-generation decoy routing systems are vulnerable to traditional website fingerprinting techniques, in which the censor can compare packet sizes, timings, and directionality to differentiate between decoy and regular traffic while fingerprinting the censored site.

All of the above decoy routing systems are vulnerable to the timing analysis attacks introduced by Schuchard et al. [SGTH12]. They showed that differences in latency due to fetching content from a possibly distant censored server through the decoy routing proxy is enough to not only detect the usage of a decoy routing system, but also fingerprint the censored webpage accessed. Although Rebound's stored queue of censored content reduces the latency that stems from proxying traffic between the client the covert destination, it does not account for the latency introduced by the relay station in replacing the contents of the HTTP GET request.

While second-generation decoy routing systems can be credited for addressing the deployability challenges of decoy routing, they fall woefully short of first-generation systems in their ability to appropriate protocols in a way that disguises their use to a knowledgable censor. Third-generation systems are a response to this short-coming that aim to disguise as much as possible the identifying patterns of decoy routing traffic. In the next section we will discuss our method for appropriating protocols that we will later apply to the appropriation of HTTP in Slitheen, our proposed third-generation decoy routing system [BG16].



3.2 A generalizable method for protocol appropriation

In this section, we discuss a generalizable method for appropriating uncensored protocols to tunnel censorship-resistant traffic. We define **regular use** of the protocol to be its original, intended use that is valuable enough to a censor to prevent them from blocking it. We define **appropriated use** of the protocol to be that which tunnels censorship-resistant traffic to the user. Our goal is to make these two uses indistinguishable by making the traffic patterns (both plaintext information and metadata) of the appropriated usage of the protocol identical to regular usage from the point of view of the censor.

3.2.1 Step 1: Use the protocol as intended

The main challenge of proper protocol appropriation is the difference in traffic patterns between the protocol as it is regularly used and the censorship resistance traffic. Our approach to solving this problem is to use the appropriated protocol as intended, such that the endpoints involved (the protocol implementation on both the client and destination machines) cannot detect a deviation from the protocol's regular use.

In the examples of decoy routing systems covered in the previous section, the decoy site has no knowledge of its role in censorship resistance efforts. It receives TLS protocol messages from the client and responds to them in the usual way, unable to detect a deviation from a regular connection. The detectable deviation only comes afterwards, when the relay station severs the connection to the decoy site. Our method carries this technique forward, and has the client continue to interact with the decoy site in the usual manner and load the entire webpage. The

entire session, from the initial connection to termination, should be indistinguishable from a regular access to the decoy site to both the decoy site itself and the censor.

To appropriate HTTP, our method would have the client load a full, valid webpage. For VoIP, the client would establish a video or voice call and send *real, captured* video and audio frames, not frames that were generated or converted from other data.

3.2.2 Step 2: Replace leaf data

Our method does no good if the client never deviates from the intended usage of the protocol. This would mean the delivery of the usual, uncensored content to the user's machine, while what the user actually desires is access to content that has been blocked. To accomplish this, we have our friend outside of the censor's area of influence replace the original, uncensored data with content that the client requests. This can be done at a router in the middle of the network, as is suggested for decoy routing systems, or at the destination itself. Our requirements are that this entity is a man-in-the-middle, able to read and alter all messages between the client and the decoy.

However, as is the case with all protocols, some of the data transmitted between the client and the destination influences future messages and traffic patterns. We define **leaf data** as data that does not alter the traffic patterns of a connection. For example, in an HTTP session, an HTML file contains information that the client's browser parses to load a web page. HTML files often contain instructions for the browser to issue subsequent connections to possibly multiple destinations for more resources. An image, on the other hand, is displayed only to the user and will not prompt the client browser to produce more network traffic. Therefore, images are leaf data and may be safely replaced by the friendly man-in-the-middle.

We have our friendly man-in-the-middle replace only leaf data with censorship-resistant traffic from the blocked, covert site. This keeps plaintext traffic patterns identical to regular usage, as well as the amount of data that passes between the client and the decoy. Linkable connections from the client are identical to a regular use of a connection to the decoy site as the leaf data will not interfere with future connections made by the client's implementation of the appropriated protocol.

In addition to only replacing leaf data, we wait to perform the replacement until the data is ready to be sent on the network to the decoy site. For an IP connection, this means the data has already been packetized into packets, and we replace the leaf data on a packet-by-packet basis. This hides the metadata patterns of packet sizes and timings. Our remaining requirement of the man-in-the-middle is to do the replacement quickly so as to eliminate any identifiable latency caused by our system.

Ideally, the leaf data (and, correspondingly the censorship-resistant traffic it is replaced with) will be encrypted so as to be unreadable by the censor. Whether the data is encrypted or otherwise encoded, the observable data seen by the censor after the replacement should be drawn from distributions that are indistinguishable from one another without knowledge of a secret key.

3.2.3 Step 3: Simulate interactive elements

The regular use of most protocols involves an interaction between the user and the data they receive. In the case of voice calls, the user speaks into a microphone and the sounds they produce are transmitted to the other endpoint. For web browsing, the user visits multiple pages. Different degrees of simulation are required depending on the bandwidth of the appropriated protocol (i.e., the amount of leaf data available for replacement), and the amount of censorship-resistant traffic that the user is requesting.

To accomplish this, we introduce the idea of an Overt User Simulator (OUS). The OUS is part of the client's software that uses the appropriated protocol to make as many connections to the decoy destination as needed to allow the user to send or receive content in a censorship-resistant way. The OUS interacts with the client-side implementation of the appropriated protocol to produce observable traffic that is not suspicious to a censor knowledgable of the system. We give a recipe for simulating user behaviour in a convincing and high-bandwidth way in Chapter 4.

3.3 Appropriating secure web browsing with Slitheen

In this section, we give an example of how to appropriate HTTPS using the method described in the previous section, and propose a state-of-the-art decoy routing system called Slitheen, whose use is indistinguishable in the eyes of a censor from regular web browsing traffic.

3.3.1 Appropriating TLS

In this section, we refer to the friendly man-in-the middle for our system as a relay station, keeping the terminology used by decoy routing systems even though this method can be deployed as an end-to-end system. In the case that the deployment is E2E, i.e., where the man-in-the-middle is the decoy site itself, the appropriation of TLS to share the session master secret is an unnecessary though still valid step that can ease deployment as it does not require modifications to the web server. In this case, the decoy site would deploy the relay station directly in front of the site so that it has access to all packets passing between the web server and clients.

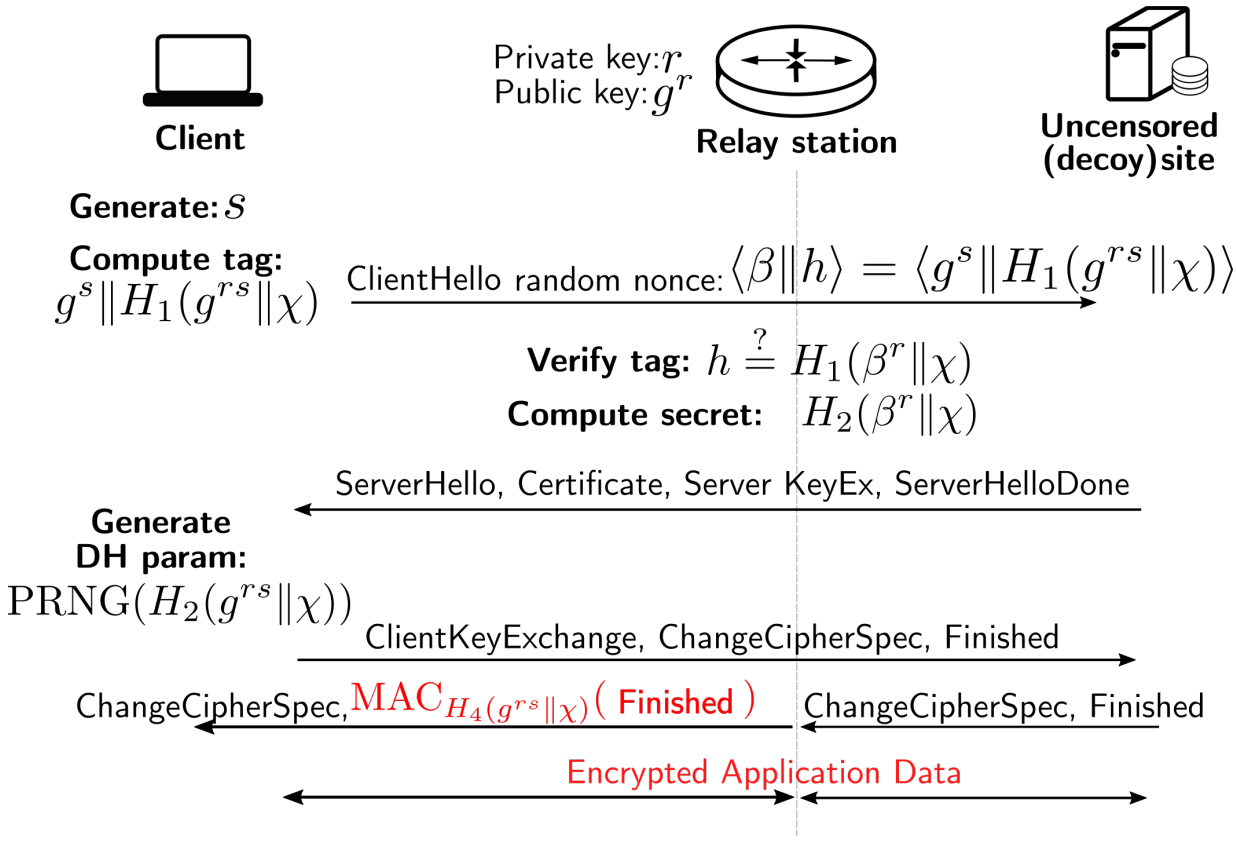


Figure 3.2: Slitheen modifications to TLS. Modifications to the original Telex tagging procedure are highlighted in red. Slitheen has the relay station modify the TLS Finished message sent by the decoy site to the client in order to alert the client that the connection has been successfully intercepted. The relay station is then able to decrypt, modify, and re-encrypt the following application data, which it does on a packet-by-packet basis.

In order to establish communication with the relay station and allow it to decrypt the traffic between the client and the decoy site, Slitheen appropriates TLS using a tagging procedure similar to that used by Telex. We make a few minor modifications to the Telex tagging procedure for security purposes, highlighted in red in Figure 3.2. To appropriate a secure web browsing session, we follow the three-step procedure outlined in the previous section.

Step 1: Use the protocol as intended

Following our method, the client completes a real, valid TLS handshake with the decoy site in a manner that is indistinguishable from a regular handshake by either the decoy site or a censor. The result is a secure, encrypted connection between the client and the decoy site. The relay station deployed between the client and the decoy can man-in-the-middle this connection and decrypt or modify its contents, but a censor is unable to view or alter the encrypted traffic.

Step 2: Replace leaf data

The amount of data we need to replace is very small, as all we need to do is provide the relay station with a seed with which they can calculate the TLS master secret of the session.

There are only two pieces of data being altered in our appropriation of TLS:

1. The ClientHello random nonce:
   This value is in regular use randomly generated and used to seed the calculation of the TLS master secret. We replace this random value with a tag that consists of a public key generated by the client, concatenated with a hash used by the relay station to detect the client's intent to appropriate the protocol.
   The replacement value is meaningful only to the client and the relay station, which hold the secret keys necessary to generate and detect the hash value of the tag. It draws from a random distribution that is computationally indistinguishable from that of the uniform random distribution used to generate the nonce in regular use.

2. The Finished message from the decoy site to the client:
   A value that, without the TLS master secret established in the previous TLS handshake messages, is indistinguishable from random. We replace this in order to signal the successful establishment of a decoy routing session between the client and the relay station. This signal is useful, as without it the man-in-the-middle is invisible to the client. Its presence signals to the client that the connection is usable for censorship-resistant traffic, and its absence is a warning that the censor may have man-in-the-middled their connection instead through the use of a compromised Certificate Authority or the installation of a compromised root certificate.
   The replacement value, also being a hash, draws from the same distribution as the original Finished message and is undetectable by both the censor and the decoy site.

Step 3: Simulate interactive elements

The interaction between the client and the decoy site is implemented by modifying popular, existing implementations of TLS. The connections are made by a modified version of a popular web browser, and the fingerprint of the TLS handshake is identical to the fingerprints produced by the regular use of the web browser and TLS implementation.

After the establishment of the encrypted connection between the client and the decoy site using our tagging procedure, the relay station is able to decrypt application-layer messages in both directions. This allows them to receive **upstream** data that contains the client's requests for censorship-resistant content, and replace **downstream** resources from the decoy site with the results of these requests.

3.3.2 Appropriating HTTP

As opposed to previous decoy routing systems, Slitheen maintains the connection between the client and the decoy site after the TLS handshake as opposed to severing it. The station extracts upstream data to the covert destination from specialized headers in *valid* HTTP GET requests to the decoy site, and replaces leaf data from the decoy site with downstream data from the covert destination. We show an overview of the Slitheen architecture in Figure 3.3.

Step 1: In Slitheen, we have the client access the decoy site exactly as a regular client would, fetching both the original HTML page from the decoy site as well as all additional resources, as determined by the browser, necessary to completely load the page.

A typical access to content on the web consists of a collection of TCP connections, over which flow HTTP requests and responses. The first connection to a decoy destination typically requests an HTML document, which in turn prompts the client's browser to issue several more requests to HTTP servers (the same or different) to collect various resources such as cascading style sheets (CSS), JavaScript, images, and videos.

Step 2: The leaf data in HTTP are the contents of HTTP requests and resources that never prompt the server or browser, respectively, to perform additional behaviour. In the upstream direction, this includes ignored HTTP headers and values. In the downstream direction, this includes images, and the displayed data of audio or video resources. Resources such as HTML, JavaScript, and CSS are not leaf data, as they are parsed by the browser and can cause the browser to produce additional network traffic that is observable by the censor. We do not replace this data, allowing the browser to parse it as it usually would.

Step 3: The Overt User Simulator (OUS) in this system is a full browser capable of establishing the encrypted connection to the decoy site(s), parsing HTTP resources, and rendering any

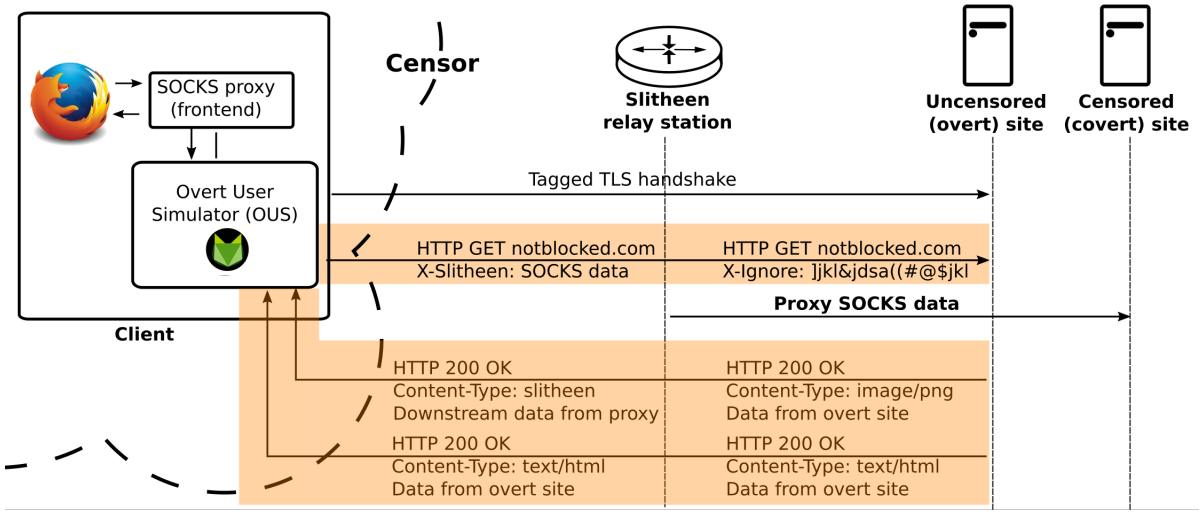


Figure 3.3: After the establishment of a TLS session between the client's Overt User Simulator (OUS) and the decoy site, the Slitheen station may monitor the encrypted traffic (shaded) in both directions. The station receives upstream proxy data from the client in an X-Slitheen header of a valid HTTP GET request to the decoy site. Once the station has relayed the upstream data to the censored site, it stores the downstream responses in a queue. When the station receives responses from the decoy site, it replaces leaf content types such as images with the queued data. This data is then forwarded by the OUS to the SOCKS front-end and finally received by the client's browser. The censor sees only the TLS handshake and encrypted traffic to and from the decoy site.

supplied data that resides on the client's machine. The browser we use is a slightly modified version of Firefox, called Slifox,[2] which we discuss in more detail in Chapter 4.

We define a Slitheen **user** to be the human who uses our system to access censorship-resistant content. A Slitheen **client** is the software installed on the user's machine that performs the steps outlined above to deliver censorship-resistant content to the user. When a user initiates a Slitheen session with the desire to tunnel censorship-resistant traffic, their client first randomly generates a 32-byte Slitheen identification number. The client then proceeds to access decoy sites through the OUS. The client's OUS initiates a decoy routing session with the relay station and appropriates the TLS session, as discussed in Section 3.3.1.

After the TLS session has been established and the tagging procedure is complete, the relay station continues to passively monitor the session as the OUS proceeds to request content from the decoy site in the usual manner by issuing valid HTTP GET or POST requests, deviating only to replace upstream leaf data with the client's Slitheen identification number and requests for censorship-resistant content.

When the Slitheen station receives HTTP requests from the client for a resource on the decoy site, the station inspects the headers of this request for an X-Slitheen header containing the Slitheen identification number of the client, and any upstream data meant for a covert destination. The station now associates the TLS session with the given Slitheen ID, replaces the contents of this header with garbage bytes, and allows it to continue to the overt site. If the X-Slitheen header contained upstream data to be proxied to a covert destination, the station simultaneously relays this data, and stores any responses in a queue of content for the Slitheen ID, to later replace leaf data from decoy sites to the client associated with the given ID. If the overt site has a large amount of images or a video stream, a large amount of content can be delivered to the client quickly, providing a low-latency censorship-resistant tunnel. To keep the size of HTTP requests consistent with the addition of the X-Slitheen header and data, the client may replace only non-essential headers or compress existing headers to be later decompressed by the relay station before they are forwarded to the overt site. If the existing headers are compressed, the X-Slitheen header is simply removed by the relay station.

When the Slitheen station receives downstream traffic from an overt site, it first decrypts the TLS record and inspects the HTML response for the content type of the resource. If the content type is not an image or video resource, the station will let the resource pass unaltered to the client. If the resource has leaf data, the station will replace the response body with data from the downstream queue pertaining to the Slitheen ID of the session and change the content type of the resource to "sli/theen" so that it is recognizable by the OUS. It then re-encrypts the modified record, recomputes the TCP checksum, and sends the packet on its way. If there is a shortage of

[2]Credit for the awesome name goes to Anna Lorimer.

downstream data, the station will replace the resource with garbage bytes, padding the response body to the expected length. When the OUS receives the resource, it sends all resources of the "sli/theen" content type to be processed and sent to the client's (real, not OUS) browser. All other resources, it processes in the usual manner. Note that the usage of a Slitheen ID allows covert data for a single client to split across multiple tagged flows.

A key part of maintaining the same network traffic metadata, such as packet sizes and timings, as a regular access of the decoy site is to replace the leaf data on a packet-by-packet basis. Whenever a packet arrives at the relay station from the overt destination, the relay station will immediately decrypt, possibly modify, re-encrypt, and forward the packet to the client with the same size and TCP state; only the (encrypted) contents of the packet will be possibly replaced with (again encrypted) censored content. We show that this replacement process introduces a minimal amount of latency, leaving the censor unable to detect the usage of a decoy routing system, and give the results of timing analysis in Section 3.3.6.

With our method for protocol appropriation, the usage of Slitheen is indistinguishable from a regular access of the decoy site. Regardless of advances in website fingerprinting or data analysis techniques, a censor cannot find distinguishing features that are not present.

3.3.3 The relay station state machine

While the replacement of the X-Slitheen headers and leaf content types is straightforward in theory, it is difficult to achieve in practice while also minimizing the latency introduced by the station. HTTP responses may be spread across multiple TLS records, and each record may contain multiple responses. Additionally, a record may be spread across multiple packets, leaving the station unable to decrypt a record to replace its contents or determine the content type of the responses it contains until the rest of the record has been received. Furthermore, packets may be delayed or dropped and arrive at the relay station out of order. Waiting for the receipt of an entire record before sending the observed packet to the client introduces an identifiable amount of latency, which may be used by the censor to detect the usage of Slitheen. Note that in addition to maintaining the sizes and timings of IP packets in our appropriated protocol, we also maintain the sizes and distributions of TLS records across packets.

A simple solution to receiving a record fragment is to forward the record unchanged and forego any possibly replaceable responses it contains. However, as record sizes for large image files are frequently large themselves, this results in a significant drop in the bandwidth available for delivering censored content. Instead, we can take advantage of counter mode ciphers such as AES-GCM and *partially decrypt* data as it arrives at the relay station. We can then partially re-encrypt the data before forwarding it to the client. The relay station can only verify and

recompute the tag of the ciphertext after it has received the entire record. At this point, if it verifies that the received tag is correct, it computes the correct tag for the replaced ciphertext. If the relay is unable to verify the tag of the received data, meaning it was originally computed incorrectly or tampered with on its way between the decoy site and the relay station, it similarly computes an incorrect tag for the client and lets the client's OUS handle it as it normally would.

However, partial encryption and decryption only works if the connection is encrypted using a counter mode cipher and if key parts of the packet arrive in the right order. If this is not the case, records will have to be forfeited and their contents not replaced. This will not affect visible traffic patterns, but will reduce the amount of replaceable leaf content and therefore the goodput of the system. To keep track of which data is leaf data, and therefore replaceable, and which data can be properly decrypted and analyzed, we implement a state machine at the relay station. This state machine is a composition of the TLS record state and the HTTP state of the decoy session.

Record state. When a packet is received by the relay station, the TLS record state of the flow determines whether the packet's contents begin with a new record, contain the contents of a previously processed record, or contain the remnants of a previous record and the beginning of a new record. The record's length, specified in the record header, determines how many full or partial records are contained in the current packet and how many bytes of subsequent packets contain the remaining contents of the record.

A flow can have an unknown record state if packets arrive at the station out of order. If the delayed packet does not contain any new record headers, the station is able to maintain the record state and processes the received packet in the usual manner, assuming the eventual receipt of the missing packet. However, if the delayed packet contained the beginning of a new record, the station has lost the record state and can only regain it after the missing packet arrives. While the record state is unknown, the station is unable to encrypt modified records for the client, as it does not know the lengths or contents of the record(s) in a received packet.

HTTP state. The station also maintains information about the HTTP state of each flow, indicating whether the next record will contain all or part of a response header, or response body. We give the state machine for HTTP responses in Figure 3.4. The end of a header is determined, as specified in RFC 2616 [FGF+99], by the receipt of two consecutive carriage return and line feed characters (CRLF): one to signify the end of the last header field, and one to signify that there are no more header fields in the message. The length of the response is determined by the status code of the response, and the transfer encoding (in which case the length is updated with each subsequent "chunk") or the content length. The Content-Type header indicates to the station whether the subsequent response should be replaced.

The HTTP state of the flow is updated upon the receipt of a new record header. Depending on

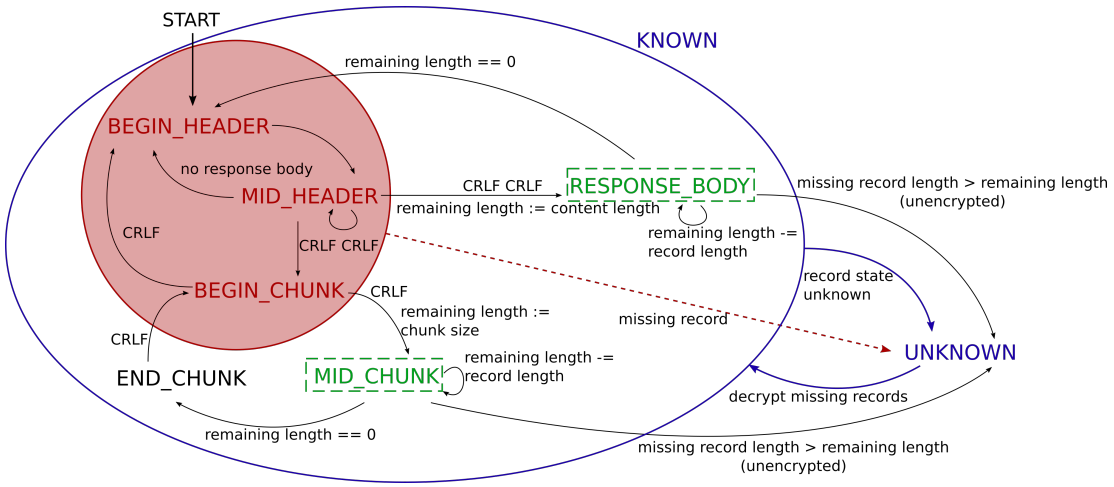


Figure 3.4: A flow may be in one of several TLS (blue) and HTTP (red, green, and black) states. When a new packet arrives that allows the relay station to find the beginning of a new TLS record, the station uses the record's length, its (possibly) decrypted data, and the length of the packet to determine the next HTTP state. States in the shaded red circle must be decrypted to decide the next state. If the flow is in a red, shaded state when the relay receives a partial TLS record it cannot decrypt due to missing data, the flow will enter into the UNKNOWN state until the remainder of the record is received and decrypted. This is represented by the dashed red arrow. States in green dashed boxes indicate states where data may be replaced. If the HTTP header showed a leaf content type, the relay station will construct a new record to replace the one(s) it receives. A flow with an HTTP state of UNKNOWN may recover its state by reconstructing partial or missing records and analyzing the decrypted data, along with the previous known state.

the HTTP state, a record does not need to be decrypted in order to be replaced. When the station receives a new record, it checks the record header to determine whether the record is contained in the TCP segment and may be decrypted, or whether the record is spread across multiple packets. It then determines, based on the HTTP state, whether the record may be replaced. If the HTTP state and the record's length indicates that it contains only a replaceable HTTP response body, the station will then construct a new record of the same length and fill it with downstream data from the client's queue. After encrypting the modified record, it sends the first part, matching the length of the record fragment in the received TCP segment, and stores the remainder of the modified record to replace the data in subsequent packets. After the entire record has been sent, the next TCP segment data will contain the header of a new TLS record.

It is possible for a packet to arrive at the station out of order, resulting in the relay station being unable to decrypt a record if it has not yet received the GCM nonce. In that case, if the received packet contains information about the response length or content type, the HTTP state of the flow will be unknown until the station receives the beginning of the record and decrypts it. In this case, the contents of the record will be forwarded immediately to the client, without modification, and a copy saved by the station to be decrypted when the beginning of the record arrives. Upon its receipt, the station can re-evaluate the HTTP state of the flow.

3.3.4 The Slitheen tunnel protocol

The tunnelled censorship-resistant data has the potential to arrive at the relay station and client, respectively, in a different order than it was written, as it can be distributed across multiple TCP connections to different overt sites. A counter allows each party to process the covert data in order, acknowledge the receipt of data, and retransmit data that was lost. The client and relay station re-order chunks of censorship-resistant data by sending it along with a header that specifies the length of the data, as well as TCP-style sequence and acknowledgement numbers to indicate the order in which it should be processed and allow for the retransmission of dropped data. If, as is the case with Slitheen, the appropriated protocol is layered on top of TCP, we can rely on the underlying acknowledgement mechanism to prevent the loss of covert data chunks within a single, successful TCP connection. However, since the delivery of censorship-resistant data is spread across multiple TCP connections that may be reset by either end, we rely on acknowledgements to retransmit data that is lost in the termination of a TCP connection.[3]

The Slitheen header consists of a 4-byte sequence number, a 4-byte acknowledgement number, a 2-byte stream ID that indicates which connection to a covert server the data belongs to, the

[3] Acknowledgements and the retransmission of covert traffic have not yet been implemented in the current version of Slitheen.

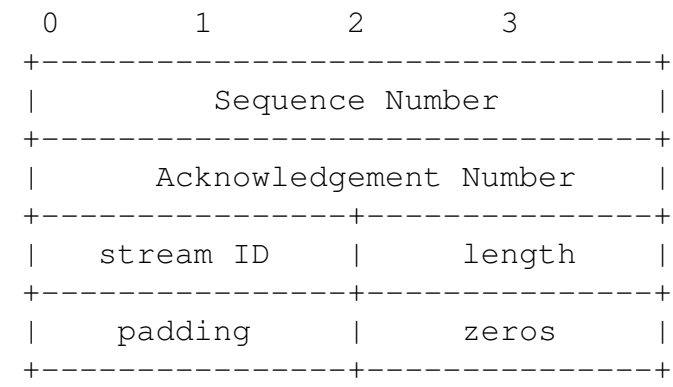


Figure 3.5: Slitheen header format. Each row represents 4 bytes.

2-byte length of the covert data chunk, and the 2-byte length of randomly generated padding. The last remaining 2 bytes of the header are padded with zeros. The header is followed by a chunk of covert data and/or garbage bytes, as specified by the header. The Slitheen header format is given in Figure 3.5.

Both the header and the body of the Slitheen chunk are encrypted to prevent an attacker from decrypting the covert traffic by comparing ciphertexts on both sides of the relay station in an effort to detect Slitheen or recover the censorship-resistant traffic. The extra layer of authenticated encryption also prevents an attacker from tampering with the covert traffic, ensuring it arrives at the client or relay station unmodified.

When the relay station receives TLS records, it decrypts, possibly modifies, and re-encrypts the record. Some TLS modes of operation, such as AES-GCM (see Figure 3.6), rely on a public *nonce* in addition to the secret key. It is important to the security of AES-GCM that two different messages are never encrypted with the same nonce and the same key. Many implementations of AES-GCM mode for TLS use sequential nonces for each message, and so when the relay station replaces message contents with covert data, it must reuse the nonce to avoid flagging to the observing censor that the censorship resistance system is in use. However, this presents a security problem as a third party capable of observing the ciphertext on both sides of the relay station can exploit patterns in the underlying plaintext to decrypt both ciphertext messages and modify the underlying plaintext without detection.

Although this attack falls outside the usual threat model for decoy routing, in which we assume that the censor is unable to compare traffic on both sides of the relay station, this puts vulnerable users of our system at risk to other attackers.

The original plaintext, P_1, will contain part of the HTTP response body of the original overt image, while the modified plaintext, P_2, will contain covert data destined for the user. The corre-

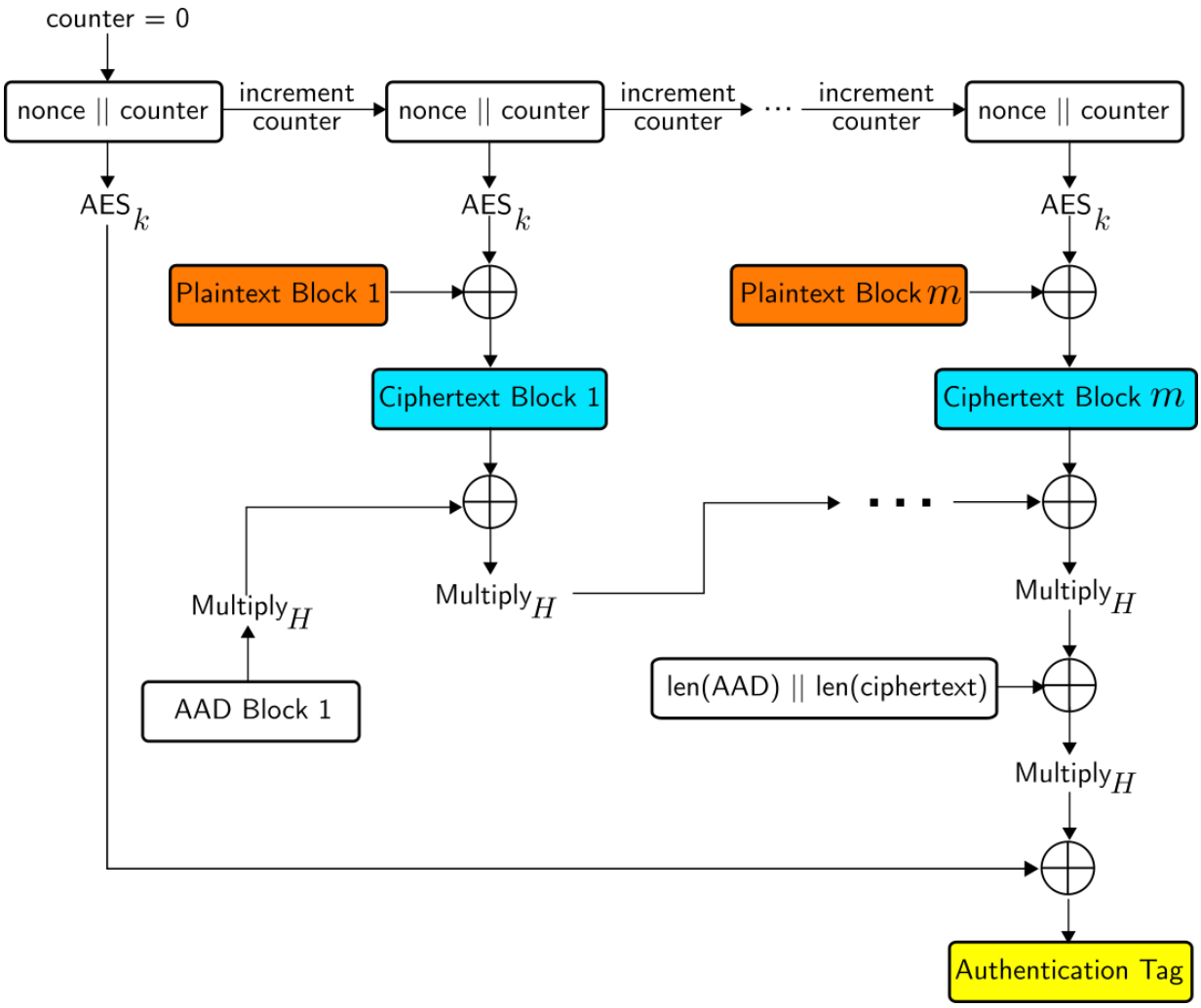


Figure 3.6: AES in Galois Counter Mode (GCM) [MV05]. The counter is initialized to zero and incremented for each encryption step, in which the nonce is concatenated to the counter and encrypted with the key k. This encrypted block is XOR'd with a plaintext block (shown in orange) to produce the corresponding ciphertext block (shown in blue). Additional Authentication Data (AAD) and the ciphertext blocks are multiplied by the hash key $H = AES_k(0)$ and XOR'd together to produce the authentication tag (shown in yellow).

sponding ciphertexts (limited to 1 block each for simplicity), seen by an observer, are computed as:

$$C_1 = E_k(n||0^{31}1) \oplus P_1$$
$$C_2 = E_k(n||0^{31}1) \oplus P_2$$

where E is AES encryption, and n is the nonce. The observer can then compute $C_1 \oplus C_2 = P_1 \oplus P_2$, and then exploit patterns in the underlying plaintexts to recover both P_1 and P_2. If the client was using Slitheen to browse a plaintext covert site, this two-time pad attack is trivial. In addition to breaking the client's confidentiality, an attacker can also modify the plaintext and compute the correct authentication tag [Jou06]. Given the ciphertexts C_1 and C_2, as shown above, and the corresponding authentication tags (where A is one block of AAD for simplicity):

$$T_1 = ((A \cdot E_k(0) \oplus C_1) \cdot E_k(0) \oplus L) \cdot E_k(0) \oplus E_k(n||0^{32})$$

$$T_2 = ((A \cdot E_k(0) \oplus C_2) \cdot E_k(0) \oplus L) \cdot E_k(0) \oplus E_k(n||0^{32})$$

where $L = \texttt{len}(A)\|\texttt{len}(C)$ and multiplications are performed in $GF(2^{128})$.

The adversary can compute:

$$E_k(0) = \sqrt{\frac{T_1 \oplus T_2}{C_1 \oplus C_2}}$$

and from that, since the additional authentication data A is known:

$$E_k(n||0^{32}) = (((A \cdot E_k(0) \oplus C_1) \cdot E_k(0) \oplus L) \cdot E_k(0)) \oplus T_1.$$

This gives them everything necessary to compute their own tag for an arbitrary ciphertext C_3:

$$T_3 = ((A \cdot E_k(0) \oplus C_3) \cdot E_k(0) \oplus L) \cdot E_k(0) \oplus E_k(n||0^{32})$$

In the event that the user's censorship-resistant traffic is encrypted with TLS, the consequences of both of these attacks are mitigated. An adversary would be unable to decrypt the client's censorship-resistant data, and any tampering would be detected in the TLS records sent between the client and the covert site. However, an adversary could use this to perform a targeted denial of service attack against decoy routing users. This attack is exceptionally damaging when the user of Slitheen is browsing a plaintext covert site, giving a third-party observer the ability to determine not only that the client is using Slitheen, but also *what covert data they are receiving*.

To defend against this attack, we propose adding an additional layer of authenticated encryption under the ciphertext on the downstream side of the relay station, both in order to make it indistinguishable from random, and to protect against modification. We chose this method rather

than simply re-encrypting with a different, randomly generated, nonce as a censor could detect the usage of a non-sequential nonce. The keys for this "superencryption" step may be derived from the client's Slitheen ID or the tag used to derive the TLS master secret.

Both the client and the relay station generate two superencryption keys: one to encrypt the 16-byte Slitheen header and another to encrypt a variable-length covert data body. The header (being only one block) is encrypted using AES in ECB mode. The censorship-resistant traffic in the covert data body is encrypted using an authenticated encryption mode such as AES-GCM that is indistinguishable from random by an attacker.

Upon the receipt of a new chunk of covert content, the client or relay station will first decrypt the Slitheen header and extract the length of the covert data chunk. The client should verify that the counter is as expected and the padding at the end of the header exists. The client can then decrypt the covert data and send it to the client's browser.

3.3.5 Implementation

We developed a proof-of-concept implementation of our system in a combination of C and C++. This implementation serves to demonstrate that our design behaves as expected, and provides a basis for our evaluations in the following section. Our code is available online for reuse and analysis.[4]

Client

Our implementation of the client consists of two distinct parts: the overt user simulator (OUS) that repeatedly connects to decoy sites, and a SOCKS proxy front-end that relays SOCKS connection requests and data between the client's browser and the OUS. The OUS takes data from the SOCKS front-end and inserts it into X-Slitheen headers of outgoing HTTP requests. It then takes downstream data from the received resources of content type "sli/theen" and returns this to the SOCKS front-end, which then sends this data to the browser. To allow the browser to send multiple simultaneous requests, we assign a stream ID to each connection. When the relay station receives downstream data for a particular stream, it includes the stream ID along with the data in the replaced resource, allowing the SOCKS front-end at the client side to demultiplex streams from the data received from the OUS.

For the tagging procedure, we modified NSS, the TLS library of Firefox, a popular open-source web browser. Our modifications allow the client software to specify the value of the ran-

[4]The code is currently available at https://crysp.uwaterloo.ca/software/slitheen.

dom nonce in the ClientHello message, as well as supply a given value for the client DH parameter. Although there are many algorithms available to negotiate a TLS master secret, our proof-of-concept implementation only allows the use of the DH key exchange methods. The ciphersuites we implemented are: DHE-RSA-AES128-GCM-SHA256, DHE-RSA-AES256-GCM-SHA384, ECDHE-RSA-AES128-GCM-SHA256, and ECDHE-RSA-AES256-GCM-SHA384. We support all the standard elliptic curves used by Firefox, including curve25519.

Other ciphersuites could easily be added to our system to expand the range of overt sites used by the client, however we recommend limiting the implementation to known secure ciphers. In the event that an overt site does not support one of the implemented ciphersuites, the client's OUS will continue to make the connection and request resources as usual, but the connection will not be used to tunnel covert content.

We use Firefox[5] as the basis for our OUS, although other common, open-source browsers such as Chromium[6] could be adapted for use as an OUS as well. We describe our choice of browser and an analysis of the OUS more in the next recipe, along with a description of our modifications to Firefox and NSS.

The SOCKS front-end receives connection requests and data from the browser and writes it to a TCP socket to the OUS, in order to be processed and inserted into the appropriated browsing session. It assigns each new connection a stream ID and sends that along with the data to the OUS to send to the relay station. The SOCKS front-end reads downstream data from the OUS and first demultiplexes it by stream ID before sending it to the browser. We wrote the SOCKS front-end in approximately 1500 lines of C code.

Slitheen Relay Station

We implemented the Slitheen station in approximately 9000 lines of C code. The station is responsible for recognizing and processing tagged TLS handshakes, proxying data to censored sites, and monitoring and replacing upstream and downstream application data to overt sites. When the station detects a tagged flow, it saves the source and destination addresses and ports in a flow table, to later identify packets in the same decoy routing session. The station continues to passively observe the remainder of the TLS handshake and then uses the tag-derived client secret and observed server handshake messages to compute the TLS master secret for the session, saving it in the flow table.

After verifying the TLS Finished message from both sides of the connection, and replacing the downstream Finished message to signal to the client that the session has been appropriated,

[5]https://www.mozilla.org/firefox/

[6]https://www.chromium.org/

the Slitheen station begins to monitor HTTP GET requests from the client for upstream data. It stores any downstream response from the covert server in a censored content queue. Once the station receives the client's Slitheen ID, it saves this information in the flow table in order to later identify the stream IDs that can replace downstream resources. As Slitheen reuses the TCP/IP headers from the overt site, we have no need to modify kernel code to set up a forged TCP state (as Telex requires). Application data is simply swapped into TCP segments as they are read from the interface, and sent back out to their destination with a recomputed TCP checksum.

Once the client and the overt site have terminated the connection by sending a TCP FIN or RST packet, the station removes the flow from the table. We save session tickets and session IDs to allow clients to resume TLS sessions for subsequent requests to the same server.

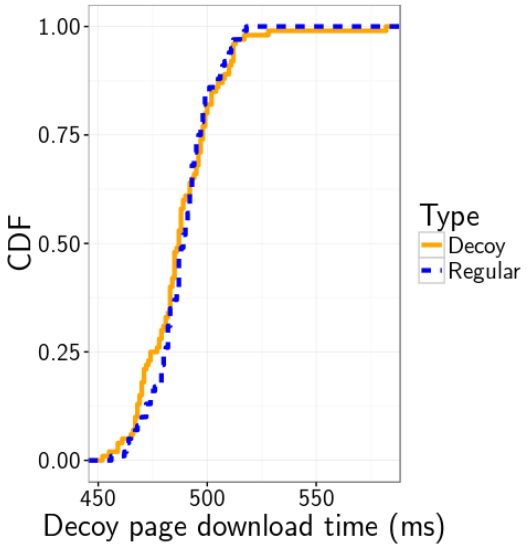
To keep the transmitted data consistent for retransmissions of TCP segments to or from the overt site, we maintain a queue of application data. When the relay station receives a retransmitted packet, it copies the appropriate amount of saved data into the packet before forwarding it to the client or overt site. This data is stored by the relay station until it expires, which occurs when the other side acknowledges the received data, or when the flow is terminated by either end.

3.3.6 Latency analysis

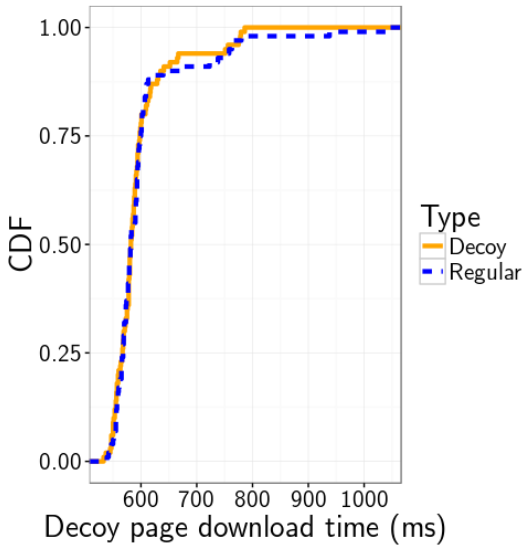
While the plaintext information and metadata such as packet sizes are by design identical to a regular access of the decoy site using our appropriation method, the remaining piece of metadata to consider are packet timings. In this section, we evaluate whether the latency added by the relay station in replacing leaf data adds a noticeable amount of latency to the connection to the loading of a decoy site.

Slitheen minimizes the latency introduced by the relay station itself by not waiting for data to be proxied to the covert destination, but rather queueing up previously collected proxy data for the client to be replaced as soon as incoming packets from the overt site arrive. Our results show that the encryption and replacement procedures do not add enough latency to identify the usage of Slitheen.

We measured the time it took to fully load the overt destination both as an overt site for tagged flows whose leaf resources were replaced with proxied data, and as a regular, untagged access. We tested two different overt destinations: www.wikipedia.org and gmail.com 100 times each from a client machine located at the University of Waterloo. We give the CDF of these load times in Figure 3.7. We performed a two-sided Kolmogorov-Smirnov test on the collected data in order to determine whether the relay station induced a different latency distribution for decoy accesses and measured a D-value of 0.11 and a p-value of 0.58 for www.wikipedia.org, and a



(a) Latency measurements for www.wikipedia.org



(b) Latency measurements for gmail.com

Figure 3.7: Cumulative distribution functions of the page load time of three overt destinations as both a decoy access and a regular access. The CDF shows a minimal difference in the latency distributions of the two types of access, and a K-S test fails to find a difference in the latency distributions due to the Slitheen station replacement and processing.

D-value of 0.12 and a p-value of 0.47 for gmail.com. These results indicate that the K-S test fails to find any significant difference in the latency distributions of the overt destination between its use as a regular or an overt site in a decoy routing session.

Defences against latency analysis attacks

While the page load times of popular sites accessed regularly and as decoy sites are statistically indistinguishable using a K-S test, a censor will attempt to identify decoy routing sessions on a connection-by-connection basis. We simulated an attack in which the censor compiles a database of expected latencies for both decoy sessions and regular browsing sessions for each overt destination by making 100 connections to the top 5 Alexa sites for each condition. We then calculated the precision and recall an adversary could achieve in classifying flows as decoy routing or regular sessions.

For these experiments, we used a Sandvine Policy Traffic Switch (PTS) 22600 capable of performing deep-packet inspection and flow diversion, and a relay station server with two 10 Gb/s connections to the PTS. If a tagged flow is detected by the PTS, it is diverted to the relay station server. The relay station server and client machine each used 8 cores and 2 GB of RAM. We conducted tests to see whether the divert functionality of the PTS and the implementation of the relay station added enough latency to tagged flows to allow a censor to reliably classify them as decoy routing sessions.

We also simulated a realistic network environment as closely as possible. For our tests, we gathered distribution statistics for Internet traffic from the Center for Applied Internet Data Analysis (CAIDA). We used the anonymized passive trace statistics through an OC48 link belonging to a large ISP in Chicago that handles approximately 2 Gb/s. We calculated the average flows/s

Table 3.1: CAIDA network traffic distribution statistics and measurements (CAIDA / our experiments)

Flow type	Average flows/s	Average Mb/s
HTTPS	4.43k / 4,330 ($\pm$ 90)	848.81 / 840 ($\pm$ 20)
HTTP	4.07k / 4,040 ($\pm$ 80)	814.51 / 800 ($\pm$ 20)
DNS	2.26k / 2,200 ($\pm$ 100)	N/A / N/A
TCP	792.74 / 790 ($\pm$ 20)	246.52 / 250 ($\pm$ 10)
UDP	527.53 / 530 ($\pm$ 20)	125.87 / 128 ($\pm$ 7)
Total	12.08k / 12,000 ($\pm$ 100)	2035.71 / 2010 ($\pm$ 40)

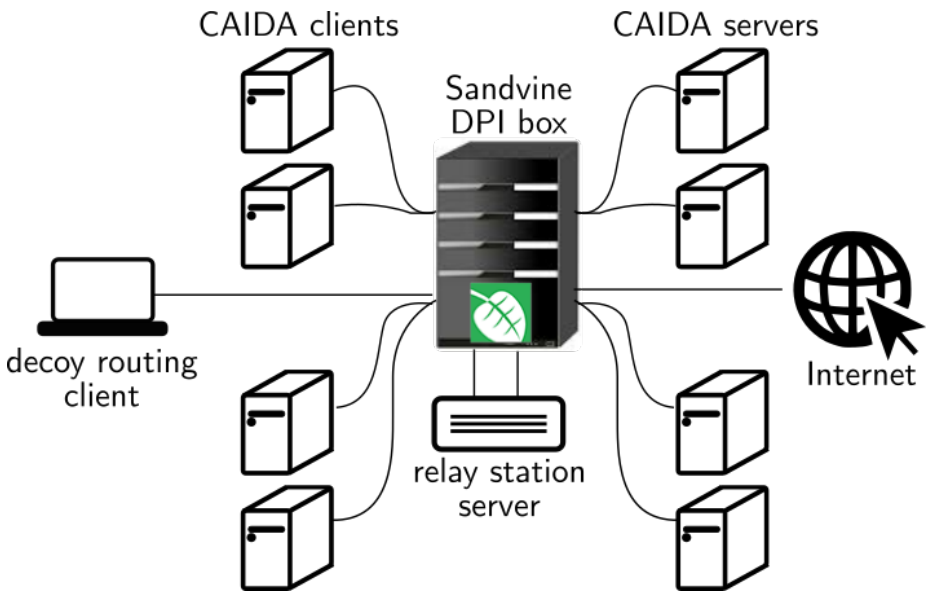


Figure 3.8: The network topology of our experiments. Machines designated as CAIDA clients and CAIDA servers were dedicated to sending background traffic through the DPI box, representative of traffic sent through an OC48 link of a large ISP according to statistics gathered from CAIDA.

and average Mbit/s for 5 major types of flows: HTTPS, HTTP, DNS, generic TCP, and generic UDP, gathered over the course of an hour on April 6th, 2016 [Cen16]. For each test, we sent a CAIDA-representative amount of traffic through the deployed relay station using four client machines and four server machines, on opposite sides of the PTS. To test the validity of our experiments, we measured the flow rate and bit rate for each type of flow at the client and server endpoints for 100 3-second captures. The results are given in Table 3.1. The experimental setup is shown in Figure 3.8.

We measured two different types of latency for each flow: the time it took to perform a full TLS handshake, and the average TCP acknowledgement time, or round-trip time (RTT) for application data. A censor will attempt to select a cutoff latency for each measurement type to identify decoy routing sessions. All flows with a higher latency than the cutoff value are classified as decoy routing sessions, while all flows with a lower latency are classified as a regular access to the overt site. We computed the CDFs of each type of latency for decoy routing sessions and regular accesses to each of the top 5 sites. From these CDFs, we can compute the true negative rate, τ, and false negative rate, ϕ, of an adversary for each possible latency cut-off point. We

calculate the *precision* of the censor as:

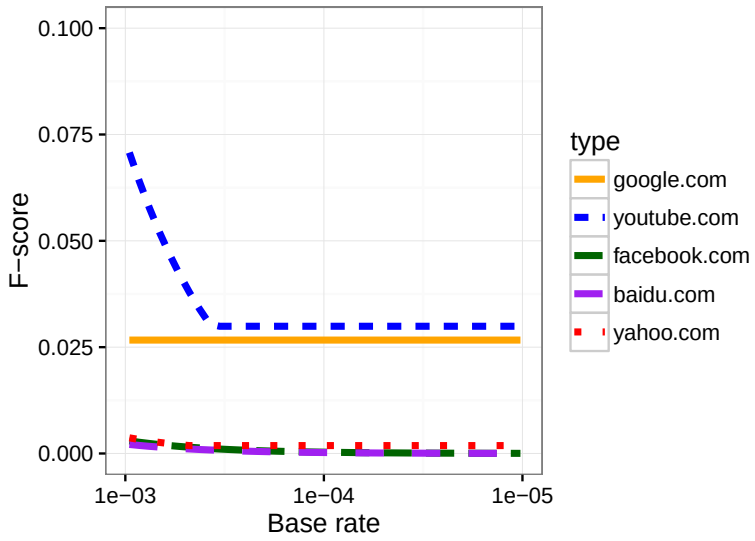
$$\text{precision} = \frac{\beta(1-\phi)}{\beta(1-\phi)+(1-\beta)(1-\tau)}$$

where β is the base rate of the incidence of decoy routing sessions. A typical censor would try to maximize precision, thereby minimizing the number of regular accesses to the overt site that are mistakenly identified as decoy routing sessions and blocked. By maximizing *recall*, a censor ensures that they are identifying and blocking as many decoy routing sessions as possible. The recall of the censor we calculate as the true positive rate:

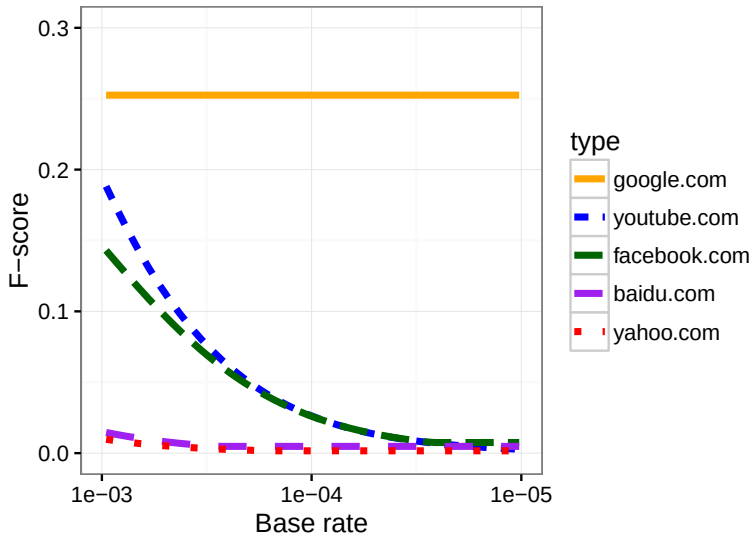
$$\text{recall} = 1 - \phi.$$

A censor can achieve a precision of 1, indicating that they do not incorrectly classify any regular accesses to the overt site as decoy routing sessions. However, often this means the censor can only identify a very small portion of decoy routing sessions while the majority will continue undetected. They can also achieve a recall of 1 by simply classifying all connections as decoy routing sessions and blocking them. For most censors, both measures are important so we define a censor's accuracy in terms of its F-score, the harmonic mean of the precision and recall values. Precision and recall can be weighted differently according to the individual goals of the censor, but we consider the default equal weighting in this thesis. For each value of β, the adversary will select a cut-off value that maximizes their F-score, given the latency distributions of each overt site. We plot the maximum F-score values based on the latency distributions for both the TLS handshake time and the TCP RTT for five of the Alexa top sites in Figure 3.9.

The maximum F-score a censor can achieve in identifying decoy routing sessions is very low for both types of latency. For the majority of sites, this value drops to almost 0 with a base rate of occurrence of decoy routing sessions of less than 10^{-4}, and meaning that if no more than one in every 10,000 connections to popular sites are decoy routing sessions, a censor is unable to reliably determine whether or not any given flow is carrying censorship resistance traffic. Even with a higher occurrence of decoy routing, the maximum F-score stays below 0.5 for most sites, making a reasonable censor that is unwilling to upset their population extremely wary of classifying and blocking potential decoy routing sessions. We note that some sites exhibit anomalous behaviour (e.g., google.com and youtube.com in their TCP RTTs and TLS handshake times, respectively). Such behaviour can be measured by the client, and those sites not selected as overt sites.



(a) The F-score of classifying flows based on TLS handshake time.



(b) The F-score of classifying flows based on the average TCP round trip time (RTT).

Figure 3.9: The maximum F-score (i.e., the harmonic mean of precision and recall) a censor can achieve in classifying flows as decoy routing sessions or as regular accesses to the Alexa top 5 sites. Precision is dependent on the base rate of decoy routing sessions. As a result, the less prevalent decoy routing sessions are, the lower a censor's accuracy in classifying flows.

Table 3.2: A comparison of the effectiveness of hiding traffic patterns of existing censorship circumvention systems that appropriate protocols for the tunnelling of censorship-resistant traffic. The features of the traffic patterns we consider are listed at the top of the table. The systems are split into three main categories: systems that were proposed before Slitheen [BG16] and our proposal of this technique, Slitheen itself, and systems that were proposed after Slitheen and take inspiration from our technique. A full circle ● indicates that the system has perfect indistinguishability by design from regular use for that traffic pattern feature. A half circle ◐ indicates the indistinguishability was experimentally determined, and an empty circle ○ indicates the system does not hide that traffic pattern.

	Plaintext protocol messages	Total amount of data sent	Network packet sizes	Network packet timings	Linkable connections
FreeWave [HRBS13]	●	○	○	○	●
DNS-sly [AFK16]	●	○	●	●	○
Domain fronting [FLH+15]	●	○	○	○	○
SWEET [HZCB17]	●	○	○	○	●
1st gen decoy routing [WWGH11, HNCB11, KEJ+11]	●	○	○	○	○
2nd gen decoy routing [WSH14, EJM+15]	●	○	○	○	○
Slitheen [BG16]	●	●	●	◐	●
DeltaShaper [BSR17]	●	◐	◐	◐	●
Waterfall [NZH17]	●	◐	●	◐	◐

3.4 Comparison to existing systems

Our protocol appropriation method far outperforms the traffic hiding techniques of previous systems. In Table 3.2, we show how our system performs in relation to existing systems on the traffic pattern features discussed in Chapter 2. A key advantage of our system is that the hiding of most traffic patterns (plaintext information, packet sizes, the total amount of data being sent, and linkable connections) is identical to a regular use of the appropriated protocol. The only traffic pattern feature that must be experimentally evaluated is the impact the parsing and replacement of leaf data by the man-in-the-middle has on the timings of network packets.

The unifying feature of all protocol appropriation techniques is the tunnelling of traffic

through the appropriated protocol in a way that maintains the plaintext (unencrypted) messages of the appropriated protocol, such as TLS handshake messages or VoIP connection messages. This is the feature that sets appropriation as a traffic pattern hiding technique apart from mimicry or obfuscation. One feature of the underlying censorship-resistant traffic that all previous systems fail to disguise is the total amount of data sent between the client and the decoy site. This feature has been shown to be sufficient for use in website fingerprinting to distinguish between covertly accessed sites [ZH16]. Furthermore, for systems that appropriate web browsing (decoy routing systems), this fingerprinting technique is trivial as the censor must only distinguish between the patterns expected for the decoy site and the patterns present in the client's traffic.

We can also assume that censors will look not just at a single flow or TCP connection, but multiple connections from the same client to try to detect anomalous behaviour. For example, in a regular web browsing sessions, client machines will typically make connections to more than one server, drawing image resources from CDNs rather than the main hosting site. Some previous censorship resistance systems hide this traffic pattern by virtue of the choice of appropriated protocol. While web browsing spawns multiple connections to different destinations, protocols such as VoIP or email are typically single connections to an expected site or service. DNS-sly fails the linkability check by requesting the resolution for domains that the client never visits, as these domains are used only to encode censorship-resistant traffic. In this table, we consider only the different network connections associated with loading a single decoy site. We discuss the loading of multiple decoy sites and the associated risks in the next recipe.

As of the writing of this thesis there have been two systems, proposed after our first publication detailing the appropriation technique used in Slitheen, that have taken inspiration from our method. These systems are listed in the table after Slitheen, and as seen, exhibit similar levels of protection for hiding traffic patterns.

Barradas et al. revisited the idea of appropriating Skype calls to relay censorship-resistant traffic to the user with a system called DeltaShaper [BSR17]. In their system, a client makes a connection through Skype to a DeltaShaper server that is seemingly just another Skype user, in a manner similar to FreeWave. Censorship-resistant traffic is encoded into payload frames, and embedded into carrier frames before being sent between the client and server. Carrier frames are obtained from pre-recorded Skype calls or via a webcam while the user is browsing censorship-resistant content. The degree to which DeltaShaper frames differ from a normal Skype call is tunable by the size of the embedded frame relative to the carrier frame. This tunability presents a trade-off between the bandwidth of the censorship-resistant communication and the indistinguishability of traffic pattern features.

DeltaShaper took inspiration from Slitheen by opening a VoIP connection to an endpoint and sending *real, captured* video and audio frames. Their suggested methods for overt user

simulation, or the generation of these frames, is to either record previous calls, or do a live capture of video and audio as the client is using DeltaShaper for censorship-resistant traffic. DeltaShaper also replaces leaf data in the sense that they are replacing a part of the carrier frame with a frame filled with censorship-resistant traffic. However, the size of this frame is not necessarily constant, and the replacement does not occur after the data has been made ready for network transmission. This results in an imperfect, yet experimentally evaluated hiding of features such as packet sizes and timings. Our method can be more fully applied to DeltaShaper; one would simply have to replace the pre-recorded carrier frames packet-by-packet immediately before they are sent on the wire to the server. This would maximize the bandwidth available for covert traffic as well as prevent any identifying traffic patterns from developing.

Our technique was directly adopted by Waterfall [NZH17], another third-generation system that achieves a high degree of resistance to traffic analysis. Waterfall performs the same leaf data replacement as Slitheen at a relay station between the client and the decoy site. While the main contributions of Waterfall concerned the deployablity of end-to-middle proxying, they made a few small changes to the appropriation of HTTP.

As opposed to inserting upstream content into the GET or POST requests for resources from the client, Waterfall inserts this data into GET or POST requests for which the browser expects an HTTP Redirect message. The data is then "bounced" back from the overt site to the relay station inside the Redirect message. HTTP Redirect messages are common, and although this requires the browser to use speculation or request these resources manually, they report such messages are common enough to not cause suspicion. This reliance on pacing the generation of redirect messages makes the linkable connections feature of traffic patterns experimentally, as opposed to perfectly, indistinguishable.

Waterfall also deviates from our method of replacing only resources that contain leaf data. They instead rely on the caching of non-leaf resources at the browser to produce the expected browser behaviour of issuing future connections and replace the entirety of all cached resources at the relay station. While they periodically update their cache, this also produces possibly identifiable traffic patterns to a highly capable and observant censor. If, for a particular client, a censor notices the full download of a resource it expects to be cached, this could raise their suspicion. The total amount of data sent between a Waterfall client and the decoy site thus differs for resources that would normally be cached.

3.5 Conclusion

Our technique for protocol appropriation makes the majority of traffic pattern features indistinguishable from the regular use of the appropriated protocol, and fully hides the patterns of the

tunnelled censorship-resistant traffic. The only traffic pattern that is not perfectly indistinguishable by design is the latency imposed by the man-in-the-middle in the parsing of the protocol messages and the replacement of leaf data. In our implementation of Slitheen, a decoy routing system that uses this method, we experimentally determined that this latency is unable to be used by a censor to reliably distinguish between the use of Slitheen and the regular access of decoy sites.

Our key contribution of this recipe is a method that can, and has, been applied to multiple protocols with similar results. It has been used and modified by two systems other than Slitheen to date and has the potential to be applied more broadly to the other systems predating Slitheen, discussed in Section 3.1.

This method is a significant step forward in the race between censors and resistors to, respectively, identify and hide censorship-resistant traffic. By hiding traffic patterns in a way that is indistinguishable from the regular use of an appropriated protocol, we are removing the reliance on assumptions of the inability of a censor to analyze traffic patterns. This gives our method the potential to stand the test of time against the growing strength of censorship infrastructure as the ability to collect, analyze, and compare data increases.

The difficulty of the appropriation resides mainly in the difficulty of overt user simulation. For VoIP connections, the interaction of the overt user is in the production of audio or video data to send over the wire. For email communication, it is the composition of emails with plausible lengths that do not leak information about the censorship-resistant traffic. The latter may be more difficult to achieve. For web traffic, which we have focused on in this section, it is in the choice of and pattern of access to decoy sites. We leave a discussion of advanced overt user simulation for our next recipe.

Chapter 4

Recipe #2: Simulating compliance

The goal of censorship resistance is to disguise traffic patterns, avoid detection, and simulate compliance with the censor's content access policies. This simulation extends beyond a single session of the appropriated protocol. Censors can not only analyze the traffic patterns of a single session, but look at the patterns of multiple sessions in an attempt to identify likely users of censorship resistance tools, even if they are unable to identify a single usage. In this recipe we look at factors that leak the usage of censorship resistance tools, and provide steps to eliminate or minimize the ability of censors to detect or use this information.

Censorship resistance systems with strong defences to traffic analysis attacks are notorious for their high latency and low bandwidth [BG16, HZCB17, BSR17]. There is a trade-off in existing systems between security in the form of hiding identifiable traffic patterns, and user experience in terms of the latency and bandwidth of censorship-resistant traffic. While the previous recipe focused on hiding traffic patterns in a single use of the appropriated protocol, we now look at the user experience and security across multiple appropriated sessions. There is an additional interaction between security and user experience that works in our favour: the higher the bandwidth of the censorship-resistant channel, the fewer appropriated sessions are needed to tunnel censorship-resistant data (or, depending on the protocol, the shorter the appropriated sessions need to be). Multiple or extended sessions may introduce identifiable patterns to the censor, meaning that the fewer are needed to tunnel a sufficient amount of traffic to the user, the easier it is to simulate regular usage.

This recipe focuses on the development of client-side software and extends the simulation of compliance across multiple sessions of the appropriated protocol. We propose two steps to avoid suspicion and increase the user experience of protocol appropriation tools:

Step 1: Eliminate the fingerprintable features of protocol implementations.

Internet protocols have several implementations, and each implementation has multiple versions as bugs are fixed or parameters are changed over time. Choosing a popular implementation and updated version of the appropriated protocol is critical to avoiding the suspicion of a censor. Differences between implementations produce **fingerprintable** features in the network traffic that have been used in the past to block censorship circumvention tools [Fif17, Wil12]. Censors can enumerate the features they expect from the most commonly used protocol implementations, and then block traffic that deviates from the expected features. This can effectively block censorship resistance tools without causing a significant amount of collateral damage. Updates to the implementation should be performed regularly to keep up with the version used by the majority of users in the regular usage of the protocol. We discuss the elimination of fingerprintable features due to implementation variance in Section 4.1.

Step 2: Find uses of the appropriated protocol that maximize bandwidth.

Although we are only replacing the leaf data of the appropriated protocol, high-bandwidth protocols typically have large amounts of leaf data available. However, the detection and replacement of this leaf data can be complicated and require the man-in-the-middle to maintain complex state machines in order to accurately parse and replace the overt traffic. We discuss maximizing the replacement of secure web browsing content through the usage of video streams in Section 4.2.

Together, we use these steps to make Slitheen both user friendly and more resistant to detection. We provide an evaluation of the performance of Slitheen from a user perspective and a comparison to recently proposed circumvention systems that also use protocol appropriation to disguise traffic patterns. We show that despite the potential for future improvements, Slitheen already delivers censorship-resistant content at rates that are usable and exceed those provided

by some recently proposed systems, while providing stronger security guarantees against traffic analysis attacks that will stand the test of time.

4.1 Eliminating fingerprintable features

There is a history of censors using browser and protocol fingerprinting techniques to identify and block censorship circumvention tools that dates back to the beginning of the cat-and-mouse game. The first reported use of DPI to detect variations in TLS implementations occurred in 2011 [Wil12], shortly before the emergence of active probing. A customized configuration of TLS that proposed a more restrictive set of cipher suites in the ClientHello message sent by Tor clients allowed censors to distinguish between connections from regular browsers and connections to the Tor network. Since only Tor clients used these cipher suites, the GFW was able to block these connections without causing collateral damage.

Although this technique was quickly discovered and defended against, circumvention tools still fall victim to these attacks by using outdated versions of popular protocols to make outgoing connections. Fifield attributes the blocking of Meek in 2016 to the use of an outdated version of Firefox to make TLS connections to the Tor network [Fif17]. In order to defend against these attacks, protocol appropriation tools must not only use popular implementations of the protocol to tunnel traffic, but must also keep their systems up to date with recent versions of the chosen implementation.

Overt user simulation in Slitheen is done through the usage of a fully functional web browser that makes tagged connections to decoy destinations and fully loads the decoy webpage. It inserts upstream requests for censorship-resistant traffic in the client's GET requests, and replaces the leaf data of the decoy site's responses with downstream censorship-resistant content. To accomplish this we modified a web browser, and that browser's corresponding TLS implementation, to tunnel our censorship-resistant traffic.

In our first implementation of Slitheen we modified a current version of OpenSSL, a popular TLS implementation, and PhantomJS, a headless browser, to make decoy connections and insert and extract censorship-resistant traffic into the appropriated secure web browsing session. However, our modifications were easily fingerprintable due to our choice of base implementations. Due to its lightweight and headless nature, PhantomJS is frequently used for testing and metrics, but rarely used for regular web browsing. Figure 4.1 shows byte-level differences in the ClientHello message of the TLS handshake and initial HTTP GET request to a decoy site between our original choice of PhantomJS and Firefox version 52, a recent version at the time of writing this thesis. Selected differences are bolded and enumerated.

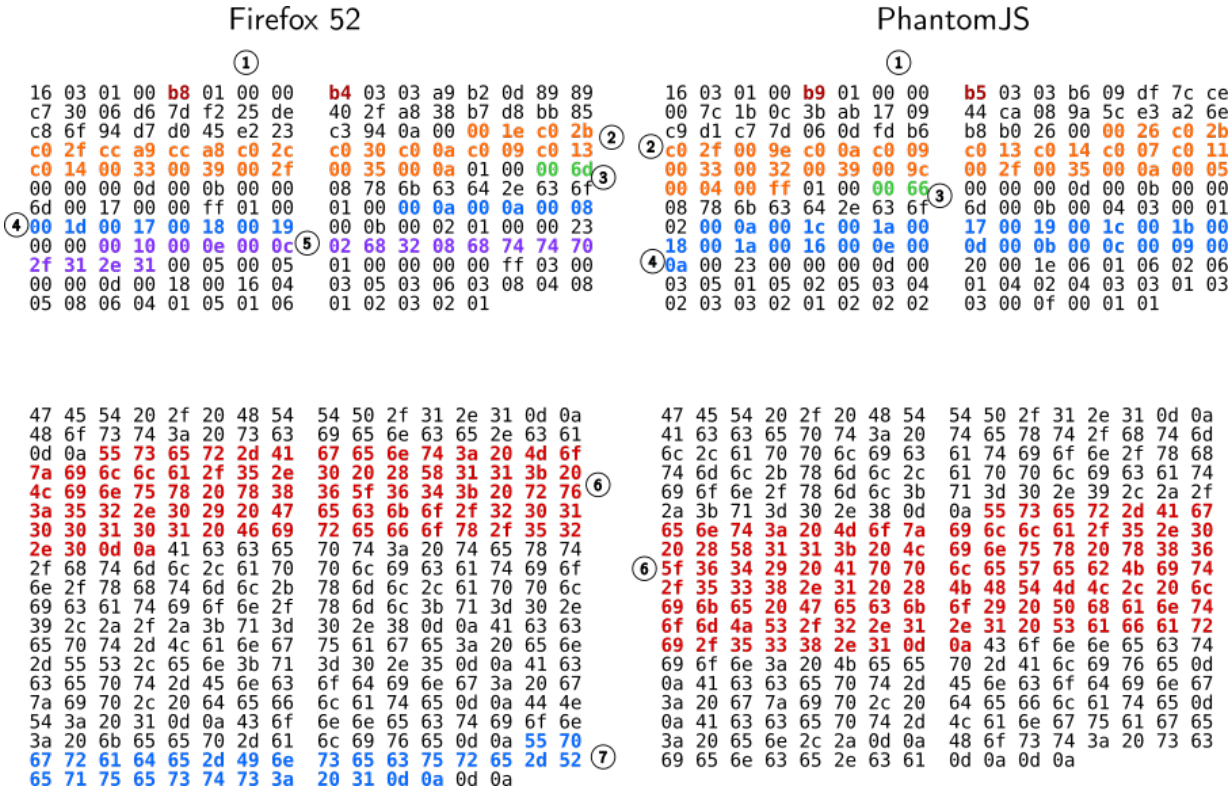


Figure 4.1: A byte-level comparison of the TLS handshake and HTTP GET requests of two different browsers: PhantomJS version 2.1.1 and Firefox version 52. Selected, fingerprintable differences are in bold and enumerated. Some differences, such as the random nonce, are expected and not bolded. Note that with TLS, the details of the differences in the HTTP requests are encrypted, however, a censor can see the difference in the size of the request.

The first set of bytes corresponds to the ClientHello message sent by each browser to initiate the TLS handshake. PhantomJS uses OpenSSL to issue TLS handshakes, while Firefox uses Mozilla Network Security Services (NSS). Aside from the overall difference in the length of the message (1) due to the difference in content, each message also suggests a different set of ciphersuites (2), a different number of TLS extensions (3), proposes different elliptic curves in the `elliptic_curves` extension (4), and lists a different set of extensions (5). All of these differences are in plaintext, are visible to the censor, and can be used to distinguish between commonly used browsers (Firefox), and browsers that are uncommon and therefore can be blocked with minimal collateral damage (PhantomJS). Note that some differences, such as the random nonce of the ClientHello message, are expected.

The next set of bytes in Figure 4.1 shows differences in the HTTP GET requests sent by each browser. Although these requests will be encrypted in our system, and thus the bytes unreadable by the censor, the selected differences result in different sizes of HTTP requests, which a passive censor can detect by observing the ciphertext. Each browser has different values for HTTP headers such as the User-Agent header (6). Browsers also include a different number and set of headers (7), which can greatly impact the length of the outgoing message.

To avoid browser fingerprinting, we re-implemented our overt user simulator by making modifications to Firefox version 52, taking care to do so in a way that does not produce identifiable differences in network traffic. We refer to our modified browser as Slifox in the remainder of the thesis, a name coined by Anna Lorimer who worked on Slitheen for eight months as an undergraduate research assistant.

4.1.1 Implementation details

Slifox includes modifications to Network Security Services (NSS), Mozilla's implementation of TLS, to implement the Slitheen tagging procedure. We also modified the media and http networking portions of the code in order to insert upstream censorship-resistant traffic into the headers of outgoing requests, and detect downstream censorship-resistant data in the HTTP responses. In total, we added 16 new files, and modified 1105 lines of code over 36 existing files in the Mozilla source code.

To implement the tagging procedure in NSS, we modified only 321 lines of code over 13 different files. We also added 5 new files with a total of 2231 lines of code. These modifications allow us to insert a tagged value in place of the ClientHello random nonce, generate the client's secret key for use in the ClientKeyExchange message, and look for a modified Finished message. The modifications to the content management parts of Firefox allow us to communicate with a SOCKS proxy front end, to get SOCKS CONNECT requests and censorship-resistant traffic

from the client, insert censorship-resistant traffic into the upstream HTTP GET requests, and look for resources with the content type "sli/theen", which are then sent back to the SOCKS proxy frontend. To accomplish this, we modified 484 lines of code over 23 files and added 11 new files with a total of 1205 lines of code.

We implemented our changes to be compatible with Electrolysis (e10s), Mozilla's multi-process framework for giving each tab its own process. To do so, we used the Inter-process Protocol Definition Language (IPDL) to specify a way for the content processes (that receive HTTP resources and therefore Slitheen data) to communicate with the main process that sends this data to the SOCKS proxy front-end. This, along with the minimally intrusive nature of our modifications, will help ease the process of updating the version of Firefox we use. This is critical to avoid browser fingerprinting attacks in the future.

Our modifications allow users to receive censorship-resistant traffic from any image content type, as well as WebM video and audio resources. While images are easier to replace, they provide a very small amount of censorship-resistant bandwidth. Video streaming, on the other hand is much more difficult to replace but provides a large amount of bandwidth to the user. We discuss the details of WebM resource replacement and the bandwidth results of using video streaming to tunnel censorship-resistant traffic in the next section.

4.2 High-bandwidth censorship-resistant traffic

The amount of censorship-resistant data a client receives in a single appropriated protocol session is dependent on the amount of leaf data available for replacement. The more protocol sessions that are required to transmit this data, the more vulnerable the client is to the scrutiny of a censor, the more load the system places at its deployment points, and the more latency the client experiences in viewing or interacting with censorship-resistant content.

In this section we discuss how we maximized the bandwidth available for censorship-resistant traffic in Slitheen by replacing video streams. We give measurements showing the throughput of our system and the user experience by showing the latency and bandwidth of censorship-resistant traffic. We compare the performance of our system to other censorship circumvention systems that appropriate allowed protocols.

4.2.1 Replacing image resources in Slitheen

In our first implementation of Slitheen, we only replaced HTTP resources with an image content type. Image resources are easy to replace as the entirety of the image is leaf content and does

not prompt the browser to produce additional network traffic as a result of its parsing. However, although some images are quite large, they do not provide a large amount of censorship-resistant bandwidth.

We visited the front pages of the Alexa[1] top 10,000 TLS sites that support our implemented ciphersuites and measured the amount of image leaf data available to a Slitheen client when using the site as an decoy. To collect these results, we used the PhantomJS headless browser to capture the size and content type of HTTP responses from each site. Note that while we use PhantomJS for these measurements, we implemented our OUS with Firefox, as described above. From the captured data, we measured the total amount of potential downstream bandwidth from each site as the sum of the sizes of all resources with an image content type. We note that this is the maximum amount of potential downstream bandwidth as some resources will not be replaced due to an incomplete deployment, overly abstract state machines at the relay station, and the occurrence of misordered or dropped packets.

We give a cumulative distribution function of the amount of image leaf data available for replacement for the Alexa top 10,000 sites in Figure 4.2. About 25% of sites offer 1 MB or more of potentially replaceable content. In total, we found that about 40% of the content from all 10,000 sites was leaf content, making less the half of the traffic from the decoy site to the user replaceable.

As mentioned in the previous recipe, leaf data can only be replaced by the relay station if key packets required to maintain the relay station's state machine arrive in the correct order. To more accurately measure the amount of downstream bandwidth for censorship-resistant traffic, we measured the bytes of leaf image data that were actually replaced with censorship-resistant content by the relay station when we made connections to decoy sites through our system. We accessed the Alexa top 500 TLS sites as decoys and counted the total bytes received, the total amount of leaf data, and the amount of leaf data that was replaced by the system. We give a CDF of the results in Figure 4.3.

Due to our method of partially decrypting incomplete TLS records, the difference between the amount of leaf data and the amount of data that was actually replaced is fairly small. This difference is due in part to misordered packets, but also imperfections in the state machine's ability to parse headers across multiple packets. With future improvements to the relay station's state machine, we can continue to close the gap between potentially replaceable content and replaced content.

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

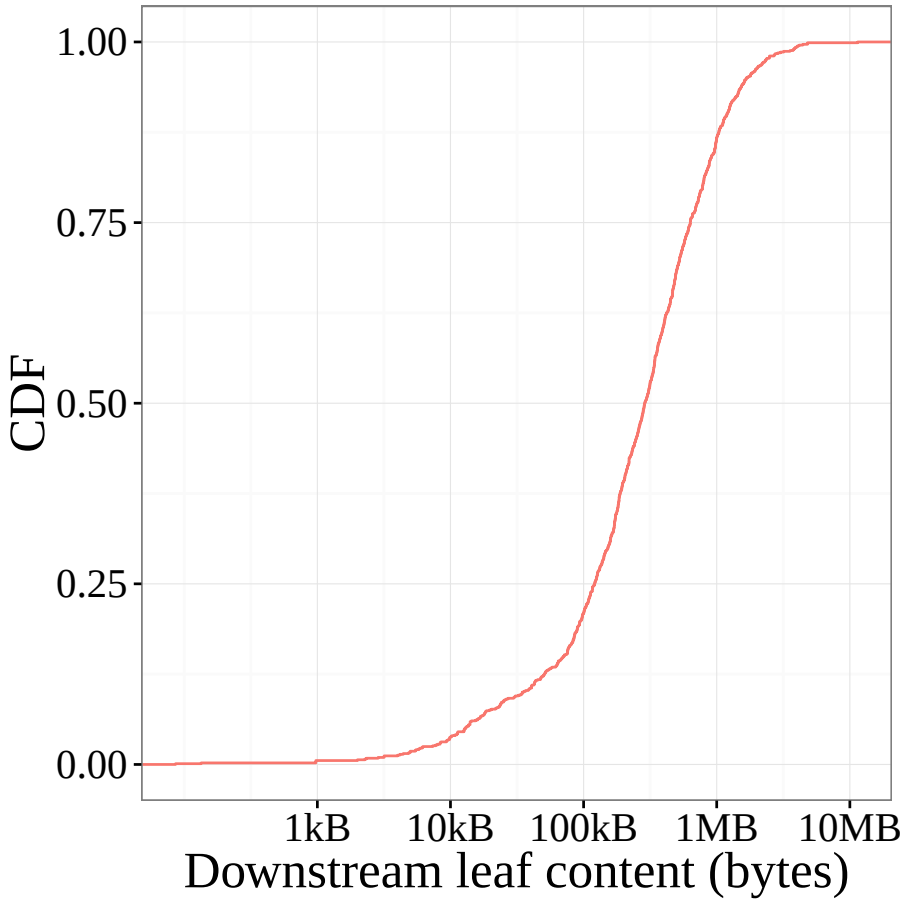


Figure 4.2: Cumulative distribution function of the potential downstream bandwidth for censorship-resistant data provided by the image resources of the Alexa top 10,000 TLS sites.

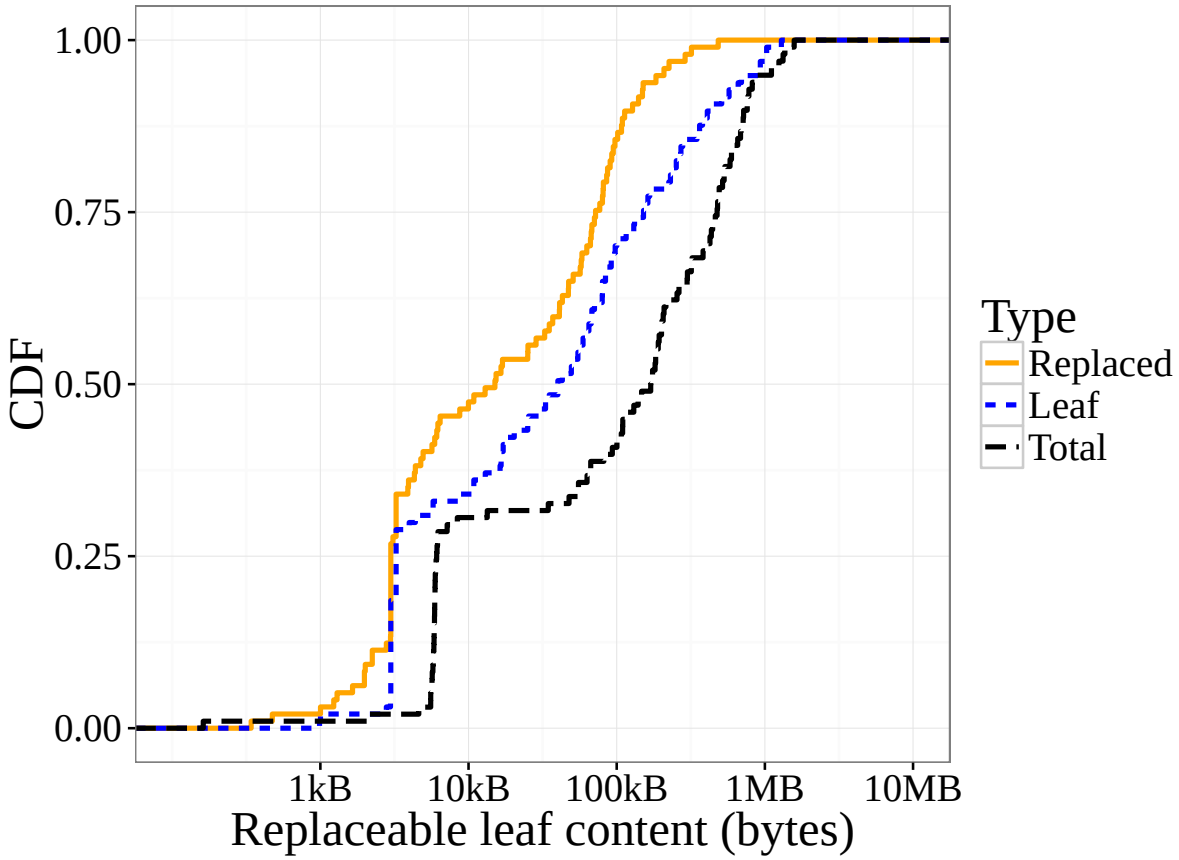


Figure 4.3: Cumulative distribution function of the image leaf data that was actually replaced by the relay station in our tests. We give the total number of bytes, the bytes of leaf content, and the bytes replaced with downstream covert content for each of the Alexa top 500 TLS sites.

4.2.2 Video replacement

Images provide an easy target for leaf data; however, multiple page loads of a decoy site are required to load even a small censorship-resistant webpage. Furthermore, if the client is interacting with the covert site, such as performing a TLS handshake, they need not only sufficient downstream bandwidth, but also enough back-and-forth communication between the OUS and the decoy sites to relay both sides of the TLS handshake messages.

A more ideal resource to use for tunnelling traffic is video streaming data. Many popular websites offer video streaming services, several of which are not currently blocked by large censors. Video streams offer a large amount of bandwidth, delivering large amounts of covert data to the user very quickly, and also involve frequent back-and-forth communication between the client's browser and the streaming service to periodically load the next part of the video. The prevalence of video streaming services and their usefulness in carrying large amounts of information has led many censorship circumvention systems to appropriate or mimic them to proxy censorship-resistant traffic to users [BSR17, HBS13, MHS16, LSH14, MMLDG12]. We note that while previous systems used video streaming, our system is the first to provide perfect resistance to traffic analysis attacks by replacing only leaf data with censorship-resistant traffic, and maximizing the amount of leaf data replaced.

There are a few practical challenges with appropriating video streams that might violate some of the assumptions of our threat model. In particular, we make the assumption that the video streaming sites exist outside of the area of influence of the censor. This is increasingly uncommon as many video streaming services host content in multiple regions and serve content from servers within the same region as the requesting client. It may be an increasingly difficult task to find popular outside streaming services or endpoints that will not draw suspicion from censors. Streaming services are also increasingly proprietary and ship their own obfuscated JavaScript to process and display the video content, making it difficult to modify the protocol for tunnelling purposes.

To replace video streams in Slitheen, we chose to begin with the WebM[2] container format for video and audio resources. We chose WebM resources due to the availability of clearly defined specifications and their popularity with streaming services such as YouTube. We implemented a WebM state machine in the relay station and modified the WebM state machine in the OUS (Slifox). As opposed to image resources, WebM resources contain metadata and timing information that browsers need to play the video and determine when to request more content from the decoy site. Therefore, we use our protocol appropriation recipe from Chapter 3 again here to appropriate WebM.

[2]https://www.webmproject.org/

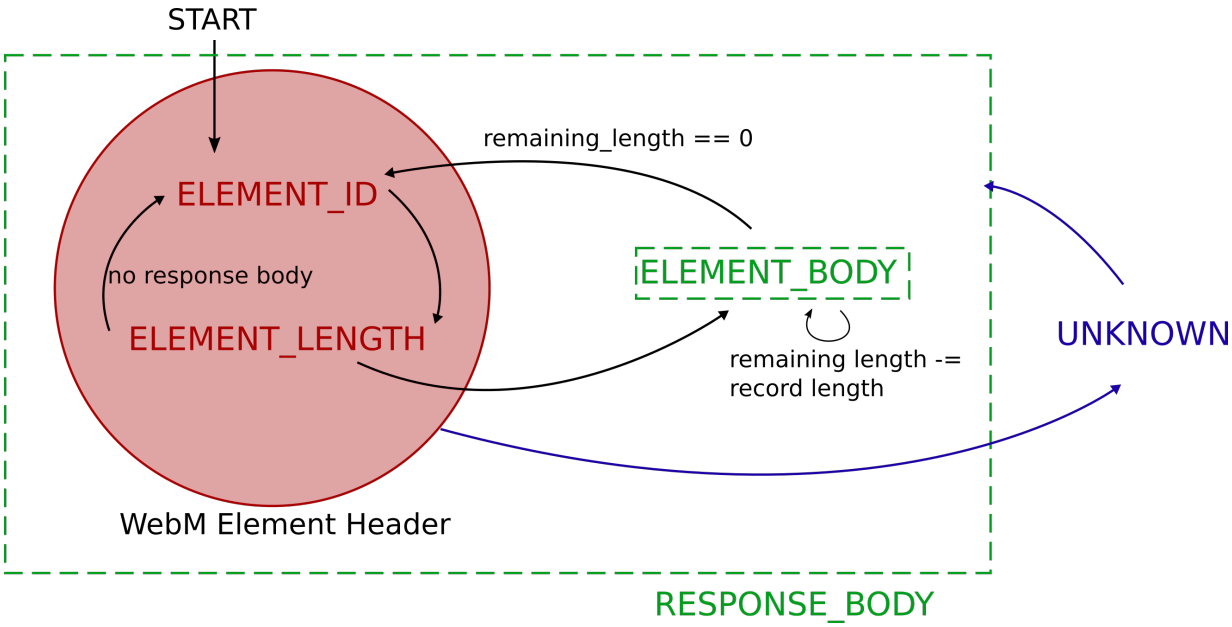


Figure 4.4: Modifications to the relay station state machine to parse WebM resources. The WebM state machine is composed with the HTTP state machine and is located in the body of HTTP resources (with or without a transfer encoding). The critical parsing region is shown shaded in red, where if a packet arrives out of order, the state is unknown until the relay station receives the missing packet.

Step 1: To start an appropriated video streaming session, we have Slifox access a video from a popular streaming service in the same manner that a regular user would. This includes performing a TLS handshake with the streaming service and navigating to a URL that corresponds to a video stream. For our tests and for the remainder of this recipe, we use YouTube as the decoy site although we acknowledge that YouTube is and has been blocked in several countries. Our results apply to any streaming service that uses the WebM format. The WebM project maintains a list of dozens of video libraries, hardware, platforms, and publishers supporting the WebM container format.[3]

Step 2: We replace only the leaf data of WebM containers. The WebM container protocol is a sub-specification of Matroshka[4]. Containers are composed of various levels of sub-containers, the elements of which are defined in no particular order. The container provides the browser with meta information, such as track numbers, timestamps, and seeking pointers. The leaf data

[3]https://www.webmproject.org/about/supporters/

[4]https://www.matroska.org/technical/specs/index.html

are the contents of video and audio frames, contained in Block elements, inside the main Cluster container of the WebM resource. Our state machine in the relay station, expanded to parse WebM resources inside of the HTTP resource body, parses the headers of WebM elements until it reaches a Block element. It then replaces the body of the block element, leaving the header with the length and timing information untouched.

The vast majority of a WebM resource consists of video and audio frames, which the relay station must parse carefully to avoid giving the browser corrupted meta information which will prevent the video from playing properly and produce identifiable network traffic patterns. We show our addition to the relay station state machine in Figure 4.4. This addition fits inside the RESPONSE_BODY and MID_CHUNK states of the previous state machine in Figure 3.4, which transitions to the UNKNOWN state in the case of misordered packets. When the relay station receives the body of a resource with content type "video/webm" or "audio/webm", it continues to parse the resource looking for leaf data. This involves parsing the element headers of the container, and skipping elements that contain metadata and not video or audo frames. We only replace simple blocks, with an element ID of 0xa3, and overwrite the element ID to be 0xef, an ID number that does not conflict with existing WebM elements. This allows Slifox to detect which blocks have been replaced with censorship-resistant data. The full state machine at the relay station is now a composition of five protocols, spanning multiple network layers: IP, TCP, TLS, HTTP, and WebM.

Step 3: To simulate a regular video streaming session, we require the OUS to process and play both video and audio frames in the way it normally would had we not replaced their contents. Many popular video streaming services, such as Twitch[5] and YouTube[6], ship their own video playback engine as JavaScript when users access the site. Luckily for our purposes, streaming services that use WebM still make use of Firefox's WebM parser and state machine, allowing us to access the censorship-resistant content and replace it with dummy video and audio frames. However if more browser functionality is moved to customized JavaScript in the future, this step will become much more difficult.

It is necessary to replace frames containing censorship-resistant traffic with dummy frames before they are sent to the playback engine to avoid errors or a change in behaviour due to the engine detecting malformed frames. We made modifications to the WebM parser in Slifox so that when a WebM resource is received by the OUS, the block elements that were successfully replaced by our relay station are discovered, their contents sent to the SOCKS proxy for delivery to the user's browser, and replaced with valid video or audio frames so that the OUS video player continues to request additional content. We chose a blank white video keyframe as a dummy

[5] www.twitch.tv

[6] www.youtube.com

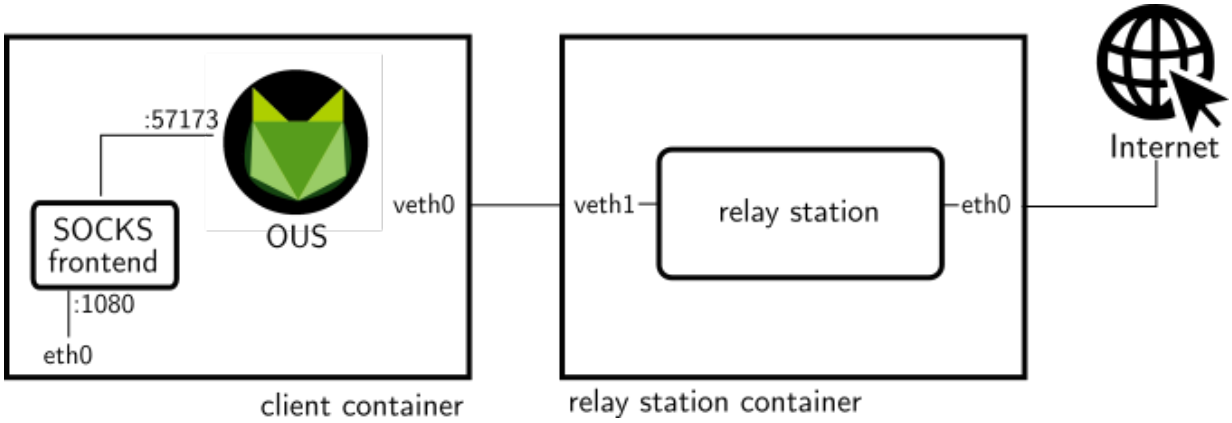


Figure 4.5: The experimental set up of our user experience tests. We run the client and relay code separately in two different Docker containers. Traffic to and from the client is routed through veth0, such that is passes through the relay station on its way to and from the outside Internet. This mimics a decoy routing deployment. The user can then connect to the client software on port 1080 to tunnel censorship-resistant traffic, similar to a normal SOCKS proxy.

video frame and an empty audio frame to replace audio resources. We do not need to keep the length of the received and dummy frames consistent as length does not significantly impact the processing and play time, and therefore the network traffic, of the video streaming session.

By replacing the video and audio frames of video streaming sessions, Slitheen becomes a very high-bandwidth, low-latency censorship circumvention system. In the subsequent sections, we present the results of experiments on the user experience of Slitheen.

4.2.3 Evaluation

We conducted several experiments to measure the effectiveness of replacing video content in Slitheen in terms of the overhead, latency, and the bandwidth available for censorship-resistant traffic. These experiments were run locally on a machine running Ubuntu 14.04 LTS. Our experiments used 16 cores and 16 GB of RAM. We ran the client and the relay station code in seperate Docker containers, networked together such that client traffic routes through the relay station on its way to the outside Internet.

We give our experimental set up in Figure 4.5. There are two Docker containers: one for the client and one for the relay station. These containers are networked together using koko[7] in

[7]https://github.com/redhat-nfvpe/koko/

a way that mimics a decoy routing deployment. Our OUS, Slifox, runs in the client container along with the SOCKS proxy frontend, connected to each other on localhost port 57173. The client container also exposes port 1080 to the host machine, which the SOCKS proxy frontend listens to, allowing the client to connect to it from the web browser of their choice.

Using this configuration, we ran a series of tests, passing censorship-resistant traffic through our system to determine the overhead, the available bandwidth for censorship-resistant traffic, and the latency of requests for censorship-resistant content.

Overhead

We calculate the overhead of our system to be the amount of extra traffic needed to tunnel censorship-resistant traffic. The overhead of our system is necessarily nonzero, as we are only replacing leaf data, and not headers or metadata needed by the protocol to preserve the illusion of the client's regular use of video streaming services. Our goal is to minimize the overhead as much as possible, both to improve the user experience of censorship-resistant browsing, and also to minimize the number of decoy connections needed to provide a sufficient amount of censorship-resistant bandwidth.

We made 100 connections to the top-trending videos in Canada on YouTube. For each connection we measured the bytes of leaf data that the relay station replaced (or the bytes of censorship-resistant traffic delivered to the client) and compared that to the total number of bytes received by the client in loading the video. We found that the overhead was approximately $5\times$, meaning that about a fifth of the throughput of the system is censorship-resistant traffic.

Our system has a large overhead compared to other protocol appropriation systems such as meek or SWEET that do not hide the metadata features of traffic patterns. While we do not have exact numbers on the overhead of these systems, we can assume it to be quite low. Our overhead is closer to, yet still larger than, that incurred by DeltaShaper, another protocol appropriation system that tunnels traffic through video-based protocols (although we provide strong defences against traffic analysis attacks). Still, the overhead of our video streaming web browsing traffic is a vast improvement over the overhead presented in Figure 4.3. From that data, we calculate the mean overhead to be 20 ($\pm$ 50) times the goodput with the large variance caused by the different ratios of image content to total content provided by each overt site. Thus, by moving from only replacing the images of decoy sites to replacing video resources, we have cut the overhead of our system down to approximately a quarter of what it was previously.

The overhead produced by our system can be reduced by further optimizations of the relay station state machine. Some HTTP resources, TLS records, or video/audio frames are not replaced by the relay station due to packets arriving at the relay station out of order, or headers

being split across multiple packets. We leave a further reduction of the overhead of our system to future work.

Latency

We measured the latency of the censorship-resistant traffic through our system as the time from the user's first HTTP GET request to the covert site, to the time the first byte of the HTTP response is received. This requires several steps to be performed at the client, relay station, and covert site. The client must first insert a SOCKS CONNECT request and a request for censorship-resistant content into the HTTP GET request of an appropriated browsing session. Next, the relay station needs to receive this request and make a connection to the covert site through which it can proxy the GET request. The covert site then responds and the relay station queues the censorship-resistant data. When downstream leaf content arrives at the relay station, it inserts the queued data into downstream leaf resources, which the OUS then extracts and sends to the user's browser.

We accessed the Alexa top ten webites ten times each, using a different top ten trending YouTube video to tunnel each access. A CDF of the latencies of all 100 accesses is given in Figure 4.6. The mean latency of censorship-resistant traffic was 13 ($\pm$ 11) seconds. The large variance in latency is due in small part to the packet distributions of different videos, causing the relay station state machine to forfeit some video or audio frames. The buffering behaviour of the OUS video playback engine has a much larger impact on the latency of covert traffic through the system. The OUS will request large chunks of video at a time and then wait until enough of the video has played before requesting more. This leads to long periods of time during which the OUS is not receiving censorship-resistance traffic.

Although some connections took almost a minute to load, over half of the connections to covert sites had a latency of less than 10 seconds to return the first resource. The median latency of our experiments was 7 seconds, making it only 2 seconds slower than SWEET, but with much stronger resistance to traffic analysis attacks. Some connections had latencies of only 2 seconds, suggesting that the proper choice of video and improvements to the relay station would make our system surpass existing systems in terms of the user experience of browsing censorship-resistant traffic.

Bandwidth

To measure the overall bandwidth of censorship-resistant traffic, we accessed a large covert site (a 500 MB file download over HTTP) through our system, using the top ten trending videos on

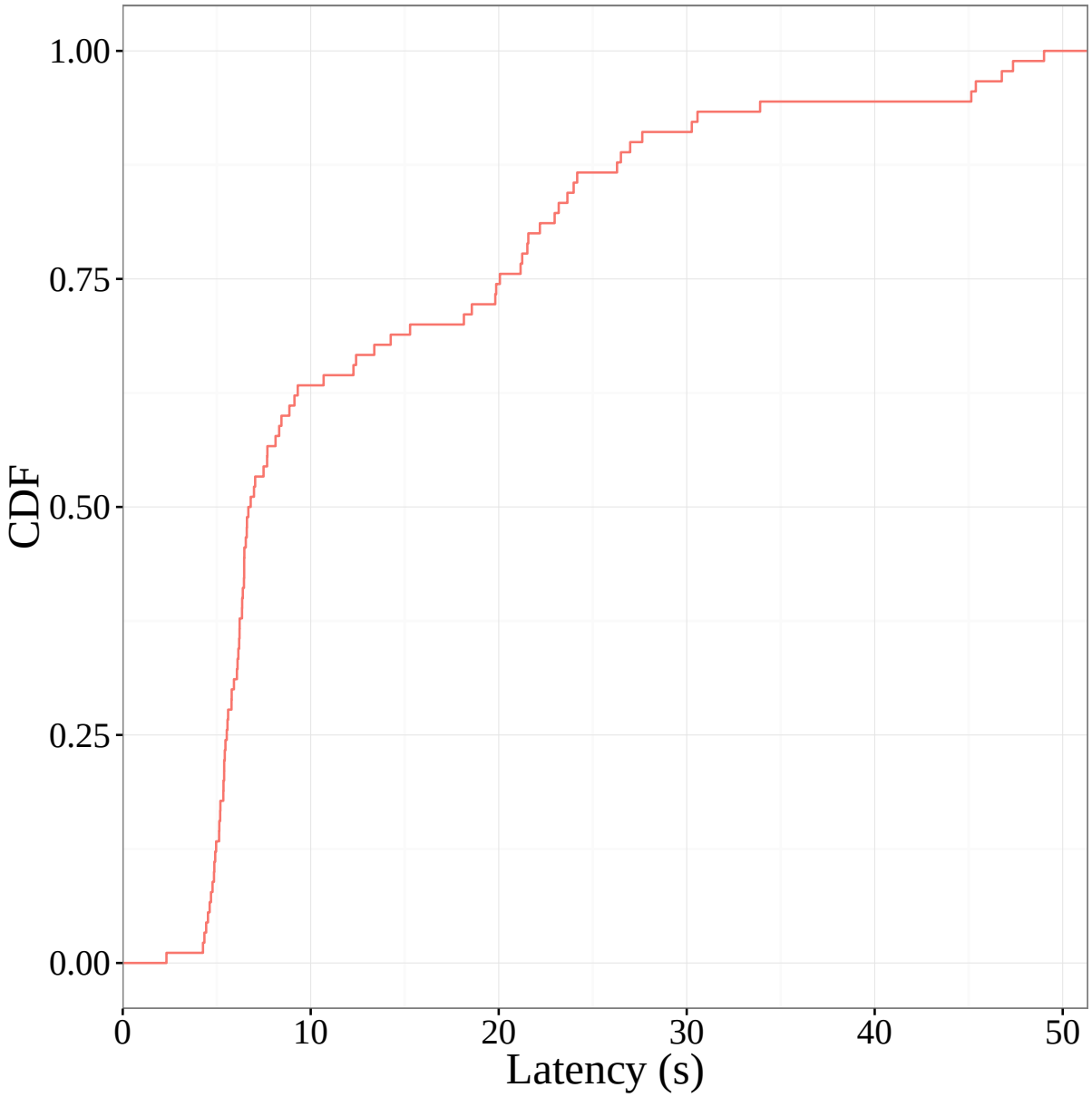


Figure 4.6: A CDF of the latency, or the time from a user request for censorship-resistant data to the corresponding response from the covert site. We plot results from loading ten different censorship-resistant sites, each tunnelled through the top ten trending videos on YouTube.

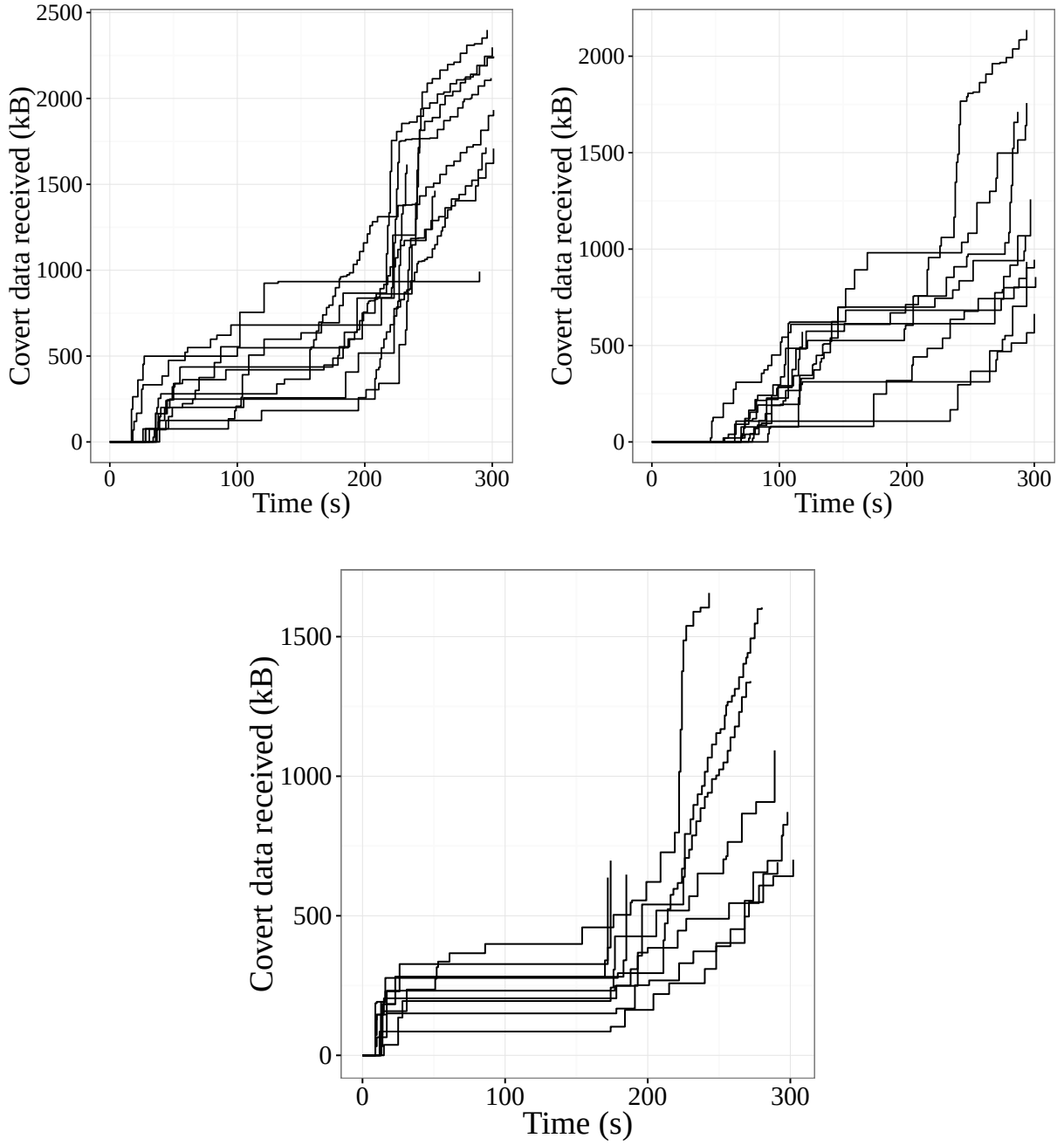


Figure 4.7: Step functions of the time it takes to load a covert site through Slitheen. Each plot shows the covert bytes received over time for ten trials of a different overt video stream. The covert site for all of these tests was a large file download over HTTP in order to test the maximum bandwidth of our system.

YouTube to tunnel traffic through. We accessed our covert site through each overt video stream ten times, for a total of 100 appropriated video streaming sessions. We ran each test for a duration of five minutes. Figure 4.7 shows the time it takes to load the covert site through ten trials of three of the different overt videos. The bursts of content followed by long periods of wait time are caused by buffering behaviour and the pattern of leaf data in the WebM protocol. As video sources are loaded, they contain first metadata information about timestamps and tracks. This is followed by video or audio frames, which we have replaced, and finally seeking information at the end of the resource. The buffering and request of video data by the OUS is reflected in the plots that show the receipt of a large amount of content followed by a lengthy waiting period as the video playback engine waits to buffer more video.

We calculate the bandwidth, or goodput, of our system to be the total number of censorship-resistant bytes received divided by the time it takes to load an entire page. There is an extremely large amount of variance in our bandwidth measurements. This is unsurprising given the plots in Figure 4.7 that show the amount of wait time while the video playback engine of the OUS waits to buffer more content. We calculated the mean covert bandwidth of our 100 page loads to be 30 ($\pm$ 20) Kbps, with a median of 30 Kbps and a maximum reported goodput of 80 Kbps. This far surpasses the 3 Kbps goodput of DeltaShaper, a similar protocol appropriation system that uses VoIP to tunnel censorship-resistant traffic. Our maximum goodput of 80 Kbps is still an order of magnitude less than that provided by meek, a protocol appropriation system that proxies traffic directly from the client to the covert site. However, while meek provides no defences against traffic analysis or website fingerprinting attacks, our system defends against these attacks by design and will continue to be secure in the presence of even more powerful censors.

There is a significant amount of variation between different overt video streams. We note that video streams with more frequent requests for more video data, and therefore shorter buffer times, are better for censorship-resistant traffic that requires multiple network round trips between the user and the covert site. We also note that the wait times shown in these plots are exacerbated by the behaviour of the OUS, which waits until an entire WebM resources has been fully received before processing the resource and sending the censorship-resistant traffic to the SOCKS proxy frontend. While each load of the same video follows more or less a similar pattern when compared to other videos, there is still variation in the processing and buffering times for each trial as shown in the variation between the step functions of the same plot. While dissecting the JavaScript playback engine shipped by YouTube that is responsible for loading their videos is extremely difficult, the differences are likely due to different network conditions at the time of each load.

All plots show large amount of initial buffering time that occurs in the first minute of the video. Without analyzing YouTube's obfuscated JavaScript playback engine, we are unable to verify the cause of this almost two minute wait time near the beginning of the video. Assuming

Table 4.1: A comparison of the overhead induced by existing systems that use protocol appropriation for censorship resistance. We measure overhead as the bytes of total traffic divided by the bytes of censorship resistance traffic.

System	Protocol	Overhead	Median latency (s)	Goodput
meek [Fif17]	HTTPS	N/A (low)	N/A	3 Mbps
SWEET [HZCB17]	SMTP	N/A (low)	5	N/A
Slitheen	HTTPS	5 ($\pm$ 2)	7	30 Kbps
DeltaShaper [BSR17]	VoIP	2	3	3 Kbps

these wait times are consistent and that there exists a "steady-state" of the video stream in which content is retrieved more regularly, we calculate an optimistic goodput of 300 ($\pm$ 200) Kbps with a maximum reported goodput of approximately 1 Mbps. This gives a good indication that if we can predict the long wait times present in video streams, we can achieve bandwidths competitive with those offered by meek.

4.2.4 Comparison to existing systems

We found that, while our system provides a much greater degree of protection against traffic analysis attacks and relies on fewer assumptions about the inability of censors to analyze large amounts of traffic, we incur a reasonable amount of overhead while doing so. While our overhead, latency, and bandwidth measurements can be improved with optimizations at the relay station state machine, we already perform well compared to similar, recently proposed, censorship circumvention systems.

Table 4.1 provides a comparison of our system to recently proposed censorship circumvention systems that use protocol appropriation methods. We note that while we provide competitive latency and bandwidth measurements, our method defends against an adversary capable of performing traffic analysis without making assumptions about the computational restrictions of censors. This means that our system will continue to be secure in the future as machine learning techniques for network classification improve.

We provide a comparison to meek, which offers no traffic analysis protections to censorship resistance traffic as an example of the baseline bandwidth possible with no traffic hiding measures. The *average* goodput we measured was only one order of magnitude less than that offered by meek, while DeltaShaper was two orders of magnitude less. Unfortunately, the nature of video streaming behaviour in the OUS causes the bandwidth to fluctuate wildly and results in

lengthy periods of time during which no covert data is received. This erratic behaviour and flow of covert traffic also has an effect on the latency of our system. We measured a median latency of 7 s, which while only slightly slower than SWEET, is much slower than DeltaShaper. The latency of our system also varied significantly, with a minimum latency of 2 s and a maximum latency of 50 s.

The results of our experiments show that our system is in its current state well suited for censorship-resistant traffic that requires a large amount of downstream bandwidth but minimal back-and-forth communication between the user and the covert site. We perform very well compared to existing systems in loading large covert sites over time, but the latency of any given request is subject to the state of the video playback engine.

4.3 Conclusion

The efficacy of protocol appropriation tools is largely dependent on the available implementations of the protocol, as well as the bandwidth available for censorship-resistant traffic. Usage of these tools spans multiple protocol sessions, and their ability to avoid detection depends not only on the session-specific disguise of plaintext and metadata, as covered in the previous recipe, but also in the simulation of regular user behaviour *across multiple appropriated sessions*. This task becomes much easier if the appropriated sessions are typically high bandwidth and have a long duration. For this reason, the quality of censorship-resistant browsing that users experience while using the tool is tied to the number of necessary appropriated sessions and therefore the level of resistance to censorship that the tool provides. In the example we gave with Slitheen, it is much easier to simulate a regular user if a small number of video streams are needed to tunnel censorship-resistant traffic, than if the OUS must browse a large number of web pages. We leave modifications to the OUS to realistically simulate multiple sessions for future work.

This recipe provides two necessary steps in the implementation of client-side software to improve the experience of the user and avoid detection and blockage by censors. For the first step, avoiding the fingerprinting of the protocol implementation, we showed how we implemented the Slitheen OUS in Firefox 52, a recent version of a popular web browser that is unlikely to be blocked due to collateral damage. For the second step, maximizing the bandwidth of censorship resistance traffic, we applied our first recipe to video streaming in Slitheen web browsing sessions. We measured the overhead, latency, and goodput of our system, and show that it performs well compared to the state of the art in protocol appropriation.

This work is accompanied by a proof-of-concept implementation and test environment for

Slitheen.[8] While we believe that the performance of our system can be improved upon significantly, it already shows promise in an initial comparison to other recently proposed protocol appropriation tools. In addition, Slitheen provides a much greater degree of security, along with a guarantee that even if the network classification abilities of censors continue to improve, Slitheen sessions will remain indistinguishable from the regular use of the appropriated protocol.

[8] https://crysp.uwaterloo.ca/software/slitheen/

Chapter 5

Recipe #3: Deployment on existing infrastructure

As evidenced by the ease with which state censors block access to content they deem contrary to their goals, existing Internet infrastructure is not well suited for Internet freedom. Just as previous recipes described ways to use existing protocols and tools to hide accesses to blocked content, this recipe is concerned with the deployment of said systems using the existing physical infrastructure available to us. Just as in previous recipes, we will see that although existing systems are not perfect for our means, with a little work they may provide tiny cracks in which resistance has a chance to grow.

Deployment is a challenging aspect of creating usable censorship resistance systems. Particularly for protocol appropriation, deployment often requires collaboration with the owners of existing services or infrastructure. The developers of censorship resistance tools often have to fight a battle on two fronts: against the censor to circumvent Internet filtering, and against the owners of Internet infrastructure to allow their systems to run. As we will discuss in Section 5.1, recent events have shown a hostility towards censorship circumvention systems by both sides. Although these systems are often designed to be deployed outside of a censor's area of influence, in a region that might even have political motivations to aid in the resistance of Internet censorship, the political and economic relationships between Internet companies and states with control over the Internet traffic of their region produces tension that has not recently played out in favour of censorship resistance.

We note that, in addition to the possibility of political and economic conflict that cause the benefactors of systems for censorship circumvention to withdraw their support, reliance on states and companies to resist oppression is inherently problematic. Today's centralized Internet infras-

tructure is not owned, run, or controlled by marginalized or oppressed groups. Whatever their claims, the goals and motives (political, economic, or otherwise) of the owners and controllers of Internet infrastructure will never be those of the groups who are resisting *their owners and controllers* who just happen to be in a different geographical location. Though our provided solutions are imperfect, our intention is to provide a way to empower Internet users without making them more vulnerable to other states and companies that already enjoy an overwhelming and imbalanced amount of authority over Internet traffic.

In this chapter, we give a generalized recipe for deploying censorship resistance systems on existing infrastructure. We discuss deployment strategies for censorship resistance systems that use protocol appropriation, their trade-offs, and our work on improving the viability of end-to-middle proxying.[1]

5.1 Options for deployment

The deployment of censorship resistance systems is challenging; the system must remain available for use and be resistant to blocking in the presence of censors that are well equipped to perform reconnaissance and discover the deployment details of the system. The following three steps address the main properties that a censorship resistance system deployment should have:

Step 1: Make the deployment difficult to block

Assuming the censor is able to discover the points of deployment used by a censorship circumvention system, it is crucial to make the task of enumerating and blocking these points difficult, expensive, or even impossible so that the system remains usable within the censor's area of influence and censorship-resistant traffic continues to bypass the censor's firewalls. This difficulty can be caused by a rapidly changing, massive number of deployed endpoints that are difficult even for the active probes of the GFW to enumerate, or the endpoints themselves could cause collateral damage that makes them politically or economically difficult to block.

Step 2: Reduce the impact of and barriers to deployment

All censorship resistance systems share the common feature that the more widespread the deployment, the more successful the system is in resisting censorship or blocking. Deployment is often done by volunteers who stand to gain very little in terms of security, economics, and efficiency from their participation in these systems. The more we can protect our volunteers from

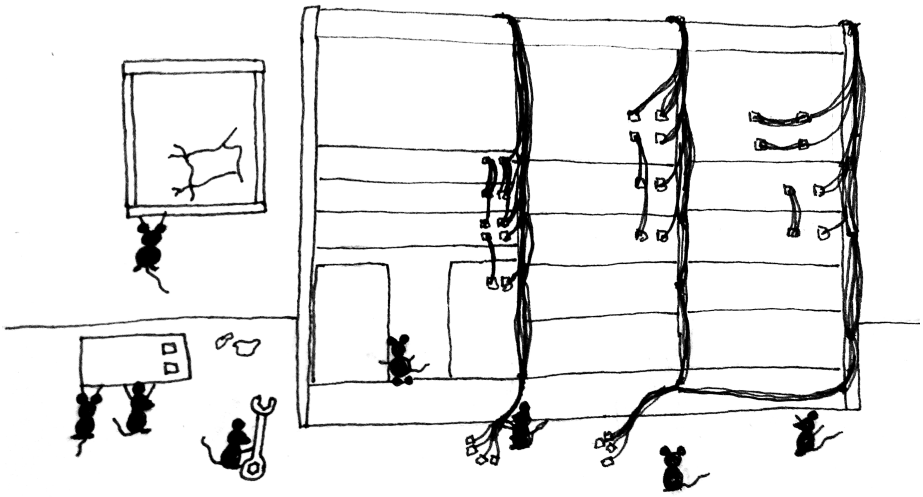
[1] This chapter contains text from our PoPETs 2018 paper [BG18].

attacks by censors and limit our impact on their quality of service, the more volunteers are likely to participate.

Step 3: Protect the identities of the users of the system

In the event that a censor is able to block or monitor traffic to the deployment points of the system, it is crucial to prevent them from then enumerating the users of our system. If, for example, the censorship circumvention system is a connection to a simple proxy which is only used for censorship resistance, a censor could then flag all clients that connect to this proxy as censorship circumventors. This is a property that not many censorship resistance deployments have, but may become much more important as Internet users are increasingly facing harassment by the state based on their mere usage of privacy enhancing technologies [Bow17].

In this section, we discuss three different deployment strategies, each with their own trade-offs, that may all be used with the protocol appropriation recipes in the previous chapters.



5.1.1 Guerilla proxies

The first technique to is to deploy censorship resistance systems at an endpoint yet unknown to the censor. This technique has the disadvantage that, once the endpoints become known to

the censor (and following Kerchkoffs' principle we can assume they eventually will), they will be blocked. However, this assumption underlies the strategy taken by Tor Project in their use of bridges [Tor15b]; the idea is to distribute information about the endpoints (called bridges because of their use to provide a way for users to connect to the Tor network in the instance where other paths to Tor are too difficult to traverse) using a variety of distribution methods to keep their existence hidden from censors as long as possible.

As discussed in Chapter 2, large censors are capable of monitoring traffic and actively probing unknown or suspicious servers to automatically determine their usage for censorship resistance. In response some pluggable transports have implemented defences for active probes by requiring knowledge of a password or key, distributed along with the bridge information to clients [AW14].

While Tor bridges are typically large servers, there have been several attempts at spawning smaller-scale endpoints for censorship resistance. Flash proxies, proposed by Fifield et al. in 2012 [FHE+12], are browser-based proxies run by volunteers that are easy to set up, thereby allowing a large number of Internet users to participate and presenting a moving target to the censor. The large and rapidly changing set of endpoints are designed to be discovered and blocked, but not before they provide a way for a subset of users to circumvent state firewalls. Snowflake [Han11,Fif17] and MassBrowser [NAH18] are more recent and more usable browser-based proxies with similar properties.

Snowflake uses the peer-to-peer WebRTC protocol. When users outside of the censor's area of influence browse a site with a Snowflake deployment, they become temporary proxies that the users from within the censored region can connect to. Although the actual proxying of censorship-resistant traffic occurs between peers, the client must still connect to a central server in order to initially access one of the available proxies. As this server is also prone to blockage, the Snowflake system uses domain fronting, described in the next subsection, to bootstrap the connection.

MassBrowser also takes the approach of combining peer-to-peer connections with a centralized management server, where the centralized server is also domain fronted. In order to further limit the ability of a censor to enumerate all volunteer relays, MassBrowser advertises only a subset of all available relays to each individual user. On the volunteer side, MassBrowser offers an easy-to-use GUI and provides volunteers with fine-grained control over their participation. Volunteers can easily stop using the system, choose what type of traffic to proxy, and control the amount of bandwidth they provide.

The guerilla deployment of censorship resistance endpoints provides a moving target to the censor, and in the case of browser-based proxies are very easy to deploy. However, there is a trade-off between deployability (the ease of spinning up fresh, unblocked endpoints), and the need for bootstrapping servers that are difficult to replace if blocked. As far as barriers to de-

ployment, guerilla deployments aim to make the cost of deployment as low as possible in order to allow many volunteers to run proxies and lessen the impact of discovery and blocking. Finally, guerilla deployments often do not protect the identities of censorship resistance users. The endpoints involved, whether they are Tor bridges or browser-based proxies, are almost always single-use addresses; a connection to the endpoint can almost certainly be classified as a connection to a censorship resistance system. A possible exception to this are browser-based proxies that live behind a large network address translation table (NAT). In this case, the browser shares an IP address with many different machines, each of which may offer a service that is not involved with the censorship resistance system.

5.1.2 Too big to block

A key element of the success of protocol appropriation is the unwillingness of censors to block the protocol, site, or service being used to tunnel censorship-resistant traffic, despite knowledge of their involvement in censorship resistance. This unwillingness to cause **collateral damage**, or the blocking of sites and services that are not used solely for censorship resistance or the original target of censorship, is a commonly seen feature for censors that do not conduct complete Internet blackouts.

Domain fronting is a technique for protocol appropriation that uses the "too big to block" deployment strategy, relying on the edge servers of large service providers or CDNs to redirect censorship-resistant traffic to blocked services such as Tor [FLH+15], or encrypted messaging applications [Mar16].

While it is still recognized that censors strongly prefer to keep as many sites and services available to their populations as possible, we have seen several recent instances where governments have blocked access to a significant portion of the Internet to prevent the usage of privacy enhancing tools. This blocking may have been a commitment to longer-term blocking, and an acknowledgement that there are some instances in which the collateral damage caused is worth it to effectively censor certain content or tools. It is also possible that the blocking was short-term and meant to put political or economic pressure on the companies involved with censorship circumvention.

In April 2018, there were massive blockages of Amazon AWS and Google in Russia, consisting of over two million IP addresses, caused by the Russian government's attempts to block access to Telegram,[2] an encrypted messaging application [Chi18]. Telegram uses domain fronting to evade censorship, with domain-fronted servers existing behind Amazon's edge services. No

[2] https://telegram.org/

more than ten days after experiencing massive outages in Russia, Amazon announced their intention to implement measures to detect and block domain fronting, or the usage of their edge services to mask censorship-resistant traffic [Mac18].

While their press release states that their reasons for doing so are to protect their customers from malware, the timing of the decision suggests it was targeted at systems that use domain fronting to circumvent censorship. This is further supported by the notice sent in the same week to Signal, another encrypted messaging application that uses domain fronting to evade censorship, notifying them of the suspension of their account [Mar18]. Tor has also reported that Google and Amazon have removed support for meek, and that support is expected to soon be withdrawn from Microsoft Azure, the only remaining CDN that works for the pluggable transport [Whi18].

The future of domain fronting is unclear, but it is possible that large services and CDNs such as Google, Amazon, and Microsoft Azure are still too big to block long-term, just unwilling to actively participate in censorship resistance and suffer from short-term outages. In this case, the next step for censorship resistance is to hide from both the censor and the hosting service.

Another trend that has the potential to impact this deployment strategy is the implementation and deployment of their own versions of popular services by large censors such as China and Russia. This reduces the need for their users to issue outgoing connections to services hosted outside of their area of influence.

Despite the withdrawal of support by domain fronting services, censors take a significant economic and political hit in the blockage of large sites and services. Although several countries have blocked Amazon recently in an effort to block access to domain fronted censorship resistance systems, these blockages have been short term and it is unsure whether longer term blocking is feasible for most censors. Although domain fronting is relatively easy for the censorship resistance system to set up (it involves in many cases merely making an account with the service and pointing the system's clients to the right domain), it is expensive to maintain and has a large impact on the service if a censor decides to block them. Finally, domain fronting does protect the users of the system. A connection to a domain fronted service appears in many ways identical to any of the numerous connections to the front service. The identification of censorship resistance traffic then rests on a censor's ability to perform traffic analysis, which we describe how to defend against in the previous two recipes.

5.1.3 End-to-middle proxying

End-to-middle (E2M) proxying is a technique used by decoy routing systems in which the censorship circumvention happens at routers in the middle of the network as opposed to end-

points [WWGH11, KEJ+11, HNCB11, WSH14, EJM+15, BG16, NZH17]. The communication with the censorship resistance system happens covertly through the guise of end-to-end (E2E) connections to endpoints that have not been blocked. The unblocked endpoints, the participant routers, and all routers on the path between the deployment and the unblocked endpoint exist outside of the censor's area of influence, as per our threat model in Section 2.3. Although the majority of decoy routing systems propose an E2M deployment, we distinguish between their use of protocol appropriation to disguise network traffic patterns (described in Chapter 3), and their deployment details (described here) by referring to the former as decoy routing and the latter as E2M proxying. As stated previously, the protocol appropriation methods used by decoy routing systems can be deployed in either an E2E or E2M manner.

In E2M proxying clients communicate with unblocked endpoints in an E2E manner, but steganographically encode messages to routers with deployed relay stations that sit on the path between the user and the unblocked site. The steganographic encoding uses public key cryptography to remain provably invisible to a censor, while allowing the relay station to proxy covert information to and from the client and the covert site. We show the basic architecture of an E2M deployment of Telex, previously described in Section 3.1, in Figure 5.1.

The advantage of E2M proxying is that in order to block access to the censorship resistance system, a censor would have to block not just one service, but all services behind the deployed relay station. Nasr and Houmansadr have shown that with the careful placement of relay stations at autonomous systems surrounding the censor's area of influence, this could result in a decision by the censor to allow censorship resistance traffic or block the majority of the Internet [NH16].

We find it very unlikely that the majority of ASes, being the corporate entities that they are, would be willing to deploy censorship circumvention tools in their infrastructure, but devote a large portion of this chapter to describing ways in which to reduce the burden on Internet service providers should they wish to participate. This technique also faces the same problem as domain fronting: if censors are willing to put pressure on ASes by blocking them for a short period of time, it is likely their response will be similar to those of their CDN counterparts. Due to the nature of deployment, hiding the existence of E2M relay stations from ASes is likely far more difficult than hiding a deployment at an endpoint such as a CDN. The currently proposed deployment requirements of E2M proxying systems would require sneaking into Internet exchange points to install and rewire hardware.

Although E2M decoy routing systems provide strong security properties against both active and passive attacks, there are numerous obstacles to deployment. The deployment of an E2M proxying system relies on the participation of autonomous systems (ASes) that own routers in the middle of the network. Previous work on the optimal placement of E2M proxies aims to maximize the number of unblocked, overt sites available and minimize the required amount of

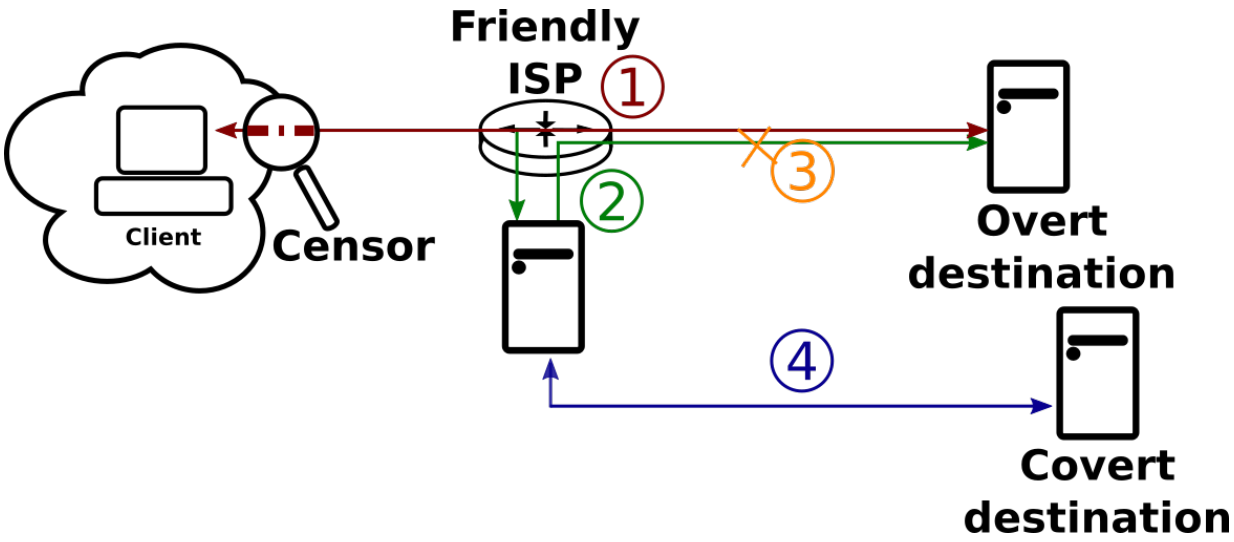


Figure 5.1: An overview of the Telex [WWGH11] E2M deployment architecture. A client first initiates a TLS handshake with the overt destination, tagging the ClientHello message (1). The friendly ISP with a deployed relay station recognizes the tag, and then continues to passively monitor the TLS handshake (2). Upon receipt of the TLS Finished messages from both sides, the station decrypts and verifies the Finished messages with the session's TLS master secret, computed from the client's tag. Finally, the station will sever the connection to the overt site (3), and assume its role as a proxy to the censored, covert site (4). Slitheen is very similar, except that in step (3), the connection between the client and the overt destination is maintained and actively used.

deployed stations [CKSR12, NH16, HWS14]. However, researchers have yet to convince large ASes to deploy E2M proxies in a production setting. While recent work on analyzing a small-scale E2M deployment [FDS+17] provides hope that ISPs are willing to deploy lightweight decoy routers, we are still a long way from convincing the majority of ASes to adopt these systems for Internet freedom purposes. Concerns such as the hardware required to block, modify, or drop traffic at the router, the effect checking for steganographic tags would have on regular traffic, and the logistics involved in setting up and maintaining a relay station remain deterrents for both large and small ASes.

Furthermore, connections to overt sites are often asymmetric. While they may cross a router with a deployed decoy routing relay station on the path to an overt site, the path taken back from the overt site to the user may not cross the same router. This makes the deployment of E2M systems more difficult, perhaps necessitating a larger number of participant ASes. While some asymmetric solutions exist [HNCB11, WSH14, EJM+15], they suffer from security vulnerabilities that could put users already under the scrutiny of a state censor at risk. Waterfall, a recently proposed asymmetric E2M system, requires a relay station only on the downstream half of a flow [NZH17]. This provides resistance against routing around decoys (RAD) attacks [SGTH12]. Furthermore, Waterfall provides significant security improvements to existing asymmetric designs by employing and improving upon the techniques used in Slitheen [BG16] to securely relay covert information in an undetectable and high-bandwidth manner. However, the registration protocol of Waterfall is prone to denial of service attacks and blocking.

E2M proxying is a deployment option that has the potential to be very difficult to block, but is also difficult to deploy. Similar to domain fronting, it also protects the identities of the users of censorship resistance systems. Since the E2E connections are made to unsuspecting endpoints that have not been blocked, and may even be whitelisted by the censor, the identification of censorship resistance traffic relies on traffic analysis techniques.

In the next section, we address the main challenges to deployability that current E2M systems face. We provide an experimental analysis of affordable, off-the-shelf hardware that can be used by ASes in the deployment of decoy routing relay stations. To address the problem of route asymmetry, we leverage the fact that routes between specific clients and overt sites are very stable, meaning they pass through the same set of ASes in subsequent flows. We propose a "gossip" protocol that may be applied to all previously symmetric systems to make them work in an asymmetric setting. In keeping with the ideas presented in Waterfall, our approach emphasizes downstream traffic. We use extremely light-weight upstream stations (simple taps) to relay information to stations on the downstream half of the flow that incur a bandwidth overhead of only $1.0055\times$ the total bandwidth through upstream station. Our design provides a more secure alternative to Waterfall's registration protocol and requires fewer deployed *heavy-weight* relay stations that perform in-line blocking and intense computations than symmetric systems. We

require as few as five heavy-weight stations for a highly connected, routing capable adversary such as China, as opposed to the hundreds of stations required by symmetric designs.

5.2 Known challenges to E2M deployment

Recently, a number of research groups have proposed solutions to the decoy router placement problem (DRP) that aim to maximize the coverage of overt sites available through E2M decoy routing stations and minimize the number of decoy routers needed to successfully inhibit a censor's ability to evade decoy routers and block overt sites [NH16, CKSR12, HWS14]. Sufficiently powerful censors can perform Routing Around Decoys (RAD) attacks by manipulating BGP and routing tables to send traffic to overt sites down paths that do not contain a deployed relay station [SGTH12, NZH17]. With enough deployed stations, these attacks become extremely difficult and expensive [HWS14]. However, we have yet to see deployment on large ASes, let alone the widespread placement of relay stations in the middle of the network.

Wustrow et al. [WSH14] were the first to closely examine deployment challenges, and developed TapDance as the result of discussions with ISPs about their reluctance to deploy existing systems. The resource requirements of relay stations and route asymmetry were cited as the most onerous to ISPs and practical usage of existing systems. Telex and Curveball both require the relay station to perform in-line flow blocking, severing the connection between the user and the overt site after the TLS handshake. This not only requires sophisticated and potentially expensive hardware, it also violates the terms of service many ISPs have with overt sites. Because TapDance does not perform in-line flow blocking, it does not have an impact on the quality of service of HTTPS traffic through the router of a deployed relay station. This has made the trial deployment of TapDance successful, at both a regional ISP and a university network [FDS+17]. During the trial, the deployed TapDance stations were able to serve up to 3,000 clients while processing 40 Gb/s of regular ISP traffic. However, the deployability of TapDance is offset by security vulnerabilities that may lead to easy blocking by a state censor, as we discuss in the next section.

To our knowledge, there have yet to be experiments on the resources needed by a relay station that performs in-line blocking to check steganographic tags and the impact these operations would have on the quality of service for all overt sites accessible through the deployed relay station. Tags need to be checked for every TLS connection, which now comprise over a third of all Internet traffic [Cen16] and require the relay station to perform expensive public key operations. In Section 5.3, we provide an extensive analysis of the impact of checking Slitheen tags using specialized hardware. We chose this tagging procedure as it is used by multiple systems, including Telex, Slitheen, and Rebound.

Another obstacle in the deployment of E2M systems is the prevalence of asymmetric flows. The upstream path from a user to an overt site may pass through a relay station, but the downstream path may take a different route and miss the relay station targeted by the user's tag. Of the seven existing E2M decoy routing systems, only Cirripede, TapDance, Rebound, and Waterfall support asymmetric flows. With these systems, as long as the user's traffic passes through a relay station on the upstream (or downstream, in the case of Waterfall) path to the overt site, the relay station can effectively deliver covert content to the user. However, as we discuss in Section 5.2.1, second-generation asymmetric solutions have significant flaws that could allow a passive censor to identify their usage. While Waterfall has strong security properties, the registration protocol is prone to denial of service and blockage by a censor. Our solution presents an alternative to client registration as well as a solution to the relaying of upstream covert data from the client to the relay station that places less strain on overt sites.

For Telex, Curveball, and Slitheen, the relay station has to see both upstream and downstream traffic of a tagged session. Our solution can be applied to all previously symmetric systems to recognize and use asymmetric flows for the delivery of covert content. We use a gossip protocol for deployed relay stations to share information about potential steganographic tags. We provide a discussion of the deployability features and security properties of existing systems in Section 5.4. We analyze the previously symmetric systems Telex, Curveball, and Slitheen both in their original form and along with our improvements to support routing asymmetry.

5.2.1 Routing asymmetry

Traffic between a client and an overt site often takes a different route, passing through different routers or ASes, in the upstream and downstream directions. Past studies have found somewhere between 80% and 90% of routes to be asymmetric [JDC10, HFKH06, SSW10]. This asymmetry becomes more prevalent in the centre of the network. John et al. [JDC10] found that only about 10% of flows are symmetric in Tier-1 networks (i.e., the backbone of the Internet), while flows at the edge of the network are symmetric about 70% of the time. The ability of a decoy routing system to work in the presence of asymmetric flows enhances the system's deployability by increasing the effectiveness of deployed stations and lowering the number of relay stations that must be deployed to defend against routing-capable adversaries. Each individual relay station can intercept traffic meant for a larger number of overt sites. Four of the existing decoy routing systems accommodate routing asymmetry. Cirripede [HNCB11], TapDance [WSH14], and Rebound [EJM+15] function properly if a user's traffic passes through a deployed relay station only in the upstream direction towards the overt site, but each has security issues or drawbacks, which we outline next. Waterfall [NZH17] takes a different approach, placing relay stations only on the downstream path from the overt site to the user.

Cirripede accomplishes routing asymmetry by handling client registration (i.e., recognizing that a client wishes to begin a decoy routing session) solely through the passive observation of TCP SYN packets. These packets are sent from the client to the overt site at the start of every connection. After recording the ISNs from 12 of the client's TCP connections, they make a rule in their routing table to divert all traffic from the client's IP address to a service proxy for a fixed period of time. During this time, as long as the client's traffic passes through this router in the upstream direction towards any overt site, it will be redirected to a service proxy that will relay data to and from the client and a covert site. Downstream data from the covert site is sent directly from the service proxy to the client, eliminating any need for a relay station to be placed downstream.

On the usability side, a disadvantage of this approach is that all of a client's traffic will be redirected to the service proxy during the fixed time set by the relay station. If a client wishes to browse a site normally, they must wait for the duration of the decoy routing session to end. There is also a security vulnerability due to the fact that traffic between the user and the covert site does not follow the same downstream path it normally would in a connection to the overt site during the proxy phase. If the overt site and the covert site are significantly far apart, a censor could easily notice a significant difference in latency or in where the traffic enters their network to identify decoy routing sessions.

TapDance implements asymmetry by waiting for the client and overt site to complete the TLS handshake before initiating the tagging procedure. The first upstream HTTP GET request from the client contains a tag in the ciphertext that gives the relay station the client's public key and the encrypted TLS master secret for the session. After retrieving the TLS master secret, the relay station can decrypt upstream data from the client and establish a connection to the covert site. It then sends covert data to the client directly, encrypting it with the TLS master secret and assuming the role of the overt server. Unfortunately, the non-blocking nature of TapDance and its inability to block or modify downstream traffic leaves the system vulnerable to active attacks by an adversarial censor. Because the relay station is sending traffic to the client on behalf of the overt site, the TCP sequence numbers for downstream data will differ from the overt site's TCP state. A censor can then replay a stale TCP packet to the overt site, prompting an acknowledgement that reveals the server's true state, inconsistent with what the censor has witnessed. TapDance also suffers from the same passive attack as Cirripede that stems from the difference in the locations of the relay station and the overt site.

Rebound's asymmetric solution presents a different problem by making traffic vulnerable to attack from a passive adversary. Rebound's upstream-only relays receive necessary handshake information from the client in an encoding method similar to TapDance. After reconstructing the TLS master secret, the relay delivers covert content to the user by encrypting it and sending it as an invalid resource name to the overt server in an HTTP GET request. To maintain a consistent

TCP state between the overt server and what a passive censor sees, a client must send a GET request with a length that matches the length of the downstream data she wishes to receive. This results in a nearly equal amount of upstream traffic and downstream traffic, which is a highly atypical traffic pattern for any type of web browsing activity. Furthermore, the ethical implications of sending an unending stream of bad requests to overt sites makes this technique undesirable.

Waterfall places relay stations on the downstream path between the user and overt site, a technique that allows for much stronger security properties in the proxy phase of the decoy routing session as well as a defence against RAD attacks. The downstream-only asymmetry of Waterfall is made possible by the separate registration protocol between the client and the registration server. The client sends a registration package with a series of identifiers, one for each future decoy routing session the client wishes to establish. The identifiers contain all necessary upstream information a relay station would need to man-in-the-middle the TLS session with the overt site. The registration server disseminates this information to relay stations, which then attempt to decrypt TLS sessions whose client IP address are included in the list of registered clients, using the connection identifier information provided. This registration process provides a usability advantage over Cirripede: clients can choose which of their subsequent flows are to be decoy sessions and which are regular browsing sessions. However, an attacker could perform a denial of service attack against suspected clients by registering a series of identifiers in their name. It is unclear how such conflicts in registration would be solved. Furthermore, the connection between the client and the registration server could be censored, requiring the client to adopt a different censorship circumvention system to make this initial connection.

The proxying of covert information to the user in Waterfall is very similar to Slitheen: overt resources are replaced in a manner that perfectly imitates the loading of an overt site. However, upstream information is bounced off of the overt site to the downstream station in a manner similar to Rebound. They suggest several methods for bouncing covert data off of the overt server, including HTTP 404 messages and HTTP Redirects, the latter of which are quite common in normal web-browsing behaviour.

In this section, we describe a solution to achieve asymmetry in previously symmetric decoy routing systems such as Telex, Curveball, or Slitheen, that maintains the security properties of these systems. Our solution can also be used as an alternative to the registration protocol of Waterfall and as an alternate way to relay upstream information to the downstream relay station, and maintains the same RAD-resistance as Waterfall due to the focus on the downstream half of the flow.

We position easily deployable, non-blocking relay stations (which are really just simple taps) in the upstream half of a connection from a user to an overt site to gossip ClientHello random

nonces to possible downstream relay stations that may be able to recognize a tag. As this random nonce is the only upstream part of the TLS handshake a relay station needs to compute the TLS master secret, the downstream station only needs this small amount of gossipped information—and not necessarily in real time—to successfully use that and subsequent flows for decoy routing. During the proxy phase of the decoy routing session, these gossip stations also relay upstream information to nearby downstream stations.

5.2.2 Asymmetric gossip protocol

Our solution for asymmetric decoy routing takes a slightly different approach from existing solutions. We require the existence of a relay station in both the upstream half of the flow (on the path from the client to the overt site), and the downstream half (on the path from the overt site to the client). These relay stations do not need to be placed on the same router, or even in the same AS. The relay station in the upstream path requires only an extremely lightweight non-blocking network tap and aids in both registration and the proxying of upstream information. This tap removes Waterfall's need for a client to connect to a registration server, provides a DoS-resistant way to tag decoy sessions, and reduces the load on the overt site by directly sending upstream covert data to the downstream station.

Our focus on the downstream relay station comes from the fact that a relay station only needs to observe one upstream handshake message to compute the TLS master secret: the ClientHello message that contains the steganographic tag in its random nonce. However, the relay station needs to see multiple downstream handshake messages: the ServerHello, ServerKeyExchange, and (in the case of Slitheen), the downstream Finished message. To minimize the communication between two relay stations on either side of the flow, the upstream relay station "gossips" received ClientHello messages to other known relay stations, in an attempt to reach a relay station on the downstream path.

This approach spans multiple flows between a client and an overt site and therefore requires route *stability*, in which although each flow is routed asymmetrically, the routers traversed in each direction do not vary significantly between the same two endpoints. There is evidence that routes are highly stable; a 2009 study by Schwartz et al. [SSW10] compared the routes taken between between over 10,000 sets of endpoints with an average of about 100 measurements for each pair over the course of four days. Analysis over this time frame is more than sufficient for the purposes of our system. They found that most pairs of endpoints had a dominant route, or one in which over half of the traffic between these pairs traversed. 25% of pairs had absolute stability where all traffic (although possibly asymmetric) always crossed the same routers. Furthermore, the number of distinct routes for endpoints that did experience variance was usually small: only

about 20% of all endpoint pairs had over 20 distinct routes, and very few had over 60.

While the existence of dominant routes indicates high route *prevalence*, this is only one aspect of stability. Prevalence measures how often traffic between two endpoints passes through the same set of routers. Another relevant aspect of stability is route *persistence*, which measures how often a route changes over a given period of time. Routes can have high prevalence but low persistence, in which most connections are sent over the same path but less common paths occur regularly between two endpoints. We desire both high prevalence and high persistence for our asymmetric tagging procedure to work. A 1996 study by Paxson [Pax06] showed that the majority of routes persisted for days to weeks. Additionally, only 10% of routes persisted for less than an hour, which is well within the necessary stability time required for our system.

The measured high persistence and high prevalence of Internet routes leads us to believe that there is a high probability that subsequent flows between the same client and overt site will cross the same downstream relay station, particularly in the short term due to load-balancing practices.

We first describe the tagging phase of our asymmetric solution using the modified TLS handshake used by Slitheen for tagging flows. We follow this with a discussion of asymmetry in the relay, or proxying, phase of the decoy routing session.

Asymmetric tagging

We use the Slitheen tagging protocol as it is a slightly modified, updated version of Telex's tagging procedure. This method varies only slightly from that used by Curveball, in which the client and the relay station share a symmetric secret that was exchanged out of band.

In Slitheen, a steganographic tag is placed in the last 28 bytes of the 32-byte random nonce of the TLS ClientHello handshake message. The first 4 bytes are reserved and usually randomly generated; however, some older implementations of TLS used them as a timestamp. Our solution uses an implementation that randomly computes these bytes, and we will refer to this random value as $\rho = \texttt{ClientHello random}[0..3]$ for ease of reference in the rest of the chapter.

To enable the downstream relay station to compute the TLS master secret from the previous ClientHello random nonce and the server's TLS handshake messages, we make one further small modification to the tag. In Telex and Slitheen, the tag and the client's TLS key exchange parameters are computed from the client-relay shared secret as well as a context string χ, where $\chi = \texttt{server\_ip} \| \rho \| \texttt{TLS\_session\_id}$. In our asymmetric setup, the downstream relay station is responsible for intercepting the flow and may not have access to the TLS session id (as this may or may not be reflected in the ServerHello message, depending on the session's resumption status). Therefore, we use a different context string $\chi = \texttt{server\_ip} \| \rho$ that depends only on

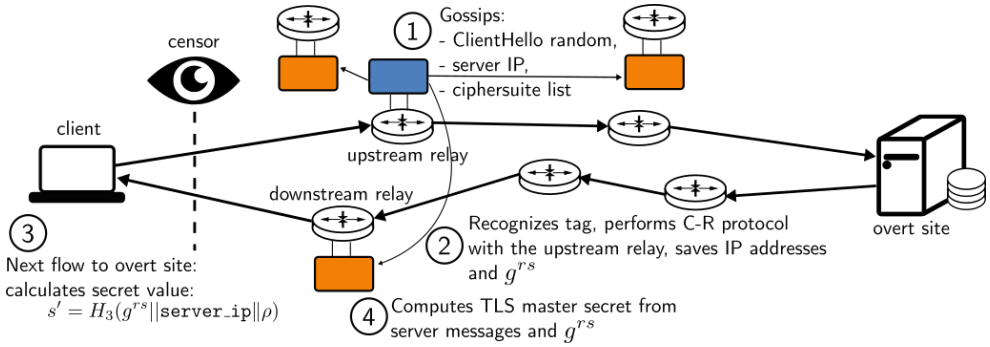


Figure 5.2: Gossip protocol for asymmetric flow tagging. The client tags a connection to the overt site for a relay station positioned in the *downstream* half of the flow. When a relay station sees a ClientHello message in an upstream flow for which it has not seen the downstream SYN|ACK packet, and does not recognize a tag in the random nonce, they gossip this nonce, along with the server IP address and the proposed list of ciphersuites, to nearby relay stations (1). Each of these relay stations checks the gossipped nonce for a tag using its own private key. If it was tagged for them, and they observe the downstream half of the flow, they perform a challenge-response protocol with the upstream station to receive the client IP address, and wait for a future TLS session between the client and the overt site to begin (2). When the client next makes a connection to the same overt site (3), it generates the new client exponent as a hash of the previous client-relay shared secret $s' = H_3(g^{rs}\|\texttt{server\_ip}\|\rho)$. The downstream relay can reconstruct the client exponent and the new ClientHello random nonce themselves. After seeing the server's handshake messages, the relay can compute the TLS master secret (4) and replace downstream content. After the tagging phase, the upstream relay station will continue to send copies of the upstream data to the downstream server; however, this communication is not time critical.

the server's IP address and the first 4 bytes of the ClientHello random nonce. Removing the TLS session ID from the context string will not affect the security of the scheme as an adversarial censor is still unable to perform a tag replay attack. In this attack, a censor would observe a suspect flow and then initiate a TLS connection to the same overt site, reusing the suspect ClientHello random nonce in the hopes of observing their own decoy routing session. Such an attack would require the censor to generate the correct TLS session key matching the tag without the client or the relay private secret, which violates the security of public-key cryptography.

We give an overview of our asymmetric solution in Figure 5.2. Our system uses four different hash functions: H_1, H_2, H_3, and H_4. In our implementation, each of these functions is a SHA256-based PRF with a function-specific constant string prepended to the input as a do-

main separator. A client begins an asymmetric decoy routing session by generating a random secret s and composing the steganographic tag $g^s \| H_1(g^{rs} \| \chi)$ for the ClientHello random nonce using the public key, g^r, of a relay station in the downstream half of the flow. Each relay station has its own public key and these are distributed along with the client-side software to the client. The discovery of overt sites whose paths contain deployed stations can be done by experimentally tagging flows to see if they are successful, guided by a publicly known deployment map of relay stations. The client computes their secret exponent in the key exchange part of the TLS handshake from the client-relay shared secret, g^{rs} by seeding a secure pseudo-random number generator with $H_2(g^{rs} \| \chi)$.

When a relay station in the upstream half of the flow receives a ClientHello message for which it does not recognize the tag, and for which it has not seen the SYN|ACK packet for the flow (indicating routing asymmetry), it gossips the ClientHello random nonce along with the flow's context information (i.e., the server IP address and ρ), as well as ciphersuite information, over encrypted connections to nearby relay stations that could possibly be on the downstream path of the flow. The number of downstream stations deployed for a specific region is likely very small, allowing a gossip station to simply gossip to all nearby deployed stations. If a relay station receives this information and recognizes the tag, it performs a challenge-response protocol with the upstream station to prove that it recognized the tag and receive all further records in that flow. We discuss this challenge-response protocol further in Section 5.2.2. It then saves the client and server IP addresses in a table along with the client-relay shared secret g^{rs}, and waits for future connections from the client to the same overt site. We emphasize that this gossipped message does *not* need to be received by the downstream station before the overt site responds to the ClientHello message. If it is late, it simply acts as a registration step, allowing the downstream station to successfully use the *next* asymmetric connection between the client and the same overt site.

To compute the TLS master secret for a decoy routing session, the relay station needs three values: 1) the premaster secret, computed from the tag in the ClientHello random nonce and the server's public key in the ServerKeyExchange message, 2) the ClientHello random nonce, and 3) the ServerHello random nonce. In the event that the downstream relay station receives the tag and flow information before the overt site has sent the ServerHello message of the TLS handshake, it can proceed to compute the TLS master secret for the current session. If the downstream station has missed the ServerHello message by the time the gossip protocol completes, it waits for the next connection from the client to the same overt site.

The next time a client makes a connection to the same overt site, the client computes the new secret exponent used to construct the steganographic tag as the hash of the previous client-relay shared secret and the IP address of the overt site: $s' = H_3(g^{rs} \| \texttt{server\_ip} \| \rho)$ where the first 4 bytes of the ClientHello random nonce, ρ, are generated from the previous shared

secret g^{rs}. [3] They then place their tag, $g^{s'} \| H_1(g^{rs'} \| \chi)$, in the ClientHello random nonce of the new TLS session along with the deterministically generated first 4 bytes. When a downstream relay station receives the server handshake messages, they extract the ServerHello random nonce, ServerKeyExchange parameters, and compute the client's secret exponent and the ClientHello random nonce from the saved client-relay shared secret, g^{rs}, and the server IP address.

After computing the TLS master secret for the session, the relay station attempts to decrypt the downstream TLS Finished message. If the decryption is successful, it replaces the hash of the Finished message, `finished_hash` with $MAC_{H_4(g^{rs'} \| \chi)}(\texttt{finished\_hash})$. When the client receives the Finished message, they will compute the keyed MAC of the unmodified TLS Finished message and compare the result with the received value. If they received an unmodified Finished message, the flow was not successfully intercepted by a relay station. If they received the keyed MAC, they know the flow has been intercepted and a decoy routing session has begun.

Asymmetric proxying

After the TLS handshake, the downstream relay station begins to proxy information between the client and a covert site. All three symmetric systems rely on upstream data from the client in order to establish a connection to a covert site and relay upstream data from the client to the covert site. We note that in this stage, the amount of upstream data from the client to the covert site is typically far less than the downstream covert data. To retrieve covert data from an upstream relay station, the downstream relay station will perform a challenge-response protocol with the upstream station, proving the session has been tagged for their private key and signalling that they wish to receive TLS application data from the upstream half of the flow. If successful, the upstream station will proceed to funnel upstream TLS records (over a point-to-point encrypted and authenticated connection) to the downstream station, which then decrypts the TLS records and proceeds in the usual manner.

The sending of these upstream TLS records has no time constraints; they can arrive at the downstream station asynchronously with downstream data from the covert site or (in the case of Slitheen) the overt site. Any delay in the receipt of this data will not affect the security or correctness of the system, but only the latency experienced by the client in their browsing of covert content. The downstream station will make a connection to the covert site specified by the client and send the client's upstream covert data through this connection. Telex and Curveball will then deliver downstream covert data directly to the client, while Slitheen will insert it into downstream leaf resources.

[3] The OUS should therefore be a browser whose TLS implementation uses random data instead of a timestamp in that field. Our Slifox OUS, described in Chapters 3 and 4, uses the NSS implementation of TLS, which does have this property.

Challenge-response protocol

We require the downstream relay station to perform a challenge-response protocol with the upstream gossip station in order to receive 1) the client information necessary to recognize future tagged flows, and 2) the upstream TLS records during the proxy phase of the decoy routing session. The reason for this requirement is to mitigate denial-of-service attacks on upstream stations and protect the privacy of both tagged and untagged traffic that passes through each upstream relay station. While the amount of additional data leakage in our gossip protocol is small (all ASes on the path between the client and the overt site have access to the same information), this prevents the usage of our decoy routing system by adversaries in expanding their ability to perform mass surveillance on Internet metadata. While we assume that participant downstream routers in our system can be trusted to not collude with the censor, and will safely allow users to access censorship-resistant content, we do not rule out the scenario in which they wish to collect extra information for surveillance purposes.

First, to prove to the upstream station that they recognize one of the gossipped ClientHello tags, the downstream station uses the gossipped tag, context string information (i.e., the server IP and ρ), and ciphersuite information (i.e., the list of client-proposed cipher suites that the decoy routing system supports as well as valid elliptic curves if applicable) and computes all possible client key exchange parameters for those ciphersuites. Note that in current implementations of decoy routing systems this is at most six different sets of parameters. The downstream station then sends hashes of these key exchange parameters to the upstream station. The upstream station compares these hashes with the hash of the key exchange parameters in the client's key exchange message. If one of them matches, they then send the connection information (i.e., the client IP address) to the downstream station so that they can recognize future tagged flows.

The downstream station must perform the above challenge-response protocol for each subsequent TLS session that the client sends to the overt site in order to receive the upstream TLS records. Because the downstream station can compute the key exchange parameters for future sessions ahead of time, they can send multiple sets of parameter hashes to the upstream station at once. Then, the upstream station can immediately forward upstream records as soon as the client sends a key exchange message whose parameters hash to a matching value.

5.2.3 Resistance to RAD attacks

The placement of decoy routers at ASes is critical for providing censorship resistance to users within censoring regions. Schuchard et al. [SGTH12] were the first to acknowledge that the number of decoy routers necessary to evade censorship in the presence of a routing-capable adversary

is much greater than previous estimates. Since the introduction of RAD attacks, there have been many proposals for the optimal placement of decoy routers [CKSR12, SGTH12, HWS14, NH16]. Although it is unrealistic that all ASes will be willing to deploy our system, these proposals provide an idea for how many decoy routers will need to be deployed to provide censorship resistance for different regions. We draw on the findings of previous work to give an estimate on the number of heavy-weight downstream and light-weight gossip stations needed to resist censorship for China (a highly connected routing-capable adversary).

The placement of downstream decoy routers was investigated by Nasr et al. [NZH17] in their analysis of Waterfall. They found that it is much more difficult and expensive for adversaries to route around downstream stations, and as a result fewer deployments were needed. Only one deployed decoy station impacts almost a quarter of the traffic from Chinese users, while five deployed stations impacts 78% of the routes.

It is much easier for an adversary to route around upstream decoy stations. We used the results from Houmansadr et al. [HWS14] to estimate the number of gossip stations needed. Their results show that if decoy stations are placed at 3% of ASes (outside of China and its ring ASes), 40% of the Internet becomes unreachable for Chinese users, meaning it is not possible for China to avoid all deployed stations without cutting off access to 40% of the Internet. This requires the placement of roughly 880 gossip stations. Table 5.1 gives a comparison of the number of necessary deployments to previous systems. We note that while we require more deployments than both TapDance and Waterfall, our gossip stations are even more deployable than TapDance stations (which have been successfully deployed [FDS+17]): we require no intensive computations in our upstream stations to check for tagged flows.

Our asymmetric solution in this section provides a more secure alternative to previous proposals for the asymmetric deployment of decoy routing systems. Our methods can be easily integrated into Waterfall, providing a more secure alternative to client registration and a method for relaying upstream covert data in a manner that is kinder to overt sites. The tiered deployment made possible by our approach presents a cost-effective way for hesitant ISPs to participate in censorship resistance without the need for hardware that can perform in-line blocking or traffic replacement.

5.2.4 Bandwidth overhead

The bandwidth overhead of the gossip protocol is small compared to the existing load of routers. We also note that the upstream stations do not need to perform in-line blocking, drastically lightening the load compared to previous symmetric systems. The overhead has three parts: (1) that induced by gossipping the ClientHello data that passes through the router to a set of known relay

Table 5.1: Estimates of the number of deployed downstream and upstream stations needed to evade censorship for China, a highly connected, routing-capable adversary. We use results from Houmansadr et al. [HWS14] to estimate a necessary 880 upstream stations to resist RAD attacks and results from Nasr et al. [NZH17] to estimate a necessary 5 downstream stations. While our solution requires more deployed stations than TapDance or Waterfall, it is applicable to all previously symmetric systems and allows for better security properties than its slightly more deployable alternatives.

System	Heavy-weight stations	Light-weight stations
Symmetric designs [WWGH11, HNCB11, KEJ+11, BG16]	880	N/A
TapDance [WSH14]	0	880
Waterfall [NZH17]	5	0
Gossip protocol + any symmetric design	5	880

stations, (2) the challenge-response protocols between the upstream and downstream stations, and (3) that of funnelling the upstream TLS application records of proven tagged flows to the downstream station. The gossipped data consists of the ClientHello random nonce, the server IP address, and the list of supported ciphersuites and supported elliptic curves. Its size is dependent on the number of ciphersuites supported by the client. Using Firefox, we measured the average gossip data size to the Alexa top 100 sites as 66 bytes. Note that there was almost no variation in the ciphersuites offered by the client in the version of Firefox we were using. Using traffic measurements from the Center for Applied Internet Data Analysis (CAIDA) [Cen16] shown in Table 3.1 in Section 3.3.6, we calculate the bandwidth overhead of gossipping ClientHello messages as

$$1 + \frac{\text{gossip bytes} \cdot \text{Average HTTPS flows/s} \cdot n}{\text{Bytes per Mb} \cdot \text{Average Mb/s}}$$

$$1 + \frac{66 \cdot 4430n}{125000 \cdot 2035.71} = 1 + 0.0011n$$

where n is the number of relay stations gossipped to. If, for example, we set $n = 5$, the number of downstream routers sufficient to defend against a highly connected adversary such as China, the overhead is only $1.0055\times$ the total bandwidth through the router of the deployed relay station. To give concrete numbers, for a router on a typical OC48 link of a large ISP that handles approximately 2 Gb/s of traffic, the router would have to transmit an extra 11 Mb/s of gossip data.

The challenge-response protocol requires the downstream station to send the upstream station a maximum of six 32-byte hashes for each TLS session, given current implementations of decoy

routing systems. The upstream station responds with a 4-byte client IP address. The total amount of data exchanged for a single-session asymmetric decoy routing session is then 196 bytes. As the base rate of decoy routing flows is very low, this number is negligible in the calculation of the overhead.

To calculate the bandwidth of the proxy phase of the gossip protocol, we measured the average bandwidth of upstream TLS application data to the Alexa top 100 sites. Note that this data is only gossipped for flows that are tagged for a downstream station, which do not likely make up the majority of traffic through a relay station. In our CAIDA data set, the proportion of all data that is upstream data in TLS flows is 0.042. The overall bandwidth overhead is then $1 + 0.0011n + 0.042\beta$ where β is the base rate of tagged TLS flows. Note that for $\beta < 10^{-3}$, the overhead induced by copying upstream data is negligible, resulting in a total overhead of only a few percent.

To compare the bandwidth cost of relaying upstream information to the downstream relay during the proxy phase of the session to Waterfall, which has a similar requirement, our approach requires strictly less additional traffic. Our approach sends upstream records directly, while Waterfall requires them to be wrapped in appropriately sized HTTP GET requests. It is important to note that while our approach requires less traffic overhead, it does require more effort from the system to determine which bytes to forward, perform the challenge-response protocol, and tunnel upstream traffic, though the demands we place on the upstream station are less than those required by TapDance, which is already shown to be deployable [FDS+17]. Importantly, unlike Waterfall, our approach places zero additional load on unsuspecting overt sites.

5.3 Relay station experiments

Wustrow et al. [WSH14] found the main obstacle in convincing ISPs and ASes to deploy decoy routing systems to be the resource requirements of existing systems in checking tags and performing in-line blocking. By checking every TLS session for steganographic tags, the deployment of decoy routing systems also has the potential to affect the quality of service for all customers whose traffic traverses a relay station.

We performed several experiments to determine the impact a deployed decoy routing station would have on existing traffic in a real-world scenario. Note that these experiments measure the cost of the heavyweight *downstream* relay stations, of which fewer need to be deployed to defend against routing-capable adversaries. The cost of an upstream gossip station in terms of its impact on quality of service is non-existent as the station does not perform in-line blocking of flows.

Our first set of tests aims to measure the effect tag checking would have on the quality of service for both TLS and non-TLS traffic of regular customers. The Slitheen tagging procedure is the most suitable for these measurements as a worst-case scenario for all proposed decoy routing systems, as the Slitheen modified TLS handshake requires the most effort from a deployed relay station.

For our tests, we used specialized (but off-the shelf) hardware capable of performing in-line blocking and efficient deep-packet inspection. Our reasons for doing so were that 1) only TapDance does not require in-line blocking, and this feature also introduces several vulnerabilities that an active attacker can exploit to easily differentiate decoy routing sessions, and 2) by showing the capabilities of existing hardware to efficiently perform decoy routing tasks we can target existing users of this hardware as the first to deploy a decoy routing system.

The relay station itself consists of two parts: a Sandvine Policy Traffic Switch (PTS) 22600 capable of performing deep-packet inspection and flow diversion, and a relay station server with two 10 Gb/s connections to the PTS. If a tagged flow is detected by the PTS, it is diverted to the relay station server. The relay station server and client machine each used 8 cores and 2 GB of RAM. The PTS is responsible for routing all traffic and checking the tags of TLS flows. If a flow is tagged for the relay station's public key, the PTS then diverts the flow through the relay station server, which performs the rest of the tagging procedure during the TLS handshake and handles the proxy phase of the decoy routing session. We provide an overview of our experimental setup in Figure 5.3.

As in Chapter 3, we rain a CAIDA-representative distribution of traffic through the relay station to simulate realistic network conditions.

5.3.1 Impact on quality of service

Deployed relay stations must check every incoming TLS ClientHello random nonce for a steganographic tag. This requires a public key operation to compute the client-relay shared secret from the first 21 bytes of the nonce, and then a hash of the shared secret with a context string. This hash is then compared to the last 7 bytes of the 28-byte random nonce. These operations take time, and we sought to measure the latency they add to TLS flows.

To do so, we made 1000 sequential, untagged TLS handshakes to the Alexa top 1000 TLS sites and measured the time between when the client sent the ClientHello message to the time it took for the client to receive the TCP acknowledgement of the message. We tested two conditions: one where the PTS checked ClientHello messages for tags, and one in which the PTS did not check for tags. For each of these tests, we ran a CAIDA-representative amount of background traffic through the relay station, as described above. We present the results in Figure 5.4.

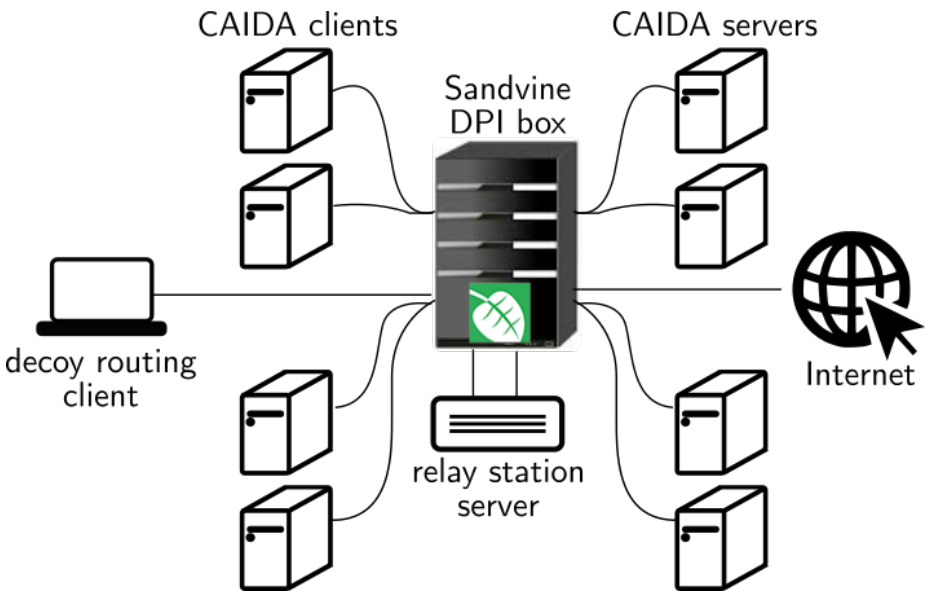


Figure 5.3: The network topology of our experiments. Machines designated as CAIDA clients and CAIDA servers were dedicated to sending background traffic through the DPI, representative of traffic sent through an OC48 link of a large ISP according to statistics gathered from CAIDA [Cen16]. [Repeated from Figure 3.8 for convenience.]

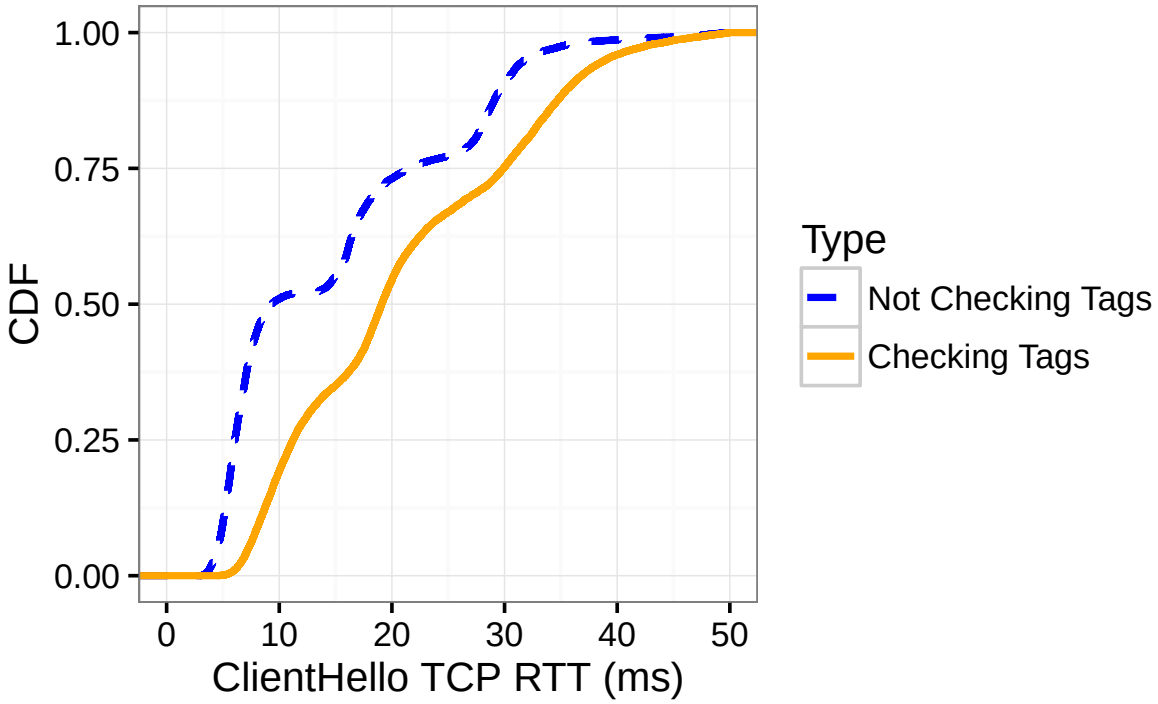


Figure 5.4: The impact of Slitheen tag checking on the latency of TLS traffic. We give cumulative distribution functions comparing the TCP round trip time (RTT) for TLS ClientHello packets with tag checking on and off.

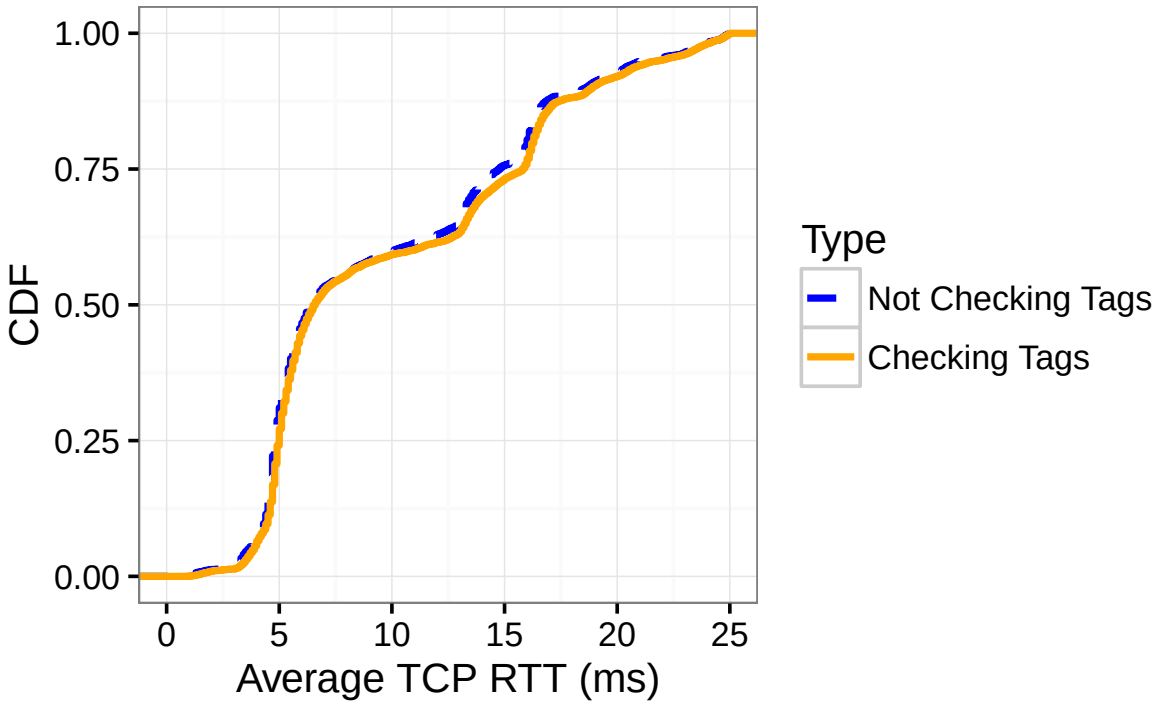


Figure 5.5: Cumulative distribution functions comparing the average TCP round trip time (RTT) for non-TLS flows with tag checking on and off.

Although deploying a relay station has an impact on the latency of ClientHello messages, the average additional latency is 7 ms, which is very low and falls within the standard deviation of each condition (10 ms).

In addition to measuring the impact a deployed relay station has on the quality of service for TLS flows, we also measured the impact it has on non-TLS flows and whether our equipment and software introduced any additional latency by performing deep-packet inspection to search for ClientHello messages. We performed a similar test as above, this time making 1000 HTTP connections to remote sites for each condition. For each connection, we calculated the average RTT of all TCP packets in the flow. The results are given as CDFs in Figure 5.5. The additional latency of deploying a relay station was 0.4 ms, which is very low, and falls within a standard deviation of each condition (10 ms). We note that at this time, the Slitheen tag checking and relay station code has not been optimized for quality of service. With further improvements, the results in this section for both TLS and non-TLS flows will likely show an even lower impact on the customers of participant ISPs.

Our results show that while the deployment of a full downstream relay station adds additional latency to flows due to checking for tags in ClientHello random nonces, the latency introduced is quite small.

5.4 Security analysis and improvements

Our proposal to add asymmetry to previously symmetric decoy routing systems has two main advantages: it has better security properties than previously proposed asymmetric systems, and it provides a path for tiered deployment, creating a less expensive defence against routing-capable adversaries.

TapDance remains the only system capable of performing decoy routing without requiring a relay station to block or modify traffic. However, this feature comes at the cost of security. We believe a better route to deployment is by providing ISPs and ASes with experimental evidence of the impact a deployed relay station would have on customer traffic using existing hardware capable of performing tag checks efficiently and blocking or modifying tagged flows. By targeting ASes that already own this hardware or showing them a clear path to deployment, we are providing more evidence that decoy routing is an attainable option and moving towards real-world deployment. Furthermore, our asymmetric solution does not require in-line blocking for upstream relays, enabling more cautious potential participants to provide a stronger defence against RAD attacks.

5.4.1 Security analysis of Slitheen

In addition to successfully disguising traffic patterns, Slitheen also defends against previously proposed attacks active attacks on decoy routing systems [WWGH11, WSH14, SGTH12]. These attacks consider two different types of adversaries: a *network adversary* capable of both monitoring and modifying traffic by dropping, injecting, or changing packets inside their area of influence, and a *routing-capable adversary*, as defined by Schuchard et al. [SGTH12] who is able to not only change traffic, but also make routing decisions on traffic that leaves their network.

The goal of the adversary is ultimately to identify a decoy routing session. An adversary may also try to identify the censored content that the client is accessing through the decoy routing session. We do not consider attacks that allow the adversarial censor to perform unrealistic computations or utilize unrealistic amounts of resources; for example, we assume that the adversary is unable to distinguish a tagged ClientHello message from a truly random nonce, as doing so would violate a cryptographic assumption. Similarly, we assume that the adversary may not

compromise the TLS session between the client and the overt site by brute-forcing the overt site's private key, or performing a TLS downgrade attack. Furthermore, deployed relay stations and overt sites are assumed to be geographically outside the censor's sphere of influence.

Network Attacks

An active adversary is capable of modifying, injecting, or dropping traffic in addition to passive monitoring. The following attacks are known active attacks against previous decoy routing systems.

Tag Replay Attack. Our system inherits protection against a tag or handshake replay attack from the Telex handshake procedure. If an adversary attempts to replay a tag, they will not be able to successfully construct the TLS Finished message without knowledge of the shared secret, resulting in a connection terminated by the client and overt destination.

State-Controlled Root Certificates. In deployments where censors are actively performing man-in-the-middle attacks on TLS traffic by mandating the installation of a state-controlled root certificate, the resultant flows will not properly proceed through the tagging procedure and will pass through the station unaltered. While this performs a denial of service attack on Slitheen, it does not reveal a client's usage of the system unless they include X-Slitheen headers in their upstream requests. To alert the client of the fact that their decoy routing session has not been safely established, we make a slight modification to the TLS handshake in which the Slitheen station modifies the Finished message hash by replacing it with

$$MAC_{H_4(g^{rs}\|\chi)}(\text{Finished}),$$

a MAC of the original Finished message. The MAC key is derived from the client-relay shared secret g^{rs} and contextual information about the appropriated session χ, as described in Section 3.3.1. This modification occurs after the station has verified that the Finished messages are correct. In the event that the Finished message from the server is not correct, the relay station will forward the unmodified, incorrect Finished message to the client in order to allow the client's TLS implementation to sever the TLS session as it normally would.

When the client receives the Finished message, they will verify it using the additional seeded input to determine whether the decoy session has been established. If it has, the client proceeds to include X-Slitheen headers with upstream data. If it has not, the client will verify the Finished message the traditional way and continue with a regular (non-decoy) fetch of the page. A man-in-the-middle capable of viewing the plaintext will therefore detect no unusual behaviour from

the client. This modification also serves to notify the client of a failure in the tagging procedure due to route asymmetry.

Server Collusion. In previous systems, the censor could collude with or set up an overt destination server to entrap clients that use that server for decoy routing purposes. This attack is outside of our threat model, but we still consider the power such an attacker might have. In Slitheen, the client's behaviour from the overt site's perspective will be identical to regular use with the exception of an X-Ignore header containing garbage bytes. If the existence of the X-Ignore header is a concern, the relay station can instead replace it with a common but mostly unused header. However, a censor that monitors information leaving the overt destination can compare ciphertexts to detect content replacement. In fact, no existing decoy routing system can completely defend against an adversary that has a complete view of packets entering and leaving both the client and the overt site. Our system increases the work of the adversary from previous systems by requiring the colluding parties to compare ciphertexts as opposed to metadata.

Routing Attacks

Routing-capable attackers were introduced by Schuchard et al. [SGTH12] and describe censors with the ability to route packets through either a tainted path (i.e., one on which a Slitheen station resides between the client and the overt destination), or a clean path (i.e., a path with no Slitheen station between the client and the overt site). While a censor may not always be able to find a clean path to the overt destination, our system defends against an adversary that does have this ability. We also note that, as in Telex, the location of relay stations can be public knowledge, and therefore routing-capable attacks to determine whether a network path contains a relay station (as opposed to whether a *particular flow is using* a relay station) do not affect the security of our system.

TCP Replay Attack. In a TCP replay attack, the censor attempts to identify the use of decoy routing by testing whether the client has a TCP connection with the overt site. The censor can replay a TCP packet sent by the client on a clean path. In TapDance and first-generation decoy routing systems, the connection between the client and the overt site has been severed or abandoned and the overt site will issue a TCP RST packet or a stale TCP sequence number, signaling to the censor the usage of decoy routing. Note that in TapDance, the adversary does not need to find a clean path, but can inject a TCP packet into the stream. Since the TapDance station does not perform in-line blocking, the packet will be forwarded to the overt destination despite the fact that it traverses a tainted path.

Our system maintains a TCP connection to the overt destination, providing a defence against this type of replay attack. Since our replacements match the sizes of requests and responses exactly, the TCP state between the client and the overt site as seen by the censor is the true TCP state. Furthermore, Slitheen eliminates the ability of the censor to identify its use through TCP/IP protocol fingerprinting. The station modifies only application-level data, which is unidentifiable by the censor as ciphertext. We do not need to mimic the server's TCP options, or IP TLS values as these are supplied by the overt site itself.

Crazy Ivan. The Crazy Ivan attack involves a censor with the ability to control the path a client's packets take to their destination to detect the usage of, or deny availability to, decoy routing. The censor allows a client to connect to the overt site through a tainted path, and waits until the TLS session has been established to redirect the flow down a clean path.

In previous systems, this attack gives the censor overwhelming evidence of decoy routing. Systems such as Telex, Cirripede, Curveball, and TapDance that sever or abandon the connection between the client and the overt destination will be unable to block packets sent down the new clean path, resulting in TCP RST packets from the overt site. By keeping the connection between the client and the overt site active, both Slitheen and Rebound offer a defence against this type of detection attack. Packets sent down a clean path will be received by the overt destination in the usual manner, prompting the server to send the requested resource (in Slitheen) or an HTTP error message with the invalid request (in Rebound). The TCP sequence and acknowledgement numbers will match those that the censor expects. The censor, unable to decrypt these packets, will see no difference in the traffic. In Slitheen, as long as the client is compressing only the unnecessary headers of HTTP GET requests in the upstream half of the flow, the server will respond as usual and produce no identifying differences in behaviour.

Forced Clean Paths. An adversary with the ability to chose between clean and tainted paths may route around a Slitheen station altogether. This would prevent the client from ever coming into contact with a participating ISP. Although the consequences of this attack would result in a complete loss of availability to the decoy system, Houmansadr et al. [HWS14] show that this attack is too expensive for realistic censors, and very unlikely. We also note that this attack is only a denial of service, and will not leak information about whether the client is using or has used Slitheen.

5.4.2 Security of the gossip protocol

While the gossip protocol does not leak any additional information of tagged or untagged flows to an adversary, and it is encrypted between the upstream and downstream stations, it does increase the number of routers that see traffic between the client and the overt site, possibly increasing the ability of a passive adversary to perform traffic analysis or surveillance attacks. However, gossipped messages do not significantly increase a censor's ability to detect the usage of censorship circumvention tools or attribute them to individual users. In this section, we discuss the impact of the gossip protocol on the security and privacy of both users of Slitheen and non-users whose upstream handshake messages are gossipped to other relay stations.

Passive adversary: The gossip protocol requires an upstream station to send the ClientHello random nonces of all seemingly untagged ClientHello messages, along with the upstream TLS application data to the downstream relay station. Note that the gossipping of ClientHello information is done for all TLS handshakes that the upstream station does *not* recognize as tagged and includes both untagged and potentially tagged flows; therefore the gossip messages do not expose censorship resistance traffic.

It is worth noting that the gossipping of application data to the downstream relay station only happens for tagged flows. A censor that can see traffic between relay stations could then perform a timing analysis attack to connect outgoing connections to gossipped messages. This is outside our threat model as we assume relay stations exist outside of the censor's area of influence and therefore probably do not cross through a censor's control. It is also practically difficult to correlate the TLS application records of any one flow to the encrypted traffic sent between two relay stations. Approximately 37% of flows are HTTPS [Cen16], meaning that a censor observing traffic on even a small router would have to decide which of the thousands $(1/\beta)$ of TLS sessions that data corresponded to.

Malicious relays: Our challenge-response protocol in Section 5.2.2 prevents a malicious downstream relay from lying about recognizing tags in order to induce extra load on the gossip station in a denial-of-service attack, or to increase their surveillance of flows outside of their usual field of view. However, precautions should be taken to prevent a censor from pretending to be an upstream station in order to use downstream stations as oracles to identify tagged flows. Downstream stations should maintain a list of approved and trusted upstream stations, as well as their public key information. This information can be updated by relay station operators as new upstream stations are deployed in much the same way as the client software maintains a list of public keys for trusted downstream stations.

5.5 Comparison to existing systems

We gave a general outline of the challenges different deployment methods face with respect to our general recipe for the deployment of censorship resistance systems in Section 5.1. In this section we look at the desired properties of censorship resistance deployments in more detail and compare our enhanced version of E2M proxying to existing systems.

The main challenge faced by E2M decoy routing systems was Step 2 of our recipe: reducing the impact of and barriers to deployment. This is exacerbated by the apparent trade-off between enhanced deployability at routers in realistic network conditions, and the ability of the system to resist detection and blocking (which are covered in Steps 1 and 3 of our recipe). Recently proposed asymmetric systems that boast less of an impact on participating ASes are easily subjected to either active or passive attacks that results in an identification of users and the blockage of censorship-resistant connections.

Symmetric decoy routing systems that require in-line blocking at the deployed relay station offer better security properties and satisfy Steps 1 and 3 of our recipe. The solution we proposed in the previous sections can be applied to all previously symmetric systems to recognize and use asymmetric flows for the delivery of covert content in way that maintains the security properties of symmetric systems. Furthermore, requiring in-line blocking only on the downstream half of the flow reduces the impact and barriers of adoption for volunteer ASes.

We provide an overview of the deployability features and security properties of existing E2M decoy routing systems in Table 5.2. The previously symmetric systems Telex, Curveball, and Slitheen are analyzed both in their original form and along with our improvements to support routing asymmetry.

While we have improved the deployability of E2M decoy routing systems, E2M proxying still carries a high deployment cost when compared to the other deployment options we have presented. Table 5.3 shows our estimation of the relative blockage resistance, deployability, and user security properties of censorship resistance systems, and therefore their adherence to our deployment recipe. We give relative values for the blockage resistance, deployability, and user protections of censorship resistance systems grouped by their deployment strategy. We consider each of the systems discussed at the beginning of the chapter, with all symmetric decoy routing systems grouped together. In this table, Slitheen includes the system we presented in previous chapters with the enhancements discussed in this one. Each of the deployment properties is assigned relatively to be very low, low, medium, high, or very high. The blockage resistance values for many systems are unknown, due to a lack of deployment and therefore data on blockage attempts.

Systems that are deployed through guerilla proxies all exhibit high deployability, but lower

Table 5.2: A comparison of the deployability features and security properties of existing systems. We indicate that a system has the property or feature listed on the left of the table with a filled circle ●. Systems that lack a feature or property are marked with an empty circle ○. Our proposed design to support asymmetric routes enables the deployment of lightweight upstream stations with no in-line blocking in addition to the original heavyweight downstream stations. (We denote the requirement for in-line blocking in only the downstream stations with the half-filled circle ◐.) This improvement to deployability provides the potential to thwart RAD attacks.

	Telex [WWGH11]	Telex + this work	Cirripede [HNCB11]	Curveball [KEJ+11]	Curveball+this work	TapDance [WSH14]	Rebound [EJM+15]	Slitheen [BG16]	Slitheen + this work	Waterfall [NZH17]	Waterfall + this work
No in-line blocking	○	◐	○	○	◐	●	○	○	◐	◐	◐
Asymmetric	○	●	●	○	●	●	●	○	●	●	●
Defends against TCP replay attacks	●	●	●	●	●	○	●	●	●	●	●
Defends against latency analysis	○	○	○	○	○	○	●	●	●	●	●
Defends against website fingerprinting	○	○	○	○	○	○	○	●	●	●	●
RAD-resistant	○	●	○	○	●	●	○	○	●	●	●
DoS-resistant registration	●	●	●	●	●	●	●	●	●	○	●

Table 5.3: A comparison of different deployment techniques. We look at the blockage resistance, deployability, and user protections of censorship resistance systems, grouped by their deployment strategy. Values for each of these properties are assigned relatively to be very low, low, medium, high, or very high. Unknown values due to a lack of deployment or data are represented with a ?. Guerilla proxies all exhibit high deployability, but lower user protections. The "too big to block" strategy has high user protections, but has been shown to be prone to blockages. E2M proxying is very difficult to deploy, but provides high user protections. The blockage resistance of E2M proxying is still unknown, but has the potential to be high given widespread deployment.

Strategy	System	Blockage resistance	Deployability	User protections
Guerilla proxies	Tor bridges	high	high	low
	Flash proxy	?	high	medium
	Snowflake	?	very high	medium
	MassBrowser	?	very high	medium
Too big to block	meek	medium	medium	very high
E2M proxying	Symmetric DR	?	low	very high
	TapDance	low	medium	medium
	Slitheen	?	medium	very high

user protections. The "too big to block" strategy has high user protections, but has been shown to be prone to blockages. E2M proxying is very difficult to deploy, but provides high user protections. The blockage resistance of E2M proxying is still unknown, but has the potential to be high given widespread deployment.

5.6 Conclusion

In this chapter, we provided a recipe for the deployment of censorship resistance systems. This recipe describes three main properties that all censorship resistance deployments should strive for: 1) resistance to blocking despite a censor's knowledge of the system and deployment strategy, 2) a low barrier to participation, and 3) protections against the discovery of users, even in the event that a censor is able to find and block deployment points.

E2M proxying provides a promising solution to Internet censorship, contingent on the participation of ASes. Its strong security properties, and compatability with protocol appropriation techniques have the potential to end the cat-and-mouse game in favour of the resistor. E2M

systems by virtue of their design provide protections to individual users as their connections to the system appear identical to connections to whitelisted IP addresses. However, before E2M proxying can be deployed and used by people in censored regions of the world, we must first address the obstacles to deployment that have prevented the owners of network infrastructure from participating in existing systems.

We proposed a new approach to routing asymmetry that provides better security than previous asymmetric systems and a path to tiered deployment that allows for several lightweight, limited systems to surround a powerful censor, limiting the censor's ability to perform routing-based attacks. We also investigated the use of existing hardware in the implementation of decoy routing systems. We experimentally tested the impact decoy router deployment would have on the quality of service for traffic flowing through a participant router and more carefully examine the latency of decoy routing sessions processed by the relay station. Finally, we identified a possible security vulnerability in existing systems and proposed a cryptographic solution that doubles as a means to more reliably deliver covert content to the user.

This work presents the next steps towards the deployment of decoy routing systems, however there is still much work to be done. With more efficient implementations of the relay station, the possible impact of deployment may be even less than what we measured with our limited improvements. We look at our results as a positive indication that decoy routing may prove to be practical in the future and may convince the owners of Internet routers to consider participating in censorship circumvention.

Chapter 6

Conclusion

Our recipes for resistance cover three main aspects of the development of censorship circumvention systems:

1. how to hide the traffic patterns of censorship circumvention tools such that the ever more powerful censors of tomorrow will be unable to identify their use amidst traffic that complies with their censorship policies,

2. how to create high-bandwidth and low-latency channels for covert data that are comparable to the performance of existing circumvention systems while providing stronger security properties, and

3. how to deploy censorship circumvention tools on the infrastructure available to us.

Despite our less-than-ideal current situation with the state of Internet censorship and increasingly bold Internet blockages by state-level censors, these recipes for usable and invisible censorship-resistant communication provide a way to hack self-determination and empowerment into a system that is increasingly manipulated by state and corporate entities.

In Chapter 3, by removing all visible traffic patterns from censorship resistance systems through the careful appropriation of popular protocols, we provide a way to end the cat-and-mouse relationship between censors and the designers of censorship resistance systems. No matter how much more powerful censors become and how much better their systems become at analyzing network traffic patterns, our tools can be indistinguishable from the regular use of these protocols.

In Chapter 4, we discuss how to simulate compliance and make secure censorship circumvention systems usable by increasing the bandwidth of tunnelled censorship-resistant traffic. Our high-bandwidth methods allows users to gain access to the content they want while limiting the amount of censorship resistance traffic needed and thereby protecting users from targeted attacks.

In Chapter 5, we provide a recipe for the successful deployment of censorship resistance systems that results in three desirable properties for their deployment: resistance to blocking, low barriers for participation, and security protections for individual users. We describe the trade-offs between different deployment strategies and how to increase the deployability of E2M deployments of Slitheen.

6.1 Defence of the thesis statement

In our thesis statement, we pointed to a trend in which many censorship circumvention systems cease to be useful as the traffic analysis and detection abilities of censors grow. This trend is backed up by a history of blocked Internet freedom tools and recent events that have eliminated the usefulness of popular circumvention strategies.

We stated that, while many systems fall victim to the cat-and-mouse game between censors and censorship resistors, it is possible to design and deploy systems that "stand the test of time... despite technological improvements that enhance the traffic analysis abilities of the censor". Our first recipe provides a generalized method that may be applied to a variety of different protocols, for making censorship resistance traffic identical to non-resistance traffic. Systems built following this recipe stand the test of time against censors that employ traffic analysis techniques by making censorship resistance traffic indistinguishable from non-resistance traffic to anyone without the secret key, which is held only by the trusted man-in-the-middle outside of the censor's area of influence. We enumerate all traffic pattern features visible to a censor: plaintext protocol messages and information, and metadata such as packet sizes, latency, and linkable Internet connections from a single user. By replacing only encrypted leaf data with encrypted censorship-resistant traffic on a packet-by-packet basis and simulating multiple connections by the user, we hide all features other than latency, which we experimentally determine to be statistically impossible for a censor to detect. No improvements in machine learning, traffic analysis, or website fingerprinting techniques will allow a censor to identify our traffic among non-resistance connections.

In our second recipe, we show how to make these systems usable and defend against the identification of individual users in specific implementations of our recipe and across multiple appropriated sessions by maximizing the goodput of censorship-resistant traffic. We accompany

these recipes with a scientific analysis of Slitheen, our proposed decoy routing system for appropriating secure web browsing traffic, and demonstrate that the implementation and use of such a system is possible. In our evaluation, we report measured bandwidths comparable to and exceeding those of existing systems. In our recipes and discussions of Slitheen, we have demonstrated that it is possible to design secure censorship resistance systems that will continue to remain useful despite advances in censorship infrastructure.

The remaining part of our thesis statement, that these systems remain undetectable "despite open knowledge of their operation and use" is addressed in all three recipes. Particularly, in our third recipe, we discuss the deployment of these systems and compare three different deployment strategies. These deployment strategies are designed to be successful despite open knowledge of the details of their deployment and the ability of censors to discover deployment points through reconnaissance. On a traffic analysis level, our protocol appropriation methods in our first and second recipes are also secure in the event that a censor has complete knowledge of how the system works. To accompany our third recipe, we present the results of our work to lower the barriers of deployment for one such strategy, end-to-middle proxying, and evaluate these improvements with our implementation of Slitheen. In support of our thesis statement this shows that we can both "design *and deploy*" strong and effective censorship resistance systems.

Through our generalizable recipes and our in-depth analysis and discussion of Slitheen, we have provided clear steps and evidence for the existence of censorship circumvention systems that will continue to remain secure and unidentifiable, even in the face of censors with superior traffic analysis and reconnaissance abilities to those seen today. We have removed the assumptions of limitations on censors' technological abilities and instead explore solutions that are by design indistinguishable from regular, non-circumvention traffic.

6.2 Future work

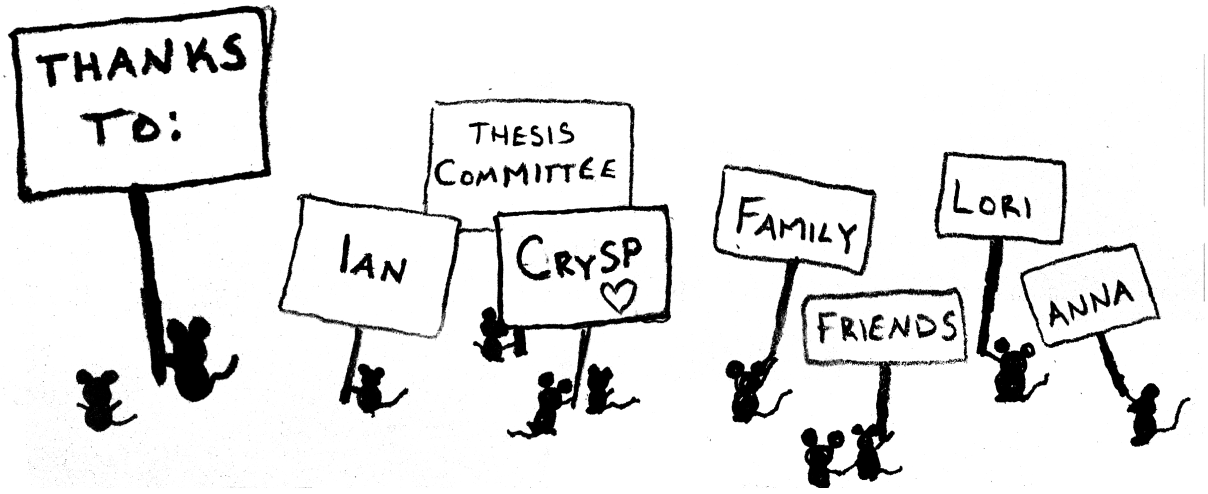
There are many opportunities to improve upon our contributions, in both the wider application of our recipes and in further improvements to our proposed system, Slitheen. As stated in Chapter 3, our method for protocol appropriation and the replacement of leaf data can be applied to other protocols as well as HTTPS. In particular, VoIP is a protocol that has long suffered from traffic analysis attacks yet provides a lot of potential in terms of a high-bandwidth, low-latency carrier for censorship resistance traffic. Our method would eliminate traffic analysis concerns and maximize the amount of goodput possible by the replacement of real video and audio data.

Although we have presented Slitheen as a decoy routing system, recent events surrounding the withdrawal of support for domain fronting suggest that an E2M deployment may suffer a

similar fate. Slitheen and other decoy routing systems can also be deployed in an E2E manner by performing the man-in-the-middle at the participating endpoint. The nature of Slitheen in adhering perfectly to the decoy site's traffic patterns make it an ideal candidate for a Tor pluggable transport and disguising the identities of Tor bridges, prolonging their lifespans and usefulness to the resistance community.

Finally, Slitheen itself can be made much more efficient and secure by further work on the relay station and the OUS. At the moment, the relay station state machine misses some resources due to WebM element headers being split across two packets. If we keep a buffer of previously seen data at the relay station, we can parse the header once the new packet is received rather than forfeit the flow. At the client side, we can automate the OUS to realistically simulate multiple web browsing sessions. For covert traffic that requires a lot of back-and-forth communication between the covert site and the user, we can access a combination of video sites that are good for high-bandwidth covert data and image sites that are good for multiple back-and-forth connections.

Our hope is that these recipes will lead to the design, development, and deployment of future systems that will not fall victim to advances in network analysis tools that have benefited censors and made many censorship circumvention systems in the last 20 years obsolete. Our hope is that eventually the detection of censorship resistance systems will become so difficult and their usage so popular that censors become useless and Internet users can begin to regain their ability to communicate freely and openly on the Internet.



References

[AAH13] 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)*, 2013.

[AFK16] Qurat-Ul-Ann Danyal Akbar, Marcel Flores, and Aleksandar Kuzmanovic. DNSsly: Avoiding censorship through network complexity. In *6th USENIX Workshop on Free and Open Communications on the Internet (FOCI 16)*. USENIX Association, 2016.

[And12] Daniel Anderson. Splinternet behind the great firewall of china. *ACM Queue*, 10.11:40, 2012.

[Ano14] Anonymous. Towards a comprehensive picture of the great firewall's DNS censorship. In *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI 14)*. USENIX Association, 2014.

[AW14] Yawning Angel and Philipp Winter. obfs4 (the obfourscator). https://gitweb.torproject.org/pluggable-transports/obfs4.git/tree/doc/obfs4-spec.txt, May 2014. [Online; accessed 6-June-2018].

[AWR14] Collin Anderson, Philipp Winter, and Roya. Global network interference detection over the RIPE Atlas network. In *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI 14)*. USENIX Association, 2014.

[BG16] Cecylia Bocovich and Ian Goldberg. Slitheen: Perfectly imitated decoy routing through traffic replacement. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, CCS '16, pages 1702–1714. ACM, 2016.

[BG18] Cecylia Bocovich and Ian Goldberg. Slitheen: Perfectly imitated decoy routing through traffic replacement. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2018(3), July 2018.

[Bow17] Owen Bowcott. Turks detained for using encrypted app 'had human rights breached'. https://www.theguardian.com/world/2017/sep/11/turks-detained-encrypted-bylock-messaging-app-human-rights-breached, 2017. [Online; accessed 6-June-2018].

[BSR17] Diogo Barradas, Nuno Santos, and Luís Rodrigues. DeltaShaper: Enabling unobservable censorship-resistant TCP tunneling over videoconferencing streams. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(4):1–18, 2017.

[Cen16] Center for Applied Internet Data Analysis. The CAIDA UCSD Statistical information for the CAIDA Anonymized Internet Traces. http://www.caida.org/data/passive/passive_trace_statistics.xml, 2016. Accessed 22-February-2017.

[Chi18] Richard Chirgwin. Google, AWS IPs blocked by Russia in Telegram crackdown. https://www.theregister.co.uk/2018/04/17/russia_blocks_google_aws_ip_addresses_to_get_telegram/, April 2018. [Online; accessed 6-June-2018].

[CKSR12] Jacopo Cesareo, Josh Karlin, Michael Schapira, and Jennifer Rexford. Optimizing the placement of implicit proxies. Technical report, Princeton, NJ, USA, 2012.

[CMW06] Richard Clayton, Steven J. Murdoch, and Robert N. M. Watson. Ignoring the Great Firewall of China. In *Sixth Workshop on Privacy Enhancing Technologies (PET 2006)*, page 20–35. Springer, June 2006.

[DCS15] Kevin P. Dyer, Scott E. Coull, and Thomas Shrimpton. Marionette: A programmable network-traffic obfuscation system. In *24th USENIX Security Symposium*, pages 367–382, 2015.

[Din12] Roger Dingledine. Obfsproxy: The next step in the censorship arms race. https://blog.torproject.org/blog/obfsproxy-next-step-censorship-arms-race, February 2012. [Online; accessed 6-June-2018].

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *13th USENIX Security Symposium*, pages 303–320, 2004.

[dP17] Nicholas de Pencier. Black code. Mongrel Media, 2017.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008.

[EDH+16] Tariq Elahi, John A Doucette, Hadi Hosseini, Steven J Murdoch, and Ian Goldberg. A framework for the game-theoretic analysis of censorship resistance. *Proceedings on Privacy Enhancing Technologies*, 2016(4):83–101, 2016.

[EFW+15] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. Examining how the great firewall discovers hidden circumvention servers. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 445–458. ACM, 2015.

[EJM+15] D. Ellard, C. Jones, V. Manfredi, W.T. Strayer, B. Thapa, M. Van Welie, and A. Jackson. Rebound: Decoy routing on asymmetric routes via error messages. In *Local Computer Networks (LCN), 2015 IEEE 40th Conference on*, pages 91–99, Oct 2015.

[EKAC14] Roya Ensafi, Jeffrey Knockel, Geoffrey Alexander, and Jedidiah R. Crandall. Detecting intentional packet drops on the Internet via TCP/IP side channels. In *Passive and Active Measurement*, pages 109–118. Springer, 2014.

[FDS+17] Sergey Frolov, Fred Douglas, Will Scott, Allison McDonald, Benjamin VanderSloot, Rod Hynes, Adam Kruger, Michalis Kallitsis, David G. Robinson, Steve Schultze, Nikita Borisov, Alex Halderman, and Eric Wustrow. An ISP-scale deployment of TapDance. In *7th USENIX Workshop on Free and Open Communications on the Internet (FOCI 17)*. USENIX Association, 2017.

[FGF+99] R. Fielding, J. Gettys, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[FHE+12] David Fifield, Nate Hardison, Jonathan Ellithorpe, Emily Stark, Dan Boneh, Roger Dingledine, and Phil Porras. Evading censorship with browser-based proxies. In *Proceedings of the 12th International Conference on Privacy Enhancing Technologies*, PETS'12, pages 239–258. Springer-Verlag, 2012.

[Fif17] David Fifield. *Threat modeling and circumvention of Internet censorship*. PhD thesis, University of California, Berkeley, 2017.

[Fif18] David Fifield. Probing obfs4 servers based on timing and byte count thresholds. https://groups.google.com/forum/#!topic/traffic-obf/GUrPkhqli0k, May 2018. [Online; accessed 7 August 2018].

[FLH+15] 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.

[GSH13] John Geddes, Max Schuchard, and Nicholas Hopper. Cover your ACKs: Pitfalls of covert channel censorship circumvention. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, CCS '13, pages 361–372. ACM, 2013.

[Han11] Serene Han. Snowflake technical overview. https://keroserene.net/snowflake/technical, January 2011. [Online; accessed 8-June-2018].

[HBS13] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. In *2013 IEEE Symposium on Security and Privacy*, pages 65–79, May 2013.

[HFKH06] Yihua He, Michalis Faloutsos, Srikanth Krishnamurthy, and Bradley Huffaker. On routing asymmetry in the Internet. In *Global Telecommunications Conference, 2005. GLOBECOM'05. IEEE*, volume 2, pages 6–pp. IEEE, 2006.

[HNCB11] Amir Houmansadr, Giang T.K. Nguyen, Matthew Caesar, and Nikita Borisov. Cirripede: Circumvention infrastructure using router redirection with plausible deniability. In *Proceedings of the 2011 ACM Conference on Computer and Communications Security*, CCS '11, pages 187–200, 2011.

[HR18] C. Huitema and E. Rescorla. SNI encryption in TLS through tunneling. Internet-Draft. https://tools.ietf.org/html/draft-ietf-tls-sni-encryption-02, March 2018.

[HRBS13] Amir Houmansadr, Thomas J Riedl, Nikita Borisov, and Andrew C Singer. I want my voice to be heard: IP over Voice-over-IP for unobservable censorship circumvention. In *2013 Network and Distributed System Security (NDSS) Symposium*, 2013.

[HWS14] Amir Houmansadr, Edmund L Wong, and Vitaly Shmatikov. No direction home: The true cost of routing around decoys. In *2014 Network and Distributed System Security (NDSS) Symposium*, 2014.

[HZCB17] A. Houmansadr, W. Zhou, M. Caesar, and N. Borisov. SWEET: Serving the web by exploiting email tunnels. *IEEE/ACM Transactions on Networking*, 25(3):1517–1527, June 2017.

[JDC10] Wolfgang John, Maurizio Dusi, and K. C. Claffy. Estimating routing symmetry on single links by passive flow measurements. In *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, IWCMC '10, pages 473–478. ACM, 2010.

[Jou06] Antoine Joux. Authentication failures in NIST version of GCM. 2006.

[KEJ+11] 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 *USENIX workshop on free and open communications on the Internet (FOCI 11)*, 2011.

[Ker83] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX:5–38, January 1883.

[KTS+16] Sanja Kelly, Mai Truong, Adrian Shahbaz, Madeline Earp, Jessica White, and Rose Dlougatch. Silencing the messenger: Communication apps under pressure. https://freedomhouse.org/report/freedom-net/freedom-net-2016, 2016. [Online; accessed 6-June-2018].

[Lew09] Andrew Lewman. Torproject.org blocked by GFW in China: Sooner or later? https://blog.torproject.org/blog/tor-partially-blocked-china, September 2009. [Online; accessed 6-June-2018].

[LSH14] Shuai Li, Mike Schliep, and Nick Hopper. Facet: Streaming over videoconferencing for censorship circumvention. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security*, CCS '14, pages 163–172, 11 2014.

[Mac18] Colm MacCarthaigh. Enhanced domain protections for amazon cloudfront requests. https://aws.amazon.com/blogs/security/enhanced-domain-protections-for-amazon-cloudfront-requests, April 2018. [Online; accessed 6-June-2018].

[Mar16] Moxie Marlinspike. Doodles, stickers, and censorship circumvention for Signal Android. https://signal.org/blog/doodles-stickers-censorship, December 2016. [Online; accessed 6-June-2018].

[Mar18] Moxie Marlinspike. Amazon threatens to suspend Signal's AWS account over censorship circumvention. https://signal.org/blog/looking-back-on-the-front, April 2018. [Online; accessed 6-June-2018].

[MHS16] 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.

[MMLDG12] 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*, CCS '12, pages 97–108, 2012.

[MV05] David McGrew and John Viega. The galois/counter mode of operation (GCM). 2005.

[Nab13] Zubair Nabi. The anatomy of web censorship in Pakistan. In *3rd USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[NAH18] Milad Nasr, Anonymous, and Amir Houmansadr. Massbrowser: Unblocking the web for the masses, by the masses. https://web.cs.umass.edu/publication/docs/2018/UM-CS-2018-003.pdf, 2018. [Online; accessed 8-June-2018].

[NH16] Milad Nasr and Amir Houmansadr. Game of decoys: Optimal decoy routing through game theory. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, CCS '16, pages 1727–1738. ACM, 2016.

[NZH17] Milad Nasr, Hadi Zolfaghari, and Amir Houmansadr. The waterfall of liberty: Decoy routing circumvention that resists routing attacks. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security*, CCS '17, pages 2037–2052, 2017.

[Pax06] Vern Paxson. End-to-end routing behavior in the Internet. *SIGCOMM Computer Communication Review*, 36(5):41–56, October 2006.

[Pet97] Fabien Petitcolas. Kerckhoffs' principles from «la cryptographie militaire». http://petitcolas.net/kerckhoffs, 1997. [Online; accessed 10 June 2018].

[PJL+17] Paul Pearce, Ben Jones, Frank Li, Roya Ensafi, Nick Feamster, Nick Weaver, and Vern Paxson. Global measurement of DNS manipulation. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 307–323. USENIX Association, 2017.

[SGTH12] Max Schuchard, John Geddes, Christopher Thompson, and Nicholas Hopper. Routing around decoys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 85–96, 2012.

[SSW10] Y. Schwartz, Y. Shavitt, and U. Weinsberg. On the diversity, stability and symmetry of end-to-end Internet routes. In *2010 INFOCOM IEEE Conference on Computer Communications Workshops*, pages 1–6, March 2010.

[TAAP16] M. C. Tschantz, S. Afroz, Anonymous, and V. Paxson. SoK: Towards grounding censorship circumvention in empiricism. In *2016 IEEE Symposium on Security and Privacy*, pages 914–933, May 2016.

[Tor15a] Tor Project. obfs2 transport evaluation. https://trac.torproject.org/projects/tor/wiki/doc/PluggableTransports/Obfs2Evaluation, 2015. [Online; accessed 6-June-2018].

[Tor15b] Tor Project. Tor: Bridges. https://www.torproject.org/docs/bridges, 2015. [Online; accessed 6-June-2018].

[WCQ+17] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V. Krishnamurthy. Your state is not mine: A closer look at evading stateful Internet censorship. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, pages 114–127. ACM, 2017.

[Whi18] Stephanie A. Whited. Domain fronting is critical to the open web. https://blog.torproject.org/domain-fronting-critical-open-web, April 2018. [Online; accessed 6-June-2018].

[Wil12] Tim Wilde. Great firewall Tor probing circa 09 DEC 2011. https://gist.github.com/twilde/da3c7a9af01d74cd7de7, December 2012. [Online; Accessed 6-June-2018].

[WL12] Philipp Winter and Stefan Lindskog. How the Great Firewall of China is blocking Tor. In *2nd USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2012.

[WPF13] Philipp Winter, Tobias Pulls, and Juergen Fuss. ScrambleSuit: A polymorphic network protocol to circumvent censorship. In *12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, pages 213–224, 2013.

[WSH14] 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, 2014.

[WWGH11] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: Anticensorship in the network infrastructure. In *20th USENIX Security Symposium*, 2011.

[WWY+12] 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*, CCS '12, pages 109–120, 2012.

[Xyn18] Maria Xynou. Investigating Internet blackouts from the edge of the network: OONI's new upcoming methodology. https://ooni.torproject.org/post/investigating-internet-blackouts/, April 2018. [Online; accessed 6-June-2018].

[ZH16] Hadi Zolfaghari and Amir Houmansadr. Practical censorship evasion leveraging content delivery networks. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, CCS '16, pages 1715–1726. ACM, 2016.