Summer 7-24-2015

Novel Cryptographic Primitives and Protocols for Censorship Resistance

Kevin Patrick Dyer
Portland State University

Recommended Citation

Dyer, Kevin Patrick, "Novel Cryptographic Primitives and Protocols for Censorship Resistance" (2015). *Dissertations and Theses.* Paper 2489.

10.15760/etd.2486

Novel Cryptographic Primitives and Protocols for Censorship Resistance

by

Kevin Patrick Dyer

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Thomas Shrimpton, Chair
Charles Wright
Wu-chang Feng
John Caughman IV

Portland State University
2015

Abstract

Internet users rely on the availability of websites and digital services to engage in political discussions, report on newsworthy events in real-time, watch videos, etc. However, sometimes those who control networks, such as governments, censor certain websites, block specific applications or throttle encrypted traffic. Understandably, when users are faced with egregious censorship, where certain websites or applications are banned, they seek reliable and efficient means to circumvent such blocks. This tension is evident in countries such as a Iran and China, where the Internet censorship infrastructure is pervasive and continues to increase in scope and effectiveness.

An arms race is unfolding with two competing threads of research: (1) network operators' ability to classify traffic and subsequently enforce policies and (2) network users' ability to control how network operators classify their traffic. Our goal is to understand and progress the state-of-the-art for both sides. First, we present novel traffic analysis attacks against encrypted communications. We show that state-of-the-art cryptographic protocols leak private information about users' communications, such as the websites they visit, applications they use, or languages used for communications. Then, we investigate means to mitigate these privacy-compromising attacks. Towards this, we present a toolkit of cryptographic primitives and protocols that simultaneously (1) achieve traditional notions of cryptographic security, and (2) enable users to conceal information about their communications, such as the protocols used or websites visited. We demonstrate the utility of these primitives and protocols in a variety of real-world settings. As a primary use case, we show that these new primitives and protocols protect network communications and bypass policies of state-of-the-art hardware-based and software-based network monitoring devices.

Dedication

To my brilliant and beautiful wife Susannah, who made this work possible, enjoyable and meaningful.

Acknowledgements

This work is a summary of published [47, 48, 36] and unpublished results. The work in Section 3 was done with Coull and published [36] in ACM SIGCOMM Computer Communication Review in 2014. The work in Section 4 was presented [47] at IEEE Security and Privacy in 2012, and done in collaboration with Coull, Ristenpart and Shrimpton. The work in Section 6 was presented [48] at ACM Conference on Computer and Communications Security in 2013, and was also in collaboration with Coull, Ristenpart and Shrimpton. Finally, the work in Section 7 was presented [49] at USENIX Security 2015 and done in collaboration with Coull and Shrimpton.

This dissertation was possible because of the patience, guidance and funding provided by my advisor Thomas Shrimpton. The quality, consistency, and vision for this work was immeasurably elevated by my coauthors Scott Coull and Thomas Ristenpart.

I am grateful for the patience of the engineers I collaborated with at Google, The Tor Project, and Lantern, for supporting software deployments of the ideas in this document. In addition, this work was made possible by a number of generous sponsors, including: Eric Schmidt, Fariborz Maseeh, The NLnet Foundation, The National Science Foundation, and Portland State University.

What's more, while pursuing the ideas in this document I had the pleasure of discussing it with, and getting feedback from many wonderful people, including: Will Landecker, Lucas Dixon, Roger Dingledine, George Kadianakis, David Fifield, Shanjian Li, Trevor Johnston, Akshay Dua, Tariq Elahi, Tim Chevalier, Charles Wright, Justin Reidy, R. Seth Terashima, and Soeren Pirk.

Table of Contents

List of Tables

List of Figures

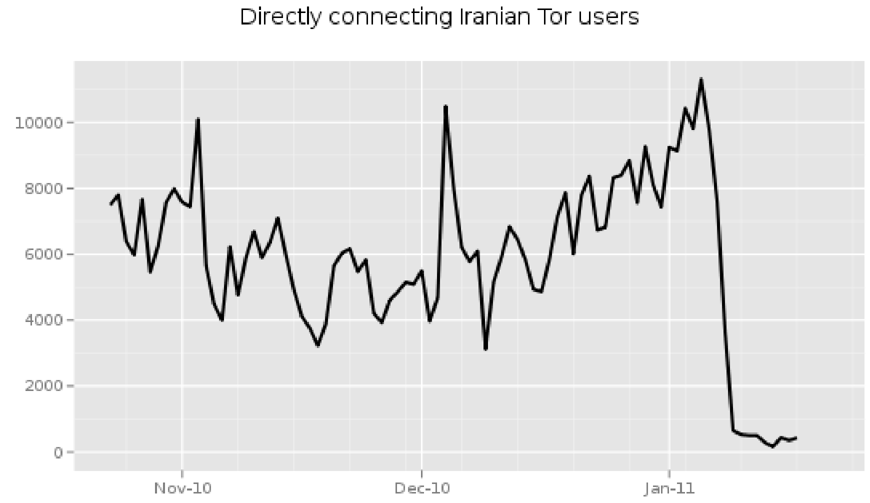
1 Introduction

Traffic analysis (TA) is the study of network traffic features. Features may include, but are not limited to: message timings or lengths, the contents of a message, or even the communicating parties. TA is commonly employed to give network operators insights into their networks. In some cases, TA is used to provide Quality of Service (QoS) and give higher priority to time-sensitive traffic, such as VoIP or real-time video. In other cases it is used to help network operators understand who or what is utilizing their network resources. However, this same technology is controversially used by governments to censor websites, such as YouTube, or to block applications, such as Tor [44].

Worldwide, more than three billion [111] people regularly access the Internet. Unfortunately, as much as a third [17] of Internet users are subject to surveillance or censorship by their own government. Hence, we are interested in specific cases of the following two questions: *What information can a network operator infer about the behavior of its users?* and *How can network users ensure the privacy of their communications?* On one hand, if a network operator wishes to enforce a specific policy, such as *restrict all access to BBC News*, how would they do so? Conversely, if a user wants to *conceal that they are visiting BBC News* from their network operator, what tools should they use? We'll spend the remainder of this dissertation exploring these questions.

1.1 Censorship Circumvention

There are a number of tools that a user might employ to protect the contents of their communications. As one example, state-of-the-art encryption does a good job concealing message contents. Nevertheless, modern encryption has fundamental lim-



The Tor Project - https://metrics.torproject.org/

Figure 1: The number of directly connecting users in Iran to the Tor network from late 2010 to early 2011. In January 2011 Iran applied blocks to restrict connections to the Tor network, which resulted in the sudden drop in usage.

itations.

How Iran banned encryption. Let's consider at a concrete example in Iran of how encryption failed to protect users' communications. On January 9, 2011 users from Iran were unable to access the Internet using privacy tools such as Tor. The blocks set in place were both immediate and effective, as we can see in Figure 1. Iran was able to prevent users from connecting to the Tor Network using *IP-based blocking* — if the server's address was on Iran's blacklist, it was inaccessible from Iran.

This is a case where, despite well-implemented cryptographic protocols, sensitive information about the users' communications was used to censor a certain type (i.e., Tor) of traffic. The government was not required to achieve a substantial cryptographic breakthrough or expend significant financial resources. Yet, they were able to restrict all access to the Tor network on a national scale.

State-of-the-art cryptography is insufficient. Intuitively, it should be the case that state-of-the-art cryptography renders these questions uninteresting. That is, if fundamentally-sound cryptographic protocols are implemented, deployed and used properly, network operators should be unable to infer any information about a user's behavior — unfortunately, this is an invalid assumption for many reasons. State-of-the-art cryptography may do a good job concealing information about individual messages. However, it does not conceal many important characteristics about communications, such as: *Which two users are communicating?* or *How much data is a specific user sending/receiving?*. On the surface, these questions may seem innocuous. However, in some countries, specific types of encryption are disallowed by network policy — hence, by using certain applications, such as Tor [44], one may incriminate themselves. More subtly, it has been shown repeatedly that it's possible to infer a user's web browsing behavior [16, 73, 60, 92, 24, 83], even when all communications are encrypted, by analyzing the lengths and timings of ciphertexts transmitted.

Overview The remainder of this dissertation is divided into seven parts. In Section 2 we survey prior literature in the area of traffic analysis. In Section 3 we explore the limitations of state-of-the-art cryptographic protocols against traffic analysis attacks. We show that popular encrypted messaging services, such as Apple's iMessage and WhatsApp, do a poor job concealing the private information of users. In Section 4 we provide a series of negative results and show that a host of per-packet countermeasures are not sufficient to protect against state-of-the-art-attacks [73, 60, 92] on user web browsing habits. In Section 5 we survey prior literature in the area of censorship and censorship circumvention. Towards resisting a specific class of these TA attacks, we propose a novel strategy in Section 6 for bypassing regular-expression-based network monitoring systems. In Section 7 we present generic, novel programmable network traffic obfuscation framework, which enables users to control their network traffic features, including message lengths and formats. Using this framework, one can bypass sophisticated network monitoring systems that analyze multiple messages in a datastream or maintain inter-connection state. We conclude with Section 8 and present thoughts for future works in this area.

2 Overview: Traffic Analysis of Encrypted Communications

Internet users increasingly rely upon encryption to secure their communications. However, unfortunately, it has been shown across many settings that state-of-the-art cryptography can leak private information about its users. This includes the analysis of encrypted VoIP traffic [130, 128, 129, 122] to recover spoken phrases. Alternatively, it has been shown that sensitive user input [29, 77, 28] can be revealed by observing multiple, encrypted web requests from a client.

In another setting, inter-packet timings were used to reduce the complexity of a brute-force attack against SSH passwords [105]. Techniques have been presented that show how to identify applications in an encrypted tunnel [15, 30, 46, 45, 138]. User traffic patterns, and the applications they use, can leak sensitive information [69, 70, 67, 66, 136], despite encryption, too.

We refer to these types of privacy failures as *traffic analysis attacks*. In this section we focus on two specific types of traffic analysis (TA) attacks. First, we'll discuss *attacks on encrypted messaging services* which reveal information about users, such as their host operating system, language used for communications, and even the lengths of word/sentences communicated during a conversation. Then, we'll consider *website fingerprinting attacks* that reveal sensitive information about users' web browsing behavior, despite strong encryption.

2.1 Attacks on Encrypted Messaging Services

Informally, we refer to *attacks on encrypted messaging services* as any type of attack that enables an adversary to learn sensitive information about a user of the service, such as Apple's iMessage, despite encryption. As a couple examples, it could be possible for an encrypted messaging service to leak sensitive information, such as:

the user's operating system, language of communications, or even the time of day they are using their computer.

Given that services such as Apple's iMessage are proprietary, this presents challenges in understanding how the systems are implemented and how they might leak sensitive information. iMessage uses the Apple Push Notification Service (APNS) to deliver text messages and attachments to users. To date, there have been two primary efforts in understanding the operation of the iMessage service and the APNS protocol. Frister and Kreichgauer have developed the open source Push Proxy project [51], which allows users to decode APNS messages into a readable format by redirecting those messages through a man-in-the-middle proxy. In another recent effort, Matthew Green [56] and Ashkan Soltani [55] showed that, while iMessage data is protected by end-to-end encryption, the keys used to perform that encryption are mediated by an Apple-run directory service that could potentially be used by an attacker (or Apple themselves) to install their own keys for eavesdropping purposes. To the best of our knowledge, there are no works that specifically examine the privacy of encrypted instant messaging services, particularly those used by mobile devices. This is surprising, given that a highly-accurate attack could affect nearly a billion users across a wide variety of messaging services.

2.2 Website Fingerprinting Attacks

A *website fingerprinting attack* has been informally defined in the literature as the following. A user establishes an encrypted connection to a proxy server, and an adversary is able to observe all ciphertexts between the client and proxy. The adversary knows *a priori* a set S of possible websites a user may visit and is able to train and test their classifier on traffic traces. In most cases, it is assumed that traffic traces are free from real-world artifacts, such as proxy-side caching. If the client is only able to request websites in the set S we call this the *closed-world* setting. If the client is not restricted to the websites in S, we call it the *open-world* setting. We refer to the set S as the *privacy set*, and its size as $k = |S|$. When a user requests a website, it's the job of the adversary to determine which website in S was requested. If the website requested is not in S, the adversary must indicate so, but is not required to identify the specific website. In the closed-world setting, the adversaries' accuracy is defined as the probability that it guesses the correct website. In the open-world setting, as noted in [3], there are a number of different success metrics to consider in addition to accuracy, including the false-positive rate. (i.e., outputting that a request is from a website in S, when it's not)

There is an extensive history of literature on traffic analysis and website fingerprinting attacks. The first academic discussion of website fingerprinting attacks was by Wagner and Schneier [114]. They relayed an observation of Yee that SSL might leak the URL of an HTTP GET request because ciphertexts leak plaintext length. Wagner and Schneier suggested that per-ciphertext random padding should be included for all cipher modes of SSL.

Cheng and Avnur [31] provided some of the first experimental evidence of web page fingerprinting attacks by analyzing pages hosted within one of three websites. Their attack assumes perfect knowledge of HTML and web page object sizes, which is not always precisely inferred from ciphertexts. They also suggested countermeasures including padding of HTML documents, fixed-length padding, and introduction of spurious HTTP requests.

Sun et al. [106] investigated a similar setting, in which the adversary can precisely uncover the size of individual HTTP objects in a non-pipelined, encrypted HTTP connection. They provided a thorough evaluation utilizing a corpus of 100,000 websites. They described a classifier based on the Jaccard coefficient similarity metric and a simple thresholding scheme. They also explored numerous countermeasures, including per-packet padding, byte-range requests, client-based prefetching, server-based pushing of content, content negotiation, web ad blockers, pipelining, and using multiple browsers in parallel.

Hintz [61] discussed a simple attack for identifying which of five popular web pages was visited over a single-hop proxy service called SafeWeb. The proposed attack does not require exact knowledge of web request sizes, but there is little evaluation and it remains unclear how the attack would fair with larger privacy sets.

Bissias et al. [16] demonstrated a weaker adversary than that of Sun et al. [106], which could observe an SSH tunnel and view only the length, direction, and timing of each ciphertext transmitted, rather than web page objects. They used cross-correlation to determine webpage similarity, which is a metric commonly used for evaluating the similarity of two time series.

Liberatore and Levine [73] showed that it is possible to infer the contents of an HTTP transaction encapsulated in an SSH connection by observing only encrypted packet lengths and the directions of unordered packets. They quantify the ability of several countermeasures, including linear, exponential, and fixed-length padding schemes, to protect against their attack, but only report on a privacy set size of $k = 1000$.

Herrmann et al. [60] collected encrypted traces from four different types of single-hop encryption technologies, and two multi-hop anonymity networks. They were the first to suggest the use of a multinomial naïve Bayes classifier for traffic classification that examines normalized packet counts. Their evaluation of countermeasures was restricted to application-layer countermeasures.

Panchenko et al. [92] presented a support vector machine classifier as an improvement upon the work of Herrmann et al. [60]. They apply it to Tor [44] traffic they generated in both a closed-word and open-world setting, showing good accuracy.

Cai et al. [24] showed that ad-hoc, application-layer defenses [77] provide little benefit for privacy sets of size $k = 100$. Wang et al. [117] demonstrated that high-accuracy website fingerprinting is possible in the open-world setting when using Tor. Chen et al. [29] and Miller et al. [83] evaluated web applications, and showed that they can leak sensitive legal, financial, or health information about users.

In other works, more principled approaches are taken to quantify information leakage in web applications [80] and to determine the efficacy of defenses [23, 116]. More recently Shi et al. [101] showed that algorithms traditionally used in signal processing could be applied to improve the efficacy of website fingerprinting attacks against webpages with dynamic contents. Mather et al. [79] showed that mutual information between the user inputs and web applications responses can be used to determine the potential for information leakage.

In contrast, a host of strategies have been proposed to mitigate website fingerprinting attacks. Wright et al. [127] suggested traffic morphing, which can minimize padding overhead while still making one web page “look like" another with respect to specific features. Luo et al. [77] presented HTTPOS, which describes a number of client side mechanisms, such as adding superfluous data to HTTP GET requests, in order to thwart TA attacks. Backes et al. [11] modeled web applications and the information they can leak, which may be a step towards general-purpose countermeasures.

Nithyanand et al. suggests a countermeasures that relies upon *a priori* knowledge of traffic that is being transmitted [90], which is a strong assumption, but can greatly increase efficiency. CS-BuFLO by Cai et al. [22], presents a more efficient variant of sending fixed-length packets at fixed-time intervals.

Finally, we note that the website fingerprinting attacks and defenses in the literature span a wide range of settings. Even though these results may be reproducible in controlled settings, there remain questions about how they scale in practice. Juarez et al. [3] address this, and show that certain variables such as user's browsing habits, caching, differences in location and browser version can have a significant impact on the efficacy of attacks and defenses.

3 Traffic Analysis of Encrypted Message Services

Over the course of the past decade, instant messaging services have gone from a niche application used on desktop computers to the most prevalent form of communication in the world, due in large part to the growth of Internet-enabled phones and tablets. Messaging services, like Apple iMessage and WhatsApp, handle *tens of billions* of messages each day from an international user base of over one billion people [74, 91]. Given the volume of messages traversing these services and ongoing concerns over widespread eavesdropping of Internet communications, it is not surprising that privacy has been an important topic for both the users and service providers. To protect user privacy, these messaging services offer transport layer encryption technologies to protect messages in transit, and some services, like iMessage and Telegram, offer end-to-end encryption to ensure that not even the providers themselves can eavesdrop on the messages [8, 57]. However, as we explored in Section 2, state-of-the-art encryption does not guarantee the privacy of the underlying message contents.

In this section, we analyze the network traffic of popular encrypted messaging services to (1) understand the breadth and depth of their information leakage, (2) determine if attacks are generalizable across services, and (3) calculate the potential costs of protecting against this leakage. Specifically, we focus our analysis on the Apple iMessage service and show that it is possible to reveal information about the device operating system, fine-grained user actions, the language of the messages, and even the approximate message length with accuracy exceeding 96%, as shown in the summary provided in Table 1. In addition, we demonstrate that these attacks are applicable to many other popular messaging services, such as WhatsApp, Viber, and Telegram, because they target deterministic relationships between user actions and the resultant encrypted packets that exist regardless of the underlying encryption

Attack	**Method**	**Accuracy**
Operating System	Naïve Bayes	100%
User Action	Lookup Table	96%
Language	Naïve Bayes	98%
Message Length	Linear Regression	6.27 chars.

Table 1: Summary of attack results for Apple iMessage.

methods or network protocols used. Our analysis of countermeasures shows that the attacks can be completely mitigated by adding random padding to the messages, but at a cost of over 300% overhead, which translates to at least a *terabyte* of extra data *per day* for the service providers. Overall, these attacks could impact over a billion users across the globe and the high level of accuracy that we demonstrate in our experiments means that they represent realistic threats to privacy, particularly given recent revelations about widespread metadata collection by government agencies [35].

Acknowledgements The work described in this section was done in collaboration with Coull [36] and a version was published in the ACM SIGCOMM Computer Communication Review in October 2014. Coull lead the project in regards to the vision, data collection and writing. Dyer lead the engineering efforts and assisted with the data collection and writing.

3.1 iMessage Overview

In this section we'll provide an overview of the iMessage service. Readers interested in the low-level details of iMessage should refer to documentation from projects focused on reverse engineering specific portions of the iMessage service [51, 55, 56], or the official Apple iOS security white paper [8]. The following information was accurate as of August 2013.

iMessage uses the Apple Push Notification Service (APNS) to deliver text messages and attachments to users. When the device is first registered with Apple, a client certificate is created and stored on the device. Every time the device is connected to the Internet, a persistent APNS connection is made to Apple over TCP port 5223. The connection appears to be a standard TLS tunnel protecting the APNS messages. From here, the persistent APNS connection is used to send and receive both control messages and user content for the iMessage service, as well as other Apple services (*e.g.,* FaceTime). If the user has not recently interacted with the sender or recipient of a message, then the client initiates a new TLS connection with Apple on TCP port 443 and receives key information for the opposite party. Unlike earlier TLS connections, this one is authenticated using the client certificate generated during the registration process[1]. Once the keys are established, there are five user actions that are observable through the APNS and TLS connections made by the iMessage service. These actions include: (1) start typing, (2) stop typing, (3) send text, (4) send attachment, and (5) read receipt. All of the user actions mentioned follow the protocol flow shown in Figure 2, except for sending an attachment. The protocol flow for attachments is quite similar except that the attachment itself is stored in the Microsoft Azure cloud storage system before it is retrieved, rather than being sent directly through Apple.

Over the course of our analysis, we observed some interesting deviations from this standard protocol. For instance, when TCP port 5223 is blocked, the APNS message stream shifts to using TCP port 443. Similarly, cellular-enabled iOS devices use port 5223 while connected to the cellular network, but switch to port 443 when WiFi is used. Moreover, if the iOS device began its connection using the cellular network, that connection will remain active even if the device is subsequently connected to a wireless access point. It is important to note that payload sizes and general APNS protocol behaviors remain exactly the same whether port 5223 or 443 are used, and

[1] The client certificate provides an identifier that we can use to develop a social network of iMessage communications.

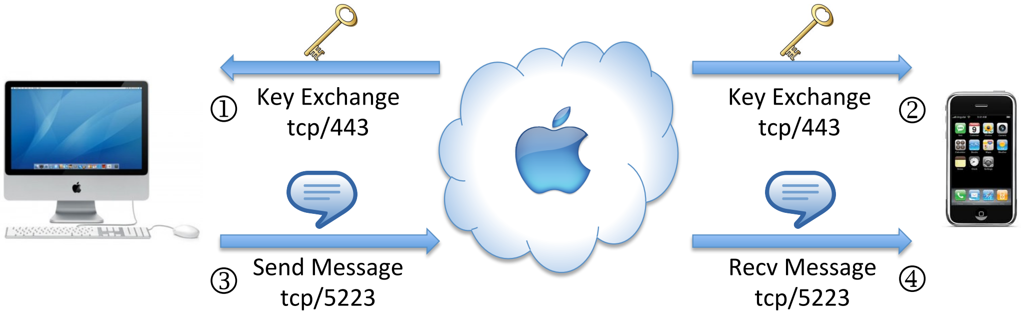


Figure 2: High-level operation of iMessage.

therefore the attacks we present in this section are equally applicable in both cases.

3.2 Analyzing Information Leakage

In this section, we investigate information leakage about devices, users, and messages by analyzing the relationship between packet sizes within the persistent APNS connection used by iMessage and user actions. For each of these categories of leakage, we first provide a general analysis of the data to discover trends or distinguishing features, then evaluate classification strategies capable of exploiting those features.

3.2.1 Data and Methodology

To evaluate our classifiers, we collected data for each of the five observable user actions (start, stop, text, attachment, read) by using scripting techniques that drove the actual iMessage user interfaces on OSX and iOS devices. Specifically, we used ApplesScript to natively type text, paste images, and send/read messages on a Macbook Pro running OSX 10.9.1, and a combination of VNC remote control software

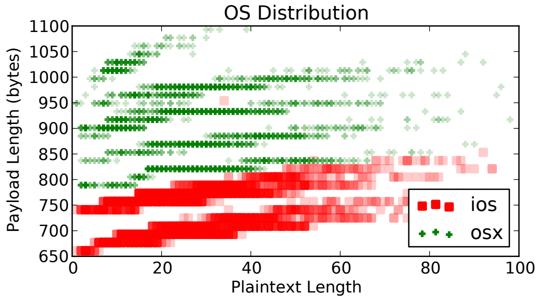


Figure 3: Scatter plot of plaintext message lengths versus ciphertext lengths for packets containing user content.

and AppleScript to control the same actions on a jailbroken iPhone 4 (iOS 6.1.4). For each user action, we collected 250 packet capture examples on both devices and in both directions of communications (*i.e.,* to/from Apple) for a total of 5,000 samples. This allows us to simulate an adversary that can passively monitor streaming iMessage traffic to or from Apple servers (steps 3 and 4 in Figure 2, respectively). In addition, we also collected small samples of data using devices running iOS 5, iOS 7, and OSX Mountain Lion to verify the observed trends.

The underlying text data is drawn from a set of over one million sentences and short phrases in a variety of languages from the Tatoeba parallel translation corpus [109]. Languages used in our evaluation include Chinese, English, French, German, Russian, and Spanish. For attachment data, we randomly generated PNG images of exponentially increasing size (64 x 64, 128 x 128, 256 x 256). Throughout the remainder of the section, we simply refer to attachments as “image" messages. Although the Tatoeba dataset does not contain typical text message shorthand, it is generated through a community of non-expert users (*i.e.,* crowd-sourced) and so actually contains several informal phrases that are not found in a typical language translation corpus. In fact, the distribution of English message lengths in our data, as shown in Table 2, is quite close to those reported in recent studies of text messaging behavior by Battestini et al. [12], with our data exhibiting a slightly shorter average message

Language	Avg Length (σ)
chinese	13.3 (8.0)
english	**40.0 (21.5)**
french	41.1 (26.8)
german	41.5 (27.4)
russian	36.4 (27.4)
spanish	39.1 (21.9)
english text msgs [12]	**50.9 (46.2)**

Table 2: Language statistics for Tatoeba dataset compared to Battestini et al. text messaging study.

length and less variability.

Each experiment in this section used 10-fold cross validation testing, where the data for each instance in the test was constructed by sampling TCP payload lengths and packet directions (*i.e.,* to/from Apple) from the relevant subset of the packet capture files. Cross-validation testing was performed such that the classes were equally represented in the sampled data (*i.e.,* uniform prior probability across all classes). The only preprocessing that was performed on the data was to remove duplicate packets that occur as a result of TCP retransmissions and those packets without TCP payloads. Performance of our classifiers is reported with respect to overall accuracy, which is calculated as the sum of the true positives and true negatives over the total number of samples evaluated. Where appropriate, we also use confusion matrices that show how each of the test instances was classified and use absolute error to measure the predictive error in our regression analysis.

3.2.2 Operating System

Our first experiment examines the difference in the observable packet sizes for the iOS and OSX operating systems. The scatterplot of iMessage packet sizes in Figure 3 shows how iOS appears to more efficiently compress the plaintext, while OSX occupies a much larger space. These two classes of data are clearly separable, but the figure also

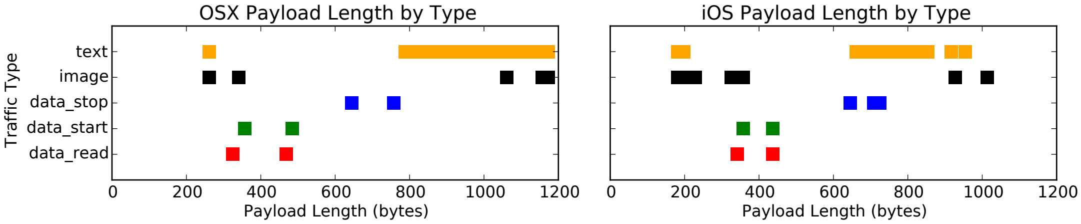


Figure 4: Distribution of payload lengths for each message type separated by operating system without control packets.

shows five unique bands of plaintext/ciphertext relationship, which hints at leakage of finer-grained information about the individual messages (which we examine in Section 3.2.4). Additionally, when we break down the distributions based on their direction (*i.e.,* to/from Apple), we see that there is a deterministic relationship between the two. That is, as messages pass through Apple, 112 bytes of data are removed from OSX messages and 64 bytes are removed from iOS messages. Aside from the ability to fingerprint the OS version, the deterministic nature of these changes indicates that it is also possible to correlate and trace communications as it passes through Apple on the way to its destination, thereby allowing us to develop a communications graph.

To identify the OS of observed devices, we use a binomial naïve Bayes classifier from the Weka machine learning library [59] with one class for each of the four possible OS, direction combinations. The classifier operates on a binary feature vector of packet length, direction pairs, where the value for a given dimension is set to “true" if that pair was observed and “false" otherwise. To determine the number of packet observations necessary for accurate classification, we run 10-fold cross-validation experiments where the 1,024 instances used for each experiment are created with $N = 1, 2, \ldots, 50$ packets sampled from the appropriate subset of the dataset for each OS, observation point class. The results indicate that we are able to accurately classify the OS with 100% accuracy after observing only five packets regardless of the operating system. A cursory analysis of iOS 5 and 7 indicates that they also

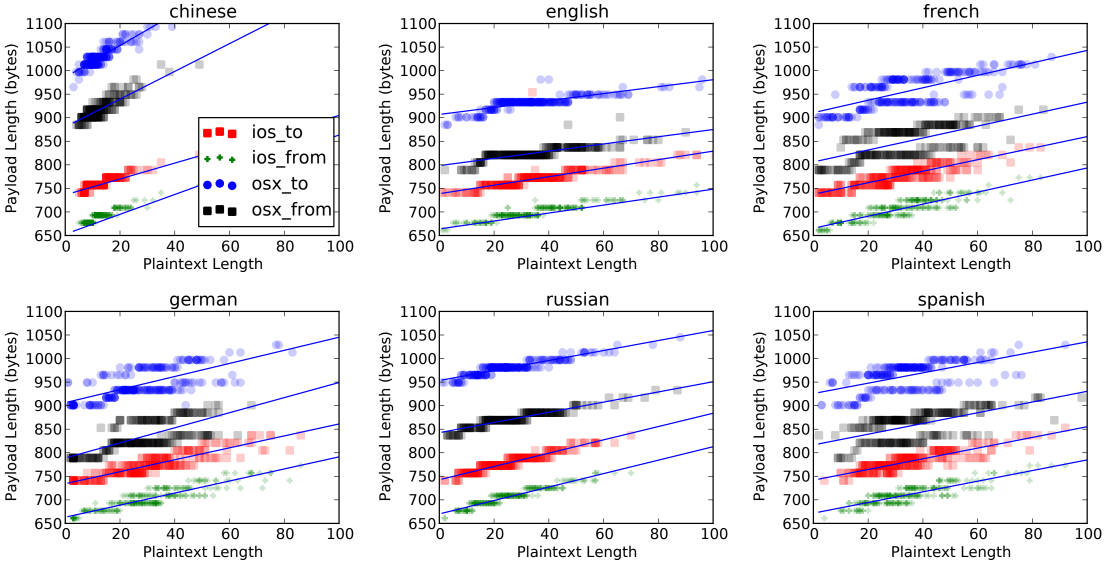


Figure 5: Scatter plots of plaintext message lengths versus payload lengths for six languages in our dataset.

produce messages with lengths that are unique from both the OSX and iOS 6.1.4 device, which indicates that this type of device fingerprinting could be refined to reveal specific version information when the size of the APNS messages changes between OS versions.

3.2.3 User Actions

Recall from our earlier discussion that there are five high-level user actions that we can observe: start, stop, text, attachment (image), and read. Figure 4 shows the distribution of payload lengths for each of these actions separated by the OS of the sending device after removing control packets (*i.e.*, packet sizes that occur within multiple classes). Most classes have two distinctive packet lengths – one for when the message is sent to Apple and one when it is received from Apple. The only classes that overlap substantially are the read receipt and start messages in the iOS data going to Apple.

The stability and deterministic nature of the lengths in most classes makes the

OSX

control	read	start	stop	image	text
1.0	0.0	0.0	0.0	0.0	0.0
0.0	**1.0**	0.0	0.0	0.0	0.0
0.0	0.0	**1.0**	0.0	0.0	0.0
0.0	0.0	0.0	**1.0**	0.0	0.0
0.0	0.0	0.0	0.0	**1.0**	0.0
0.005	0.0	0.0	0.0	0.0	**0.995**

iOS

control	read	start	stop	image	text
0.99	0.0	0.0	0.0	0.0	0.01
0.0	**0.5**	0.5	0.0	0.0	0.0
0.0	0.0	**1.0**	0.0	0.0	0.0
0.005	0.0	0.0	**0.995**	0.0	0.0
0.005	0.0	0.005	0.0	**0.99**	0.0
0.005	0.0	0.0	0.0	0.0	**0.995**

Table 3: Confusion matrix for message type classification.

use of probabilistic classifiers unnecessary. Instead of using heavyweight machine learning methods, we create a hash-based lookup table using each observed length in the training data as a key and store the associated class labels. In addition to creating classes for the five standard message types derived from user actions, we also create a class for the payload lengths of identified control packets. When a new packet arrives, we check the lookup table to retrieve the class label(s) for its payload length. If only one label is found, the packet is labeled as that message type. In the case where two class labels are returned, we choose the class where that payload length occurs most frequently in the training data.

In an effort to focus our evaluation, we assume that the OS has already been accurately classified such that we have four separate message-type classifiers, one for each combination of OS and direction. Each of the classifiers is evaluated using 10-fold cross validation with instances drawn from the respective subsets of the dataset, for a total of 1,250 instances per classifier. Confusion matrices showing the results for

OSX and iOS are presented in Table 3. The accuracy is surprisingly good for both iOS and OSX given such a simple classification strategy. As it turns out, all message types can be classified with accuracy exceeding 99%, except for iOS read messages that are easily confused with start messages, as was suggested by Figure 4.

3.2.4 Message Attributes

The final experiment in our analysis of information leakage examines if it is possible to learn more detailed information about the contents of messages, such as their language or plaintext length. The foundation for this experiment is built upon the observation that Figure 3 (in Section 3.2.2) shows several distinct clusters when comparing plaintext message length to payload length. While the clusters are most prevalent in the OSX data, the iOS data also has a similar set of clusters (albeit more compressed). When we separate this data into its constituent languages, as in Figure 5, the reason for these clusters becomes clear. Essentially, each cluster represents a unique character set used in the language (*e.g.,* ASCII, Unicode). For languages that use only a single character set, like English (ASCII), Russian (Unicode), or Chinese (Unicode), there is only one cluster approximating a linear relationship between plaintext and payload lengths, with a “stair step" effect at AES block boundaries. The other three languages all use some mix of ASCII and Unicode characters, resulting in an ASCII cluster with better plaintext/payload length ratios, and Unicode cluster that requires more payload bytes to encode the plaintext message. These graphs also help to answer our question about the possibility of guessing the message lengths, which is supported by the approximately linear relationship that appears.

To test our ability to classify these languages, we use the Weka multinomial naïve Bayes classifier, with raw counts of each length, direction pair observed so that we can take full advantage of the subtle differences in the distribution. As with previous

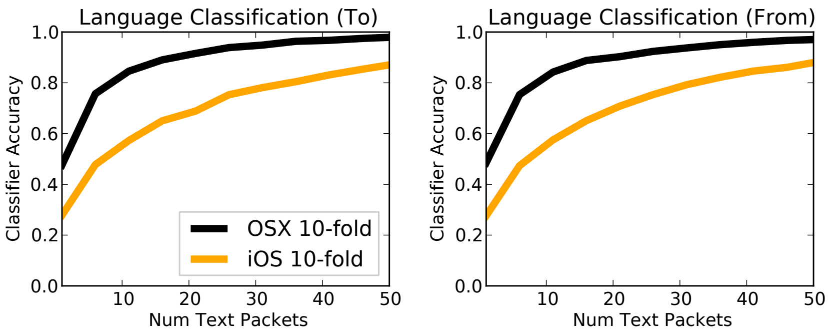


Figure 6: Language classification accuracy.

experiments, we assume that earlier classification stages for OS and message type were 100% accurate in order to focus specifically on this area of leakage. The results from 10-fold cross validation on 1,024 instances generated from $N = 1, 2, \ldots, 50$ text message packets are shown in Figure 6. Classification of languages in OSX data is noticeably better than iOS, as we might have expected due to compression. On the OSX data, we achieve an accuracy of over 95% after 50 packets are observed. When applied to the iOS data, on the other hand, accuracy barely surpasses 80% at the same number of packets. However, as the confusion matrices in Table 4 show, by the time we sample 100 packets all languages are achieving classification accuracies of at least 93% regardless of the dataset.

Given that language classification can be achieved with high accuracy after a reasonable number of observations, we now move on to determining how well we can predict message lengths within those languages. For this task, we apply a simple linear regression model using the payload length as the explanatory variable and the message length as the dependent variable. The models are fitted to the training data using least squares estimation. Again, we performed 10-fold cross validation with 1,024 instances and calculated the resultant absolute error. In general, the values are small – an error of between 2 and 11 characters – when we consider that the sentences in the language dataset range from two characters to several hundred, with

OSX

chinese	english	french	german	russian	spanish
1.0	0.0	0.0	0.0	0.0	0.0
0.0	**1.0**	0.0	0.0	0.0	0.0
0.0	0.0	**0.985**	0.0	0.0	0.015
0.0	0.0	0.0	**1.0**	0.0	0.0
0.0	0.0	0.0	0.0	**1.0**	0.0
0.0	0.0	0.01	0.0	0.0	**0.99**

iOS

chinese	english	french	german	russian	spanish
1.0	0.0	0.0	0.0	0.0	0.0
0.0	**0.995**	0.0	0.0	0.005	0.0
0.0	0.0	**0.95**	0.035	0.015	0.0
0.0	0.0	0.03	**0.965**	0.005	0.0
0.0	0.005	0.015	0.0	**0.95**	0.03
0.0	0.0	0.01	0.0	0.06	**0.93**

Table 4: Confusion matrix for language classification.

an average error of 6.27 characters. This is equivalent to an 18.4% average error rate based on the statistics from Table 2. Those languages with multiple clusters, like French and Spanish, fared the worst since the linear regression model could not handle the bimodal behavior of the distribution for the multiple character sets. For completeness, we also applied a regression model to the image transfers to and from the Microsoft Azure cloud storage system. The regression model was extremely accurate for the attachments, with an absolute error of less than 10 bytes.

3.3 Beyond iMessage

Thus far, we have focused our attacks exclusively on Apple iMessage, however we note that they rely only on the user's interaction with the messaging service and a deterministic relationship between those actions and packet sizes. In effect, the attacks target fundamental operations that are common to all messaging services. To illustrate this concept, we used the same data generation procedures described in

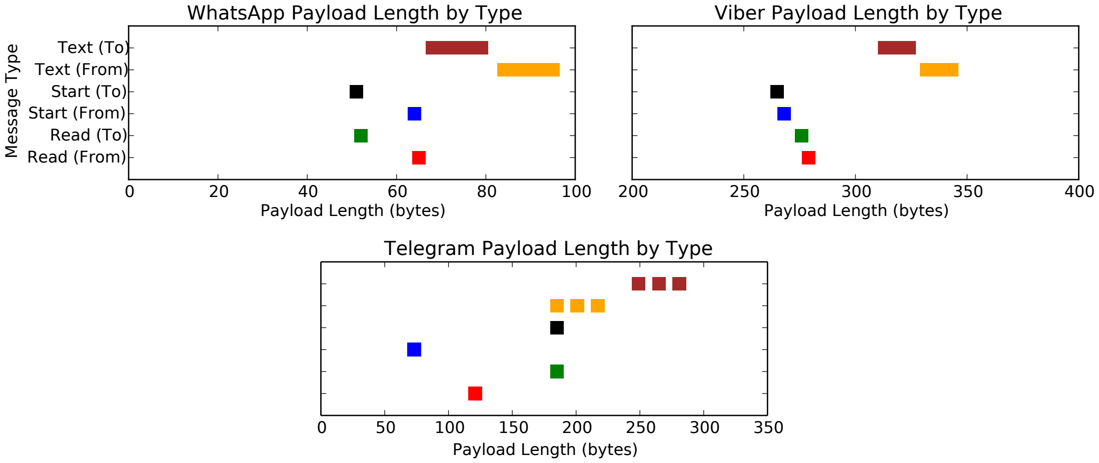


Figure 7: Distribution of payload lengths by type for WhatsApp, Viber, and Telegram.

Section 3.2.1 to examine the leakage of user actions and message information in the WhatsApp, Viber, and Telegram messaging services. Figure 7 shows the distribution of packet lengths associated with the user actions that we have considered throughout this section for those services. Just as with Apple iMessage (*c.f.,* Figure 4), these three messaging services clearly allow us to differentiate fine-grained activities by examining individual packet sizes. Moreover, when we examine the relationship between plaintext message lengths and ciphertext length, as in Figure 8, there is a clear linear relationship between the two.

Figures 7 and 8 illustrate two very important concepts in our study. First, it shows that the same general strategies used to infer user actions, languages, and message lengths can be used across many of the most popular messaging services regardless of their individual choices in data encoding, protocols, and encryption. Second, it is clear that WhatsApp and Viber provide even weaker protection against information leakage than iMessage, since there are exact one-to-one relationships between packet sizes and plaintext message lengths. Specifically, in Section 3.2.3, we mentioned that Apple iMessage data showed a “stair step" pattern due to the AES block sizes used, which naturally quantizes the output space and adds uncertainty to message length

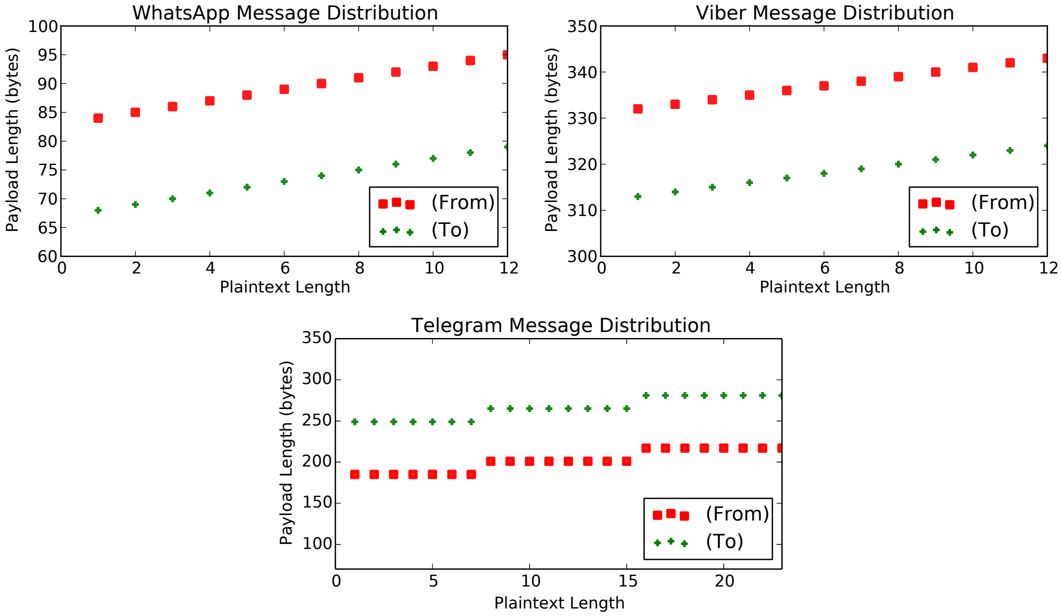


Figure 8: Scatterplot of plaintext message lengths versus payload lengths for WhatsApp, Viber, and Telegram.

predictions, while Viber and WhatsApp allow us to *exactly* predict message length. Telegram, with its use of end-to-end encryption technology, appears to be very similar to iMessage in terms of its payload length distributions. Therefore, we can expect the accuracy of the attacks will be at least as good as what was demonstrated on Apple iMessage traffic.

To mitigate against such privacy failures, it is possible to apply standard padding-based countermeasures. Apple iMessage and Telegram already implement a weak form of countermeasure through packet sizes quantized at AES block boundaries. A much more effective approach, however, would be to add random padding independently to each packet up to the maximum observed packet length for each service, thereby destroying any relationship to user actions. When implemented on our Apple iMessage data, the random padding methodology reduced all of our attacks to an accuracy of 0% at the cost of 613 bytes (328%) of overhead per message for iOS and 596 bytes (302%) for OSX. Although the absolute increase in size is rather small, we must consider that services like iMessage handle upwards of 2 billion messages every day, which translates to an additional terabyte of network traffic daily. For the more popular WhatsApp service, a similar increase would incur at least 4 terabytes of overhead. Other countermeasure methods, such as traffic morphing [127], may actually provide a more palatable trade-off between overhead and privacy, though further analysis is beyond the scope of this initial study.

4 Website Fingerprinting Countermeasures

Internet users rely on encrypted tunnels to keep their web browsing activities safe from eavesdroppers. A typical scenario involves a user establishing an encrypted tunnel to a proxy that then relays all subsequent HTTP traffic (in both directions) through the tunnel. The use of encryption should hide the contents of the traffic and, intuitively, the identity of the destination website(s). Yet modern encryption does not obfuscate the length of underlying plaintexts, nor the number of plaintexts that are encrypted. This information may seem harmless, but in fact it enables the range of *traffic analysis* (TA) attacks discussed in Section 2.

One commonly suggested TA countermeasure is to hide the plaintext length by adding padding prior to encryption. Padding countermeasures are standardized in TLS, explicitly to "frustrate attacks on a protocol that are based on analysis of the lengths of exchanged messages" [41]. Similar allowances for padding appear in SSH and IPSec. More advanced countermeasures, such as traffic morphing [128], manipulate whole streams of packets in order to precisely mimic the distribution of another website's packet lengths.

The seemingly widespread intuition behind these countermeasures is that they patch up the most dangerous side channel (packet lengths) and so provide good protection against TA attacks, including website identification. Existing literature might appear to support this intuition. For example, Liberatore and Levine [73] show that padding packets to the network MTU (e.g., 1500 bytes) reduces the accuracy of one of their attacks from 98% to 7%.

The results we present in this section challenge this intuition. We perform the first comprehensive analysis of low-level countermeasures (e.g., per-packet padding) for the kind of website identification attacks considered by prior work (c.f., [73, 127, 60, 92]): a closed-world setting for privacy sets, in which the *a priori* set of possible websites a user might visit is known to the attacker, coupled with the ability for the attacker to train and test on traffic traces that are free of real-world artifacts (e.g., caching effects, interleaved flows, and user-specific content). We consider nine distinct countermeasures, apply them to two large, independent datasets of website downloads, and pit the resulting obfuscated traffic against a total of seven different attacks. The results are summarized in Figure 5. What we uncover is surprisingly bleak:

None of the countermeasures are effective . We show that two classifiers —a new naïve Bayes classifier called VNG++ and a support vector machine classifier due to Panchenko et al. [92]— achieve better than 80% accuracy in identifying which of $k = 128$ websites was visited in a closed-world experiment. (Random guessing achieves 0.7% accuracy.) When $k = 2$ these classifiers achieve over 98% accuracy. This holds for all nine countermeasures considered, including ones inspired by the SSH, TLS and IPSec RFCs, and state-of-the-art ones such as traffic morphing [130].

Hiding packet lengths is not sufficient. We initiate a study of classifiers that do not directly use fine-grained features such as individual packet lengths. The VNG++ classifier just mentioned uses only "coarse" information, including overall time, total bandwidth, and size of bursts. In fact, we provide a naïve Bayes classifier that uses *only* the total bandwidth for training and testing, yet still achieves greater than 98% accuracy at $k = 2$ and 41% accuracy at $k = 128$. This implies that any effective countermeasure must produce outputs that consume indistinguishable amounts of bandwidth to be effective for the adversarial model we consider.

Attack	**Classifier**	**Features Considered**	$k = 128$	$k = 775$
Liberatore et al. [73] (LL)	NB	Packet lengths	25%	8%
Herrmann et al. [60] (H)	MNB	Packet lengths	3%	0%
Panchenko et al. [92] (P)	SVM	Packet lengths, Order Total bytes	82%	63%
Time (TIME)	NB	Total trace time	9%	3%
Bandwidth (BW)	NB	Upstream/Downstream total bytes	41%	18%
Variable n-gram (VNG)	NB	Bytes in traffic bursts	69%	54%
VNG++	NB	Total trace time, Upstream/Downstream total bytes, Bytes in traffic bursts	80%	61%

Table 5: Summary of attacks evaluated in our work. The **Classifier** column indicates the classifier used: naïve Bayes (NB), multinomial naïve Bayes (MNB) or support vector machine (SVM.) The **Features Considered** column indicates the features used by the classifier. The $k = 128$ and $k = 775$ columns indicate the classifier accuracy for a privacy set of size k.

Coarse information is unlikely to be hidden efficiently. Our coarse-feature attacks, in particular the bandwidth-only attack, strongly suggest that resource-efficient countermeasures will not (on their own) effectively hide website identity within a small privacy set. So, we investigate an inefficient strawman countermeasure, Buffered Fixed-Length Obfuscation (BuFLO, pronounced "buffalo"), that combines and makes concrete several previous suggestions: it sends packets of a fixed size at fixed intervals, using dummy packets to both fill in and (potentially) extend the transmission. We subject it to the same analysis as the other countermeasures. This analysis shows that should BuFLO fail to obfuscate total time duration and total bandwidth, then attacks *still* achieve 27% accuracy at $k = 128$. With a bandwidth overhead of over 400%, we can, in theory, finally reduce $k = 128$ accuracy to 5%.

Relevance to other settings. While the adversarial model that we consider is consistent with previous work, we admit that there are several factors (e.g., caching, open-world identification) that are not captured. Indeed, these may reduce the effectiveness of the attacks, and improve countermeasure efficacy, in practice. There may also be some other settings, such as Voice over IP (VoIP) traffic [130, 128, 129, 122], where the nature of the application-layer protocol enables some countermeasures to work very well. That said, the model considered in this section (and its predecessors) is one that a general-purpose countermeasure ought to cover.

Finally, our analysis does not cover application-layer countermeasures such as Camouflage [60] and HTTPOS [77], which both make intimate use of spurious HTTP requests to help obfuscate traffic patterns. We suspect, however, that the lessons learned here might help direct future analysis of application-layer countermeasures, as well.

Acknowledgements This work in this section was performed in collaboration with Coull, Ristenpart and Shrimpton. It was presented at IEEE Security and Privacy in 2012 [47]. The analysis and writing of the results was lead by Shrimpton and Ristenpart, with the help of Dyer and Coull. Dyer lead the engineering effort to reproduce prior results, and surfaced new attacks with the guidance of Shrimpton, Ristenpart, and Coull.

4.1 Experimental Methodology

Like previous works [73, 127, 60, 92], our experiments simulate a closed-world setting in which an adversary has access to the timing, lengths, and directionality of packets sent over an encrypted HTTP tunnel (e.g., to or from a proxy server). We assume secure encryption algorithms are used and no information can be learned from the encrypted contents itself.

We base our simulation on two datasets that have been widely used by previous works on web page identification. The Liberatore and Levine dataset [73] contains timestamped traces from 2,000 web pages. The Herrmann et al. [60] dataset contains timestamped traces from 775 web pages. A *trace* is defined as a record of the lengths and timings of ciphertexts generated by accessing a web page using an OpenSSH single-hop SOCKS proxy. Please refer to the previous works [73, 60] for further details about data collection methodology.

Each of our experiments is performed with respect to a particular classifier, a particular countermeasure, and a specified set of n web pages. An experiment consists of a number of trials; we will say in a moment how the particular number of trials is determined. At the start of each experimental trial, we uniformly select a subset of $k \leq n$ web pages to define the privacy set for that trial.[2] Next we establish k sets of 20 traces, one for each web page, as follows. For every web page in the data set, there are $m > 20$ chronologically sorted sample traces. We select a random trace index $i \in \{0, 1, \ldots, m - 19\}$, and take traces $i, i + 1, \ldots, i + 19$ for each of the k web pages. The first $t = 16$ of the traces from each of the k sets are used as the training data for the classifier, and the remaining $T = 4$ traces form the testing data set.[3] The countermeasure is applied to both the training and testing data, and the classifier is trained and then tested to determine its accuracy. Classifier accuracy is calculated as (c/Tk), where c is the number of correctly classified test traces and k is our privacy set size.

In each experiment, we perform $2^{(15-\log_2(k))}$ trials, so that there are a total of $T \cdot 2^{15}$ test data points per experiment. We consider values of $k \in \{2, 4, 8, 16, 32, 64, 128, 256, 512, 775\}$ in order to capture countermeasure performance across a number of scenarios. Intuitively, smaller values of k present easier classification (attack)

[2] We do not believe the uniform distribution represents typical user web-browsing behavior. In practice, we expect that biased sampling from the privacy set would further aid an attacker.

[3] We considered values of $t \in \{4, 8, 12, 16\}$ and observed effects consistent with those reported by Liberatore and Levine [73]: as t increases there was a consistent, modest increase in classification accuracy.

settings, and larger values of k present more difficult classifier settings.

We note that the engineering effort required to produce our results was substantial. To aid future research efforts, the Python framework used for our experiments is publicly available[4].

4.2 Traffic Classifiers

A sequence of works detail a variety of TA attacks, in the form of classifiers that attempt to identify the web page visited over an encrypted channel. These classifiers use *supervised* machine learning algorithms, meaning they are able to train on traces that are labeled with the destination website. Each algorithm has a *training* and a *testing* phase. During training, the algorithm is given a set $\{(X_1, \ell_1), (X_2, \ell_2), \ldots, (X_n, \ell_n)\}$, where each X_i is an *vector* of *features* and ℓ_i is a *label*. During testing the classification algorithm is given a vector Y and must return a label. In our case, a vector X_i contains information about the lengths, timings, and direction of packets in the encrypted connection containing a web page ℓ_i, and the format of a vector X_i is dependent upon the classifier. In the remainder of this section, we present a high-level overview of the operation of the three published classifiers that we use in our evaluation, and we refer interested readers to more detailed descriptions elsewhere [84, 73, 60, 92].

4.2.1 Liberatore and Levine Classifier

Liberatore and Levine [73] (LL) proposed the use of a naïve Bayes classifier (NB) to identify web pages using the direction and length of the packets. The naïve Bayes classifier determines the conditional probability $\Pr(\ell_i|Y)$ for a given vector of features Y using Bayes' rule: $\Pr(\ell_i|Y) = \frac{\Pr(Y|\ell_i)\Pr(\ell_i)}{\Pr(Y)}$. The probability is computed for all labels ℓ_i with $i = \{1, 2, \ldots, k\}$ and k representing the size of the privacy set (or

[4] http://www.kpdyer.com/

number of labels being considered), and the label with the highest probability is selected as the classifier's guess. The probability $\Pr(Y|\ell_i)$ is estimated using kernel density estimation over the example feature vector provided during training, and $\Pr(\ell_i)$ is assumed to be $1/k$. The feature vectors used by the LL classifier are derived from the count of the lengths of the packets sent in each direction of the encrypted connection. Specifically, the feature vector contains $2 \cdot 1449 = 2898$ integers that represent the number of packets seen in the given vector with each of the potential direction and packet length combinations (i.e., $\{\uparrow, \downarrow\} \times \{52, \ldots, 1500\}$). For example, if we observe a packet of length 1500 in the ↓ direction (e.g., server to client) we would increment the counter for (↓,1500).

4.2.2 Herrmann et al. Classifier

Herrmann, Wendolsky and Fedarrath [60] (H) take a similar approach to Liberatore and Levine, however they make use of a multinomial naïve Bayes (MNB) classifier. Like the naïve Bayes classifier with density estimation, the multinomial naïve Bayes classifier attempts to estimate the probability $\Pr(\ell_i|Y)$ for each of the $i = \{1, 2, \ldots, k\}$ potential labels and the given feature vector Y. The key difference is that the multinomial classifier does not apply density estimation techniques to determine the probability $\Pr(Y|\ell_i)$, but instead uses the aggregated frequency of the features (i.e., normalized distribution) across all training vectors. Thus, the H classifier uses normalized counts of (*direction*, *length*), whereas the LL classifier examined raw counts. Furthermore, Herrmann et al. suggest a number of approaches for normalizing these counts. For our evaluation, we combine *term frequency transformation* and *cosine normalization*, as these were identified by Herrmann et al. to be the most effective in the SSH setting.

4.2.3 Panchenko et al. Classifier

Panchenko et al. [92] (P) take a completely different approach by applying a support vector machine (SVM) classifier to the problem of identifying web pages. A support vector machine is a type of binary linear classifier that classifies points in a high-dimensional space by determining their relation to a separating hyperplane. In particular, the SVM is trained by providing labeled points and discovering a hyperplane that maximally separates the two classes of points. Classification occurs by determining where the point in question lies in relation to the splitting hyperplane. Due to the complexity of SVM classifiers, we forego a detailed discussion of their various parameters and options.

We configure our SVM as follows. We use the same radial basis function (RBF) kernel as Panchenko et al. with parameters of $C = 2^{17}$ and $\gamma = 2^{-19}$. The P classifier uses a wide variety of coarse and detailed features of the network data mostly derived from packet lengths and ordering. Some of these features include the total number of bytes transmitted, total number of packets transmitted, proportion of packets in each direction, and raw counts of packet lengths. There are also several features known as “markers" that delineate when information flow over the encrypted connection has changed direction. These markers aggregate bandwidth and number of packets into discrete chunks. Each of the features considered by the P classifier are rounded and all 52 byte TCP acknowledgement packets are removed to minimize noise and variance in the training and testing vectors.

4.3 Countermeasures

For ease of exposition and analysis, we organize the considered countermeasures into three categories: those that are inspired by the padding allowed within the SSH, TLS and IPSec standards (Type-1); other padding-based countermeasures (Type-2); and countermeasures that make explicit use of source and target packet-length distributions (Type-3). In what follows, we describe the operation of the countermeasures we evaluate and discuss the overhead they generate. Lengths are always measured in bytes.

4.3.1 Type-1: SSH/TLS/IPSec-Motivated Countermeasures

A common suggestion, already used in some implementations, like GnuTLS[5], is to obfuscate plaintext lengths by choosing random amounts of extra padding to append to the plaintext prior to encryption. SSH, TLS and IPSec allow up to 255 bytes of padding in order to align the to-be-encrypted plaintext with the underlying block cipher boundary, and also to provide some obfuscation of the original plaintext length. We consider two ways in which this might be implemented within SSH/TLS/IPSec: (1) choose a single random amount of padding to be applied across all plaintexts in the session, or (2) choose a random amount of padding for each plaintext.

Session Random 255 padding: A uniform value $r \in \{0, 8, 16 \ldots, 248\}$ is sampled and stored for the session.[6] Each packet in the trace has its length field increased by r, up to a maximum of the MTU.

Packet Random 255 padding: Same as Session Random 255 padding, except that a new random padding length r is sampled for each input packet.

We note that our simulation of Session Random and Packet Random padding in this setting are not exactly what would be implemented in reality because we do not have access to the size of the plaintext data from the datasets available to us.

[5] `http://www.gnu.org/software/gnutls/`

[6] We assume that the underlying encryption block size is 8 bytes. For the Liberatore and Levine dataset, we know this assumption is true. We do not expect classification accuracies to be different if, in fact, the block size was 16 bytes.

Instead, our assumption is that the plaintext data is sufficiently short to fit into a single TCP packet and therefore is closely approximated by simply adding the padding to the length of the ciphertext. What we simulate, therefore, is likely to overstate the efficacy of the countermeasure since the (at most) 255 bytes of padding would be dominated by the true size of the plaintext (e.g., up to 2^{14} bytes for TLS), thereby providing relatively little noise. In contrast, our simulation allows for a much larger ratio of plaintext to padding, which in turn adds significantly more noise.

4.3.2 Type-2: Other Padding-based Countermeasures

The second class of countermeasure we consider are those padding mechanisms that are not easily supported in existing encrypted network protocol standards due to the amount of padding added. In this scenario, we assume the countermeasure will be capable of managing fragmentation and padding of the data before calling the encryption scheme. Most of the countermeasures considered by prior work fall into this category, though we also consider a randomized scheme that has not been previously explored.

Linear padding: All packet lengths are increased to the nearest multiple of 128, or the MTU, whichever is smaller.

Exponential padding: All packet lengths are increased to the nearest power of two, or the MTU, whichever is smaller.

Mice-Elephants padding: If the packet length is ≤ 128, then the packet is increased to 128 bytes; otherwise it is padded to the MTU.

Pad to MTU: All packet lengths are increased to the MTU.

Packet Random MTU padding: Let $\mathtt{M}$ be the MTU and ℓ be the input packet length.

For each packet, a value $r \in \{0, 8, 16, \ldots, \texttt{M} - \ell\}$ is sampled uniformly at random and the packet length is increased by r.

4.3.3 Type-3: Distribution-based Countermeasures

Wright et al. [127] presented two novel suggestions as improvements upon traditional per-packet padding countermeasures: *direct target sampling* (DTS) and *traffic morphing* (TM). On the surface, both techniques have the same objective. That is, they augment a protocol's packets by chopping and padding such that the augmented packets appear to come from a pre-defined target distribution (i.e., a different web page). Ideally, DTS and TM have security benefits over traditional per-packet padding strategies because they do not preserve the underlying protocol's number of packets transmitted nor packet lengths. Although the full implementations details of DTS and TM are beyond scope of this section (see [127]), we give a high-level overview here.

Direct target sampling: Given a pair of web pages A and B, where A is the source and B is the target, we can derive a probability distribution over their respective packet lengths, D_A and D_B. When a packet of length i is produced for web page A, we sample from the packet length distribution D_B to get a new length i'. If $i' > i$, we pad the packet from A to length i' and send the padded packet. Otherwise, we send i' bytes of the original packet and continue sampling from D_B until all bytes of the original packet have been sent. Wright et al. left unspecified morphing with respect to packet timings. We assume a negligible overhead to perform morphing and specify a 10ms inter-packet delay for dummy packets.

In our experiments, we select the target distribution uniformly at random from our set of k potential identities. The selected web page remains unchanged (i.e., no countermeasures applied), while the remaining $k - 1$ web pages are altered to look

	Overhead (%)	
Countermeasure	LL	H
Session Random 255	9.0	7.1
Packet Random 255	9.0	7.1
Linear	4.2	3.4
Exponential	8.7	10.3
Mice-Elephants	41.6	39.3
Pad to MTU	81.2	58.1
Packet Random MTU	40.1	28.8
Direct Target Sampling	86.4	66.5
Traffic Morphing	60.8	49.8

Table 6: Bandwidth overhead of evaluated countermeasures calculated on Liberatore and Levine (LL) and Herrmann et al. (H) datasets.

like it. After the source web page has stopped sending packets, the direct target sampling countermeasure continues to send packets sampled from D_B until the L1 distance between the distribution of sent packet lengths and D_B is less than 0.3.

Traffic morphing: Traffic morphing operates similarly to direct target sampling except that instead of sampling from the target distribution directly, we use convex optimization methods to produce a morphing matrix that ensures we make the source distribution look like the target while simultaneously minimizing overhead. Each column in the matrix is associated with one of the packet lengths in the source distribution, and that column defines the target distribution to sample from when that source packet length is encountered. As an example, if we receive a source packet of length i, we find the associated column in the matrix and sample from its distribution to find an output length i'. One matrix is made for all ordered pairs of source and target web pages (A, B). The process of padding and splitting packets occurs exactly as in the direct target sampling case. Like the direct target sampling method, once the source web page stops sending packets, dummy packets are sampled directly from D_B until the L1 distance between the distribution of sent packet lengths and D_B is less than 0.3. In our simulations we select a target distribution using the

		LL	H	P
$k = 2$	Type-1	85%	71%	99%
	Type-2	97%	80%	99%
	Type-3	98%	76%	99%
$k = 128$	Type-1	41%	13%	91%
	Type-2	46%	5%	90%
	Type-3	25%	3%	82%

Table 7: The lowest average accuracy for each countermeasure class against LL, H, and P classifiers using the Hermann dataset. Random guessing yields 50% ($k = 2$) or 0.7% ($k = 128$) accuracy.

same strategy described for DTS.

4.3.4 Overhead

Although the focus of our evaluation lies in understanding the security provided by these countermeasures, we realize that their cost in terms of bandwidth overhead and latency is an important factor that determines whether they are applicable in practice or not. To this end, we present the bandwidth overhead induced by the countermeasures for both the Liberatore and Levine and Herrmann et al. datasets in Figure 6. Overhead is calculated as (*bytes sent with countermeasure*)/(*bytes sent without countermeasure*) times 100. We note that these overhead measurements differ from those of earlier work because we do not apply countermeasures to TCP acknowledgement (52-byte) packets. For example, Liberatore and Levine [73] report a Pad to MTU overhead of 145% and Wright et al. [127] report 156%. We argue that acknowledgement packets are present regardless of the content being downloaded and there is no standard mechanism for application-layer countermeasures to apply padding to TCP acknowledgement (52-byte) packets. Nevertheless, as we will see in the following section, there is almost no correlation between overhead and the level of confidentiality provided by the countermeasure.

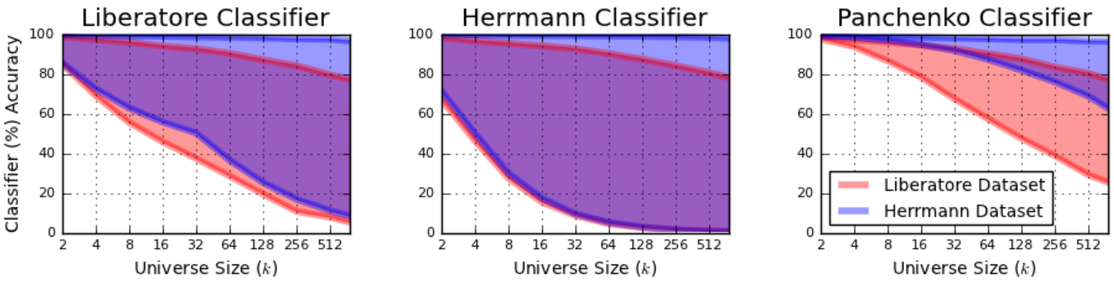


Figure 9: Comparison of accuracy silhouettes for the Liberatore and Levine and Herrmann datasets across all countermeasures for the LL, H, and P classifiers, respectively.

4.4 Existing Countermeasures versus Existing Classifiers

We pit the LL, H, and P classifiers from Section 4.2 against traffic simulated as per the nine countermeasures of the previous section. The testing methodology used was described in Section 4.1. We also look at classifiability of the raw traffic, meaning when no countermeasure (beyond the normal SSH encryption) is applied.

We note that despite the appearance of the LL, H, and P classifiers in the literature, all the results we report are new. In particular, the H and P classifiers were never tested against any of these countermeasures, while the LL classifier did look at efficacy against Linear, Exponential, Mice-Elephants, and Pad to MTU but only at $k = 1000$. Figure 7 contains a high-level summary for $k = 2$ and $k = 128$. We refer the interested reader to Appendix B for comprehensive results.

In the rest of this section we analyze the results from various points of view, including the role of the dataset, the relative performance of the classifiers, and the relative performance of the different countermeasures.

4.4.1 Comparing the Datasets

Before beginning digging into the results in earnest, we first evaluate the consistency and quality of the two available datasets. We do so to determine the extent to which results gathered using them represent the identifiability of the web pages rather than artifacts of the collection process, such as connection timeouts and general collection failures. In Figure 9, we show the silhouette of the accuracy achieved by the three classifiers across a number of universe sizes and countermeasures using each of the datasets. That is, the lower boundary of each silhouette is the best-performing countermeasure while the upper boundary represents the worst-performing (which turned out to always be no countermeasure, as one would expect).

Ideally, the classifier accuracies should be roughly similar, or at least show similar trends. Instead, what we notice is a trend toward strong drops in performance as the web page universe size increases in the Liberatore dataset, whereas in the Herrmann dataset we see a much smoother drop across multiple universe sizes and across all classifiers. This is most notable under the P classifier (far right of Figure 9).

To take a closer look at the differences between the datasets, we report some basic statistics in Figure 8. The fraction of traces that have short duration, particularly ones that are clearly degenerate (≤ 10 packets), is much higher in the Liberatore dataset. Such degenerate traces act as noise that leads to classification errors. We suspect that they arise in the dataset due to collection errors (e.g., incomplete website visits), and may imply that some previous works [73, 127] may underestimate the privacy threat posed by web page traffic analysis attacks. Despite the extra noise, the classifiers performed well, just consistently lower at high values of k as compared to the Herrmann dataset. In addition, the Herrmann dataset was collected in 2009, as opposed to the Liberatore dataset, which was collected in 2006. Despite all these differences we found the high-level trends and conclusions are the same across both datasets. For these reasons, we will focus our analysis only on the Herrmann dataset for the remainder of this section. Appendix B contains details for classifier performance using the Liberatore dataset at $k = 128$.

	LL	H
Traces with 0 packets in one direction	3.1%	0.1%
Traces with $\leq$ 5 bidirectional packets	5.2%	0.2%
Traces with $\leq$ 10 bidirectional packets	13.8%	0.4%
Traces with $\leq$ 1s duration	29.4%	6.4%
Median trace duration	2.4 sec.	3.6 sec.
Median bidirectional packet count	106	256
Median bandwidth utilization (bytes)	78,382	235,687

Table 8: Statistics illustrating the presence of degenerate or erroneous traces in the Liberatore and Levine and Hermann datasets.

4.4.2 Comparison of Classifiers

Figure 10 gives a three-by-three grid of graphs: one column per classifier and one row for countermeasure type. We start by quickly comparing the relative performance of the three classifiers, which is observable by comparing the performance across the three columns.

The first thing to notice is that at $k = 2$, essentially all of the classifiers do well against all of the countermeasures. The LL and P classifiers are particularly strong, even against the DTS and TM countermeasures. The overall best classifier is clearly the P classifier. It is robust to all the countermeasures. The H classifier edges out both the P and LL classifiers for raw traffic, but is very fragile in the face of all but the simplest countermeasure (Linear padding). The LL classifier proves more robust than the H classifier, but has more severe accuracy degradation compared to P as k increases.

4.4.3 Comparison of Countermeasures

Consider the first row of Figure 10, where we see a comparison of the two Type-1 randomized padding schemes. Curiously, it is better to pick a single random padding amount to apply to each packet within a trace than to pick fresh random amounts per

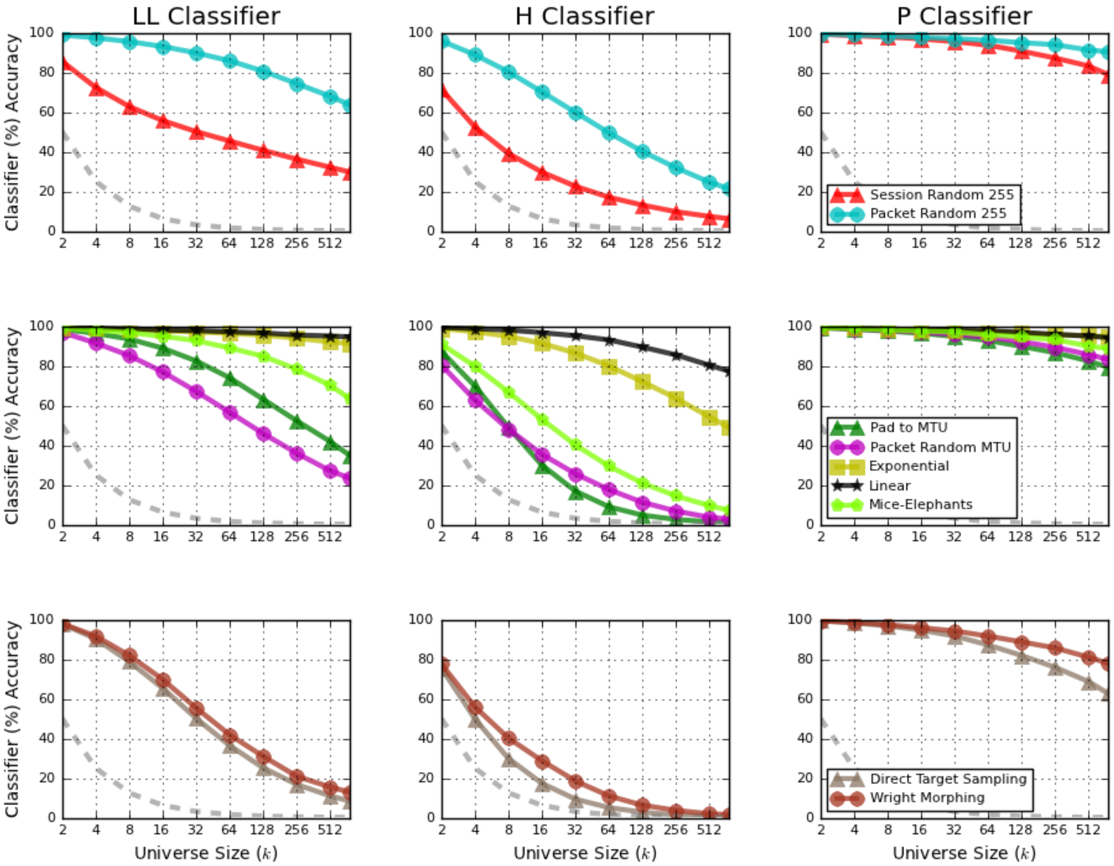


Figure 10: Average accuracy as k varies for the LL (left column), H (middle column), and P (right column) classifiers with respect to the Type-1 (top row), Type-2 (middle row), and Type-3 (bottom row) countermeasures. The dotted gray line in each graph represents a random-guess adversary.

packet. Applying a single random amount across all packets shifts the distribution of packet lengths in a way that is unlikely to have been seen during training. On the other hand, randomizing per packet "averages out" during training and testing.

Common intuition about the Pad to MTU countermeasure is that it ought to work well against TA attacks since it ensures that no individual packet length information is leaked. However, as we seen in the second row of Figure 10, we see this intuition is wrong in large part because the *number* of packets is still leaked. The LL classifier, for example, exploits this fact, since it trains on the number of packets of each (*direction*, *length*). When the packets are padded to the MTU, there are only two numbers,

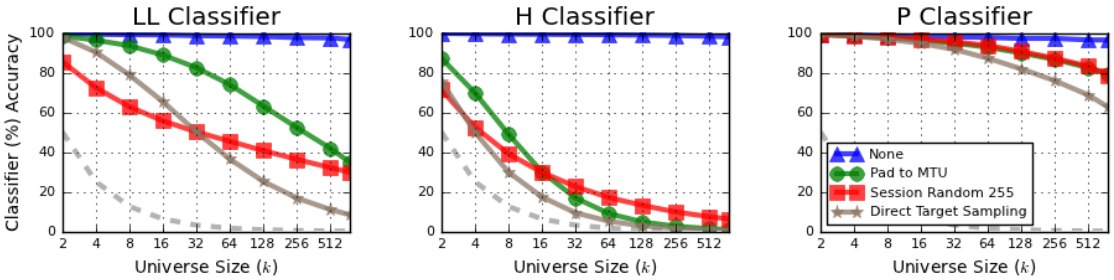


Figure 11: Comparison of the overall best performing countermeasure of each type against the LL, H, and P classifiers.

namely for (↑,1500) and (↓,1500). The LL classifier does well because the number of packets transmitted is relatively consistent across traces for a particular web page. (We will investigate this more in the next section.) This also is our first evidence that exact packet-length information is *not* necessary for high-accuracy classification.

Next, we turn to the Type-3 countermeasures. Recall that these countermeasures focus on altering a specific feature of the web page traffic, namely the distribution of normalized counts, so that one web page looks like another with respect to that feature. In theory then, the distribution of packets produced by the DTS and TM countermeasures should match that of the target web page and, unlike Type-1 and Type-2 countermeasures, the number of packets from the source web page should be concealed, in part. This is not true in all cases, however, as Type-3 countermeasures do not substantially change the total bandwidth of data transmitted in each direction, nor the duration of the trace with regards to time. In fact, no countermeasure considered here substantially changes the total bandwidth. Moreover, these countermeasures do not hide "burstiness" of the data, which may be correlated to higher level structure of the underlying HTTP traffic (e.g., a downstream burst represents a web page object). Therefore, DTS and TM perform best against the H classifier, which examines the same normalized packet count distribution, while the P classifier performs particularly well with its use of packet burst information.

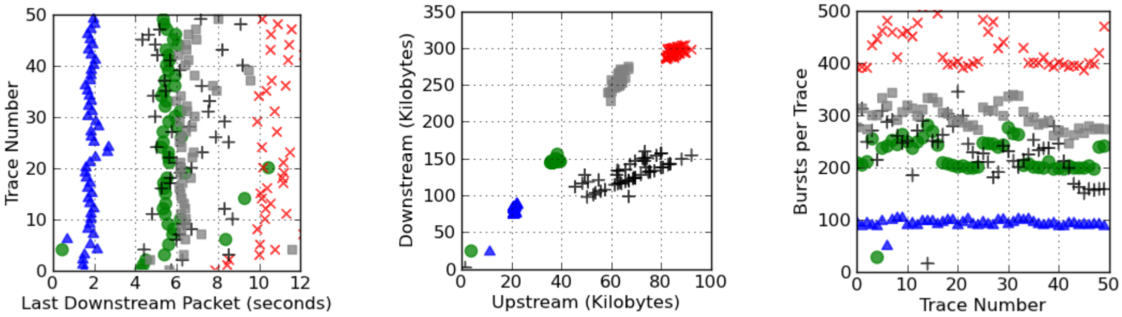


Figure 12: Each scatterplot is a visual representation of the first fifty traces, from the first five websites in the Herrmann dataset. Each symbol of the same shape and color represents the same web page. (left) Distribution of traces with respect to duration in seconds. (middle) Distribution of traces with respect to bandwidth utilization, where we distinguish the upstream and downstream directions. (right) Distribution of traces with respect to the number of bursts per trace.

We compare the best countermeasure from each type in Figure 11: Session Random 255 (Type-1), Pad to MTU (Type-2), and DTS (Type-3). A few surprises arise in this comparison. First, Session Random 255 performs better or about the same as Pad to MTU. This is surprising, as Session Random 255 is a significantly lighter-weight countermeasure. It has only 7% overhead compared to Pad to MTU's 58%, and can potentially be dropped into existing deployments of SSH and TLS. That said, even at $k = 128$, it is unlikely to be satisfying to drop accuracy only down to 90%. DTS does better than the others across all values of k against the best classifier (P), but we note that simpler countermeasures actually can do a better job against the LL and H classifiers for lower k values.

4.5 Exploring Coarse Features

Our study of existing classifiers reveals that some fine-grained features, such as individual packet lengths, are *not* required for high-accuracy classification. Indeed, the fact that the P classifier performs so well against the Pad to MTU countermeasure means that it is using features other than individual packet lengths to determine classification. This leads us to the following question: Are coarse traffic features sufficient for high-accuracy classification?

To answer this question, we explore three coarse features: total transmission time, total per-direction bandwidth, and traffic "burstiness".[7] From these features we build the time (TIME), bandwidth (BW), and the variable n-gram (VNG) classifier using naïve Bayes as our underlying machine learning algorithm. See Figure 13 for a visual summary of their performance. Later, we put these three coarse features together, and build the VNG++ naïve Bayes classifier. We will see that VNG++ is just as accurate as the (more complex) P classifier.

4.5.1 Total Time

We begin with the most coarse and intuitively least useful feature, the total timespan of a trace. How much do traces differ based on total time? The left-most plot in Figure 12 depicts the time of the first 50 traces from five websites in the Herrmann dataset. There is clear regularity within traces from each website, suggesting relatively low variance for this feature.

To test the usefulness of total time in classification, we implemented a naïve Bayes classifier that uses time as its only feature. This simple time-only classifier is quite successful for small k, as shown in Figure 13. At $k = 2$, it is able to achieve better than an 80% average accuracy against the three best countermeasures from each class as determined by performance on the P classifier. As the privacy set increases, the likelihood of multiple websites having similar timing increases, and so the accuracy of the time classifier goes down. At $k = 775$, it achieves only about 3% accuracy, although this is still substantially better than random guessing (0.1%) and may provide value as a supplementary feature in order to increase a classifier's accuracy.

[7] We note that these features are more coarse than individual packet lengths, in the sense that knowing the latter likely implies knowing the former, but not the other way around.

Figure 13 also shows that the time classifier performs roughly the same against raw traffic (i.e., the "None" countermeasure) and with traffic countermeasures applied. As one might expect padding-based countermeasures (Type-1 and Type-2), do not directly modify the total time taken by traces. On the other hand, distribution-based countermeasures (Type-3) potentially inject dummy packets into a trace, but this is most often no more than 10-12 packets sent in quick succession. Thus, these also do not change the total time significantly.

4.5.2 Total Per-Direction Bandwidth

Next, we turn to total bandwidth consumed per direction. We see the consistency of total bandwidth in the center plot in Figure 12, which displays the upstream and downstream bandwidths of the first 50 traces of five websites from the Herrmann dataset. This plot shows a clear clustering of the websites with both very low variance within website clusters and high degrees of separability (i.e., spacing) between clusters.

Therefore, we expect bandwidth-based classification will work well as long as websites within the privacy set do not have too much overlap in terms of total per-direction bandwidth. Figure 13 shows that, indeed, the bandwidth classifier performs well. In fact, the real surprise is just *how* well the bandwidth-only classifier works for all privacy set sizes despite the coarse nature of the feature. At $k = 2$, the classifier provides close to perfect accuracy of over 99% against all countermeasures. Moreover, compare the behavior of the bandwidth-only classifier to that of the LL and H classifiers (c.f., Figure 11), which do not use bandwidth as a feature, as k increases. The bandwidth classifier is clearly more robust to changes in privacy set size. This might seem surprising, since countermeasures such as Pad to MTU and Session Random 255 should, intuitively, obfuscate bandwidth usage. They do, but these per-packet

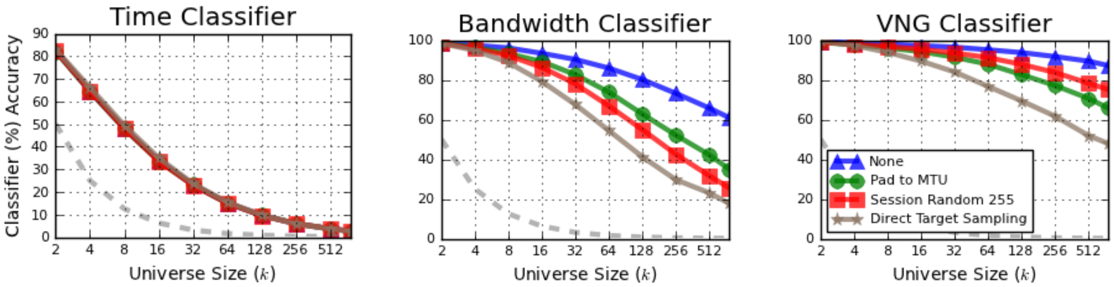


Figure 13: The average accuracy against the raw encrypted traffic (None), and the best countermeasures from each type, as established in Section 4.4. (left) the time-only classifier. (middle) the bandwidth only classifier. (right) the VNG ("burstiness") classifier.

paddings only add noise to the low order bits of total bandwidth. Specifically, the change to bandwidth usage is too small relative to what would be needed to make two websites' bandwidths likely to overlap significantly. This is true for all of the padding-based countermeasures (Type-1 and Type-2). Distribution-based countermeasures DTS and TM, however, offer the best resistance to the bandwidth classifier for higher k values. Here, they outpace other countermeasures by several percentage points. This seems to be due to the insertion of dummy packets, which can add more noise than per-packet padding for total bandwidth use.

4.6 Variable n-gram

The time and bandwidth features already provide impressive classification ability despite their coarse nature, but do not yet give the accuracy that the Panchenko classifier achieves. We therefore look at a third feature, that of burst bandwidth. A burst is a sequence of non-acknowledgement packets sent in one direction that lie between two packets sent in the opposite direction. The bandwidth of a burst is the total size of all packets contained in the burst, in bytes. For instance, if we have a trace of the form

$$(\uparrow, 100), (\downarrow, 1500), (\downarrow, 100), (\uparrow, 200), (\uparrow, 300)$$

	Classifier		
Countermeasure	P	P-NB	VNG++
None	97.2 ± 0.2	98.2 ± 0.9	93.9 ± 0.3
Session Random 255	90.6 ± 0.3	59.1 ± 2.3	91.6 ± 0.3
Packet Random 255	94.9 ± 0.3	93.7 ± 1.6	93.5 ± 0.3
Linear	96.8 ± 0.2	96.9 ± 1.1	94.3 ± 0.3
Exponential	96.6 ± 0.3	97.4 ± 0.9	94.8 ± 0.3
Mice-Elephants	94.5 ± 0.6	95.1 ± 0.8	91.7 ± 0.4
Pad to MTU	89.8 ± 0.4	91.7 ± 1.5	88.2 ± 0.4
Packet Random MTU	92.1 ± 0.3	84.1 ± 1.7	87.6 ± 0.3
Direct Target Sampling	81.8 ± 0.5	76.8 ± 2.5	80.2 ± 0.5
Traffic Morphing	88.7 ± 0.4	82.6 ± 5.6	85.6 ± 0.7

Table 9: Accuracies (%) of P, P-NB, and VNG++ classifiers at $k = 128$.

then there are three bursts with bandwidth 100, 1600, and 500. The intuition underlying this is that bursts correlate with higher-level properties of the traffic, such as individual web requests. This observation was first made by Panchenko et al. [92].

The right-most plot in Figure 12 shows the number of bursts for each of the first 50 traces for five websites in the Herrmann dataset. Even the number of bursts correlates strongly with the web page visited. Although this relatively limited information is capable of providing some classification ability, it turns out that burst bandwidths prove even more powerful.

Recalling that an n-gram model would coalesce n packets together into one feature, we can view bandwidth bursts as a variable n-gram model in which n varies across the trace. Then, our VNG (Variable n-Gram) classifier partitions a trace into bursts, coalesces packets into variable n-grams described by (*direction*, *size*) pairs, rounds the resulting sizes up to the nearest multiple of 600 bytes[8], and then applies a naïve Bayes classifier. Figure 13 shows how well the VNG classifier performs, already achieving better than 80% accuracy for all padding-based countermeasures, and

[8] Panchenko et al. experimentally determine this rounding value as a way to maximize classification accuracy via dimensionality reduction.

achieving significantly higher accuracy levels for distribution-based approaches than any other classifier except the P classifier.

4.6.1 Combining Coarse Features: the VNG++ Classifier

To extract all potential identifying information from these coarse features, we combine the time, bandwidth, and variable n-gram classifiers to give a simple, yet impressively effective, classifier that dispenses with use of individual packet lengths for classification. Specifically, we use total time, bandwidth in each direction of the connection, and variable n-grams as features of a naïve Bayes classifier. A graph of the VNG++ classifier's accuracy as k varies is given in Figure 14.

In comparing VNG++ to the P classifier, we note that the latter uses a large assortment of features (as discussed in Section 4.2), including fine-grained ones such as frequency of individual packet lengths. It also applies a more complicated machine learning algorithm in the form of an SVM. Figure 14 depicts the performance of the P and VNG++ classifiers against the best performing countermeasures of each type, as well as data with no countermeasure applied. Note that for clarity the y-axis starts at 50%, unlike other graphs. From this figure, two clear trends arise. First, VNG++'s performance against no countermeasure degrades slightly faster with k than the P classifier. This highlights that fine-grained features can provide some small benefit in classifying unprotected traces. Second, when we consider countermeasures, VNG++ matches P in performance. This holds despite the use of fewer features and the simpler machine learning algorithm used by the former. As it turns out, in the face of countermeasures, the coarse features are the damaging ones and fine-grained features are not particularly helpful.

A final question lingers: does using an SVM provide any advantage over a naïve Bayes classifier? We implemented a naïve Bayes version of the P classifier. This

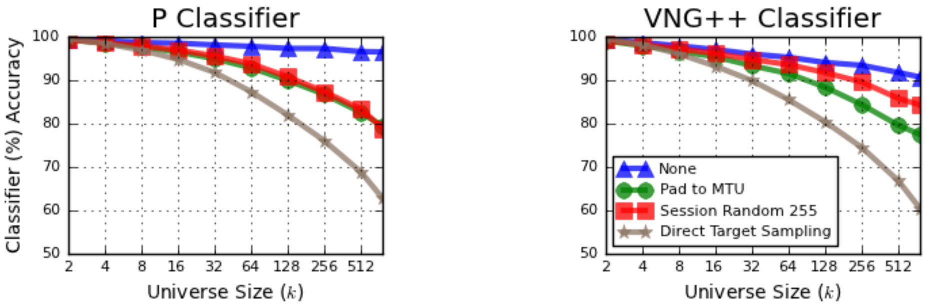


Figure 14: Accuracy of P (left) and VNG++ (right) classifiers against the best-performing countermeasures from Section 4.2.

P-NB classifier uses a 1-1 mapping of the features used by P to analogues suitable for use with a naïve Bayes classifier. A comparison of performance at $k = 128$ for P, P-NB, and VNG++ are given in Figure 9. Overall, we see that the results are consistent across all three classifiers. A single exception is the accuracy of P-NB for Session Random 255, which results in a surprisingly low classifier accuracy.

4.6.2 Discussion

The nine countermeasures considered so far attempt to obfuscate leaked features of the traffic via padding and insertion of dummy packets. As we've seen, however, these fail to protect significant amounts of identifying information from being leaked from coarse features of the encrypted traffic, rather than the fine-grained, per-packet features typically targeted by TA countermeasures. Unfortunately, these kinds of features are precisely the ones that are most difficult to efficiently hide.

Obfuscating total bandwidth is an obvious case in point. To prevent this feature from leaking information, a countermeasure must ensure a similar amount of bandwidth use across all websites in any given privacy set. Since we do not want to forego functionality (e.g., shutting down connections prematurely), this translates into a countermeasure that inserts dummy traffic until we achieve a total bandwidth close to that of the maximum bandwidth usage of any website in the privacy set.

Hiding burst bandwidth is also problematic. As seen in Figure 12, different websites can have quite different patterns of bursts. A countermeasure must smooth out these patterns. In theory, a traffic morphing-like countermeasure can attempt to imitate a target trace's burst patterns, however this will require buffering packets for potentially long periods of time. Thus, countermeasures for preventing website traffic analysis must incur both bandwidth and latency overheads.

In all, our analyses leaves little wiggle room for countermeasures to operate within. Providing robust protection against fingerprinting attacks for arbitrary websites in a closed-world setting, such as the one presented here, is going to have to be inefficient.

4.7 BuFLO: Buffered Fixed-Length Obfuscator

Our analysis thus far leaves us with the conclusion that, despite the long line of work on TA attacks and countermeasures, we have no packet-oriented countermeasure that prevents website fingerprinting attacks. We therefore want to know whether any measure can work, even prohibitively inefficient ones.

Following the analysis of the last section, we know that any effective countermeasure must hide the total time, bandwidth use, and burst patterns. To that end, we consider a new countermeasure **Bu***ffered* **F***ixed*-**L***ength* **O***bfuscator*, or BuFLO. It is a realization of the "fool-proof" folklore countermeasure that, intuitively, should defeat any TA classifier by removing all side-channel information. BuFLO operate by sending fixed-length packets at a fixed interval for at least a fixed amount of time. If a flow goes longer than the fixed time out, BuFLO lets it conclude while still using fixed-length packets at a fixed interval. In an ideal implementation, BuFLO will not leak packet lengths or packet timings, and so BuFLO should do a good job at closing side-channels that enable TA classifiers. This type of countermeasure has been investigated in the context of other TA attacks, such as those on anonymity networks [118, 140]

Our simulation-based analysis of BuFLO provides some positive evidence for packet-level countermeasures, but in fact our results here are mostly negative, thereby reinforcing the lessons learned in prior sections. BuFLO is, as one might expect, incredibly inefficient. Moreover, we will see that even mild attempts to claw back some efficiency can fail: setting the minimum session too aggressively short opens up vulnerability to our coarse-feature classifiers.

Finally, we note that our BuFLO countermeasure has been explored further in a number of follow on works [22, 90, 23], after the publication [47] of these initial results.

4.7.1 BuFLO **Description**

A BuFLO implementation is governed by three integer parameters d, ρ and τ:

- Parameter d determines the size of our fixed-length packets.
- Parameter ρ determines the rate or frequency (in milliseconds) at which we send packets.
- Parameter τ determines the minimum amount of time (in milliseconds) for which we must send packets.

A BuFLO implementation at the start of communications will send a packet of length d every ρ milliseconds until communications cease and at least τ milliseconds of time have elapsed. Specifically, data is buffered into discrete chunks, and these chunks are sent as quickly as possible via the steady flow of the fixed-length packets. When no data is in the buffer, dummy data is sent instead. This assumes that the application-layer signals the start and end of communication. Alternatively, we could have chosen τ as an *upper bound* on the duration of our communications session and forcibly close the connection even if communications are still in progress. This would disable any websites that take longer to load, making it unlikely to be a pragmatic choice.

4.7.2 Experiments

In this section, we examine BuFLO for various parameters using the Hermann dataset and provide detailed results in Appendix A. Since we are using a simulation-based experiment, these results reflect an ideal implementation that assumes the feasibility of implementing fixed packet timing intervals. This is at the very least difficult in practice [52] and clearly impossible for some values of ρ. Simulation also ignores the complexities of cross-layer communication in the network stack, and the ability for the BuFLO implementation to recognize the beginning and end of a data flow. If BuFLO cannot work in this setting, then it is unlikely to work elsewhere, aiding us in our exploration of the goal of understanding the limits of packet-level countermeasures.

We evaluated BuFLO empirically with parameters in the ranges of $\tau \in \{0, 10000\}$, $\rho \in \{20, 40\}$ and $d \in \{1000, 1500\}$. The least bandwidth-intensive configuration, at $\tau = 0$, $\rho = 40$ and $d = 1000$ would require at least 0.2 Mbps of continuous synchronous client-server bandwidth to operate[9]. Surprisingly, with this BuFLO configuration and a privacy set size of $k = 128$, the P classifier still identifies sites with an average accuracy of 27.3%. This is compared to 97.5% average accuracy with no countermeasure applied. At the other extreme of our experiments with $\tau = 10000$, $\rho = 20$ and $d = 1500$ it would require at least 0.6 Mbps of synchronous client-server bandwidth to operate. Here, the P classifier can *still* identify sites with a privacy set size of $k = 128$ with an average 5.1% accuracy.

[9] Calculated by $\left(\frac{1000}{\rho}\right) \cdot \left(\frac{8d}{10^6}\right)$.

4.7.3 Observations about BuFLO

BuFLO cannot leak packet lengths, nor can it leak packet timings. Yet, our experiments indicate that an aggressively configured BuFLO implementation can still leak information about transmitted contents. This is possible because BuFLO can leak *total bytes transmitted* and the *time required to transmit a trace* in two circumstances:

- The data source continued to produce data beyond the threshold τ.
- The data source ceases to produce data by the threshold τ, but there is still data in the buffer at time τ.

The first situation can occur if our our threshold τ is not sufficiently large to accommodate for all web pages that we may visit. The latter situation occurs when values ρ and d are not sufficiently configured to handle our application's data throughput, such that we transmit all data by time τ.

What is more, in some circumstances an inappropriately configured BuFLO implementation can can actually benefit an adversary. At $k = 128$ with $\tau = 0$, $\rho = 40$ and $d = 1000$ (see Figure 18) the BuFLO countermeasure can increase the accuracy of the Time classifier from 9.9% to 27.3%! In retrospect this is not surprising. If we throttle the bandwidth of the web page transfer, we will amplify its timing fingerprint.

These results reinforce the observations of prior sections. Namely, that TA countermeasures must, in the context of website identification, prevent coarse features from being leaked. As soon as these features leak, adversaries will gain some advantage in picking out web pages.

4.8 Concluding Discussion

Although a significant amount of previous work has investigated the topic of TA countermeasures, and specifically the case of preventing website identification attacks, the results were largely incomparable due to differing experimental methodology and datasets. Our work synthesizes and expands upon previous ones, and it provides sharper answers to some of the area's central questions:

Do TA countermeasures prevent website fingerprinting? None of the nine countermeasures considered here prevents the kind of website fingerprinting attack addressed by prior works [73, 127, 60, 92]. From a security perspective this setting is conservative, and makes several simplifying assumptions. (The attacker knows the privacy set; it trains and tests on traffic generated in the same way; the collected traffic does not account for (potentially) confounding effects, such as browser caching, interleaved web requests, etc.) Nevertheless, our negative results suggest that one should not rely *solely* upon these countermeasures to prevent website fingerprinting attacks.

Do TA attacks require individual packet lengths? No. We implemented three coarse-feature classifiers: one using only total time as a feature, one using only total per-direction bandwidth, and one tracking only data bursts (the VNG classifier). These did not make direct use of individual packet lengths or packet counts as features, yet attained high accuracy against the countermeasures. This highlights the point that masking fine-grained information is insufficient, unless such masking also hides telling large-scale features (e.g., individual object requests, size of web objects, etc.).

Does classification engine matter? Our experiments suggest it is the *features*, and not the underlying classification engine, that matters. We implemented a naïve Bayes-based classifier that used the same features as those exploited by the SVM-based Panchenko et al. classifier, and our experiments show that these two perform almost identically.

Does the privacy-set size (k) matter? For the considered setting, it seems not to matter much. When no countermeasure is used, attacks can achieve roughly the same accuracy for $k = 2$ through $k = 775$. When countermeasures are applied, the best classifier's accuracy does drop slowly as k increases. This suggests that the countermeasures do obfuscate some features that can improve accuracy. That said, at the largest k, the best classifiers offer better than 60% accuracy against all of the countermeasures.

Our work paints a pretty negative picture of the usefulness of efficient, low-level TA countermeasures against website-fingerprinting attacks. But pessimism need not prevail. Future work could investigate more detailed modelings of real-world traffic, and investigate applications of TA countermeasures beyond website fingerprinting. This may uncover settings in which some countermeasures are more successful than they were in our experiments. In addition, the coarse features (e.g. bandwidth) that appear near impossible to obfuscate efficiently at the level of individual packets might be better handled at the application layer. Previous works [60, 77] suggest application-layer countermeasures with promising initial evaluations. Future work could provide more extensive investigation of such countermeasures.

5 Overview: Internet Censorship and Censorship Circumvention

Network operators increasingly deploy deep packet inspection (DPI) to improve visibility into network activities and control those activities based on application-layer content. In practice, the most advanced DPI systems use a combination of simple pattern matching and regular expressions. Regular expressions encode fingerprints for, in particular, the protocols of interest, and whether packet contents match against these expressions informs what is called port-independent protocol identification [34, 94, 68, 9, 40]. Performance and scalability of the regex-based traffic classification strategies has been extensively studied [58, 112, 13].

Alternative protocol identification strategies have been explored and include using packet sizes and timings (cf. Section 2), the types and number connections initiated by a host (its "social behavior") [67, 70], and various machine learning techniques [138, 89, 88]. However, the feasibility of deploying more sophisticated classification strategies for DPI at scale remains unclear [89, 104, 39].

A controversial application of DPI is when nation-states use it to censor their citizens' use of the Internet. In Figure 10 we provide a summary of six countries and the evidence to support that DPI is used for censorship. Intuitively, it should be the case that encryption does a good job resisting the DPI adversaries in Figure 10. URLs should be concealed, keywords can't be identified, and, if industry standard encryption is used, the adversary would have to make a difficult decision: either blacklist or whitelist all encrypted traffic. However, it turns out that even encryption can have characteristic fingerprints. As an example, Iran blocked Tor based on its TLS certificates in 2011 [43]. In more extreme cases, countries are willing to throttle or completely block whole classes of encrypted traffic [96]. Unfortunately, traditional encryption is an insufficient solution.

Censorship	**Country**					
Strategy	**China**	**Iran**	**Pakistan**	**Syria**	**Russia**	**Turkey**
DNS/URL	[33, 113]	[113, 5, 10]	[87]	[27]	[113]	[113]
keyword	[93, 135, 71, 124]	[10]	-	[27]	-	-
application	[124]	[10]	-	[27]	-	-

Table 10: A summary of blacklist strategies employed by six countries. Each citation references an empirical study that confirms that the blacklist strategy was employed for an extended period of time within the country. A "-" indicates there is no published evidence to support that the blacklist strategy has been deployed in the specific country.

Year	**Name**	**Type**	**Low latency?**	**Used by...**	**Ref.**
2010	Collage	tunneling	no	-	[21]
2011	Dust	randomization	yes	-	[123]
2011	Telex	tunneling	depends	-	[134]
2011	Cirripede	tunneling	depends	-	[64]
2012	StegoTorus	mimicry	yes	-	[120]
2012	SkypeMorph	mimicry	no	-	[85]
2012	Censorspoofer	tunneling	no	-	[115]
2012	SWEET	tunneling	no	-	[139]
2013	FreeWave	tunneling	no	-	[63]
2013	ScambleSuit	randomization	yes	Tor	[125]
2010-2014	obfs2/obfs3/obfs4	randomization	yes	Tor	[125]
2014	CloudTransport	tunneling	no	-	[18]
2014	Facet	tunneling	no	-	[72]

Table 11: A summary of proposed censorship-circumvention solutions. **Strategy** describes the underlying strategy used for the system. **Low latency** indicates if the system can be used for tasks such as web browsing. **Used by...** listed which systems the solution has actually been deployed in.

There have been a number of proposed solutions for censorship circumvention, we list some of them in Figure 11. Censorship-circumvention solutions can be divided into three types: randomization, mimicry and tunneling. We discuss each of these types in detail, in the remainder of this section.

5.1 Randomization

For systems implementing the randomization approach, the primary goal is to remove all static fingerprints in the content and statistical characteristics of the connection, effectively making the traffic look like "nothing." The obfs2 and obfs3 [110] protocols were the first to implement this approach by re-encrypting standard Tor traffic with a stream cipher, thereby removing all indications of the underlying protocol from the content. Recently, improvements on this approach were proposed in the ScrambleSuit system [125] and obfs4 protocol [110], which implement similar content randomization, but also randomize the distribution of packet sizes and inter-arrival times to bypass both DPI and traffic analysis strategies implemented by the censor. The Dust system [123] also offers both content and statistical randomization, but does so on a per-packet, rather than per-connection, level. While these approaches provide fast and efficient obfuscation of the traffic, they only work in environments that block specific types of known-bad traffic (i.e., blacklists). In cases where a whitelist strategy is used to allow known-good protocols, these randomization approaches fail to bypass filtering, as was demonstrated during recent elections in Iran [38].

5.2 Mimicry

Another popular approach to circumvention is to mimic certain characteristics of popular protocols, such as HTTP or Skype, so that blocking traffic with those characteristics would result in significant collateral damage. Mimicry-based systems typically perform shallow mimicry of only a protocol's messages or the statistical properties of a single connection. As an example, StegoTorus [120] embeds data into the headers and payloads of a fixed set of previously collected HTTP messages, using various steganographic techniques. However, this provides no mechanism to control statistical properties, beyond what replaying of the filled-in message templates achieves. SkypeMorph [85], on the other hand, relies on the fact that Skype traffic is encrypted and focuses primarily on replicating the statistical features of packet sizes and timing. Ideally, these mimicked protocols would easily blend into the background traffic of the network, however research has shown that mimicked protocols can be distinguished from real versions of the same protocol using protocol semantics, dependencies among connections, and error conditions [63, 53]. In addition, they incur sometimes significant amounts of overhead due to the constraints of the content or statistical mimicry, which makes them much slower than randomization approaches.

We also note that mimicry system share many characteristics with systems that perform network traffic generation. Most traffic generation systems focus on simple replay of captured network sessions [108, 62], replay with limited levels of message content synthesis [37, 102], generation of traffic mixes with specific statistical properties and static content [26, 119], or heavyweight emulation of user behavior with applications in virtualized environments [132]. Many mimicry and tunneling systems share similar strategies, however the key difference is that mimicry systems must transport useful information to circumvent filtering.

5.3 Tunneling

Like mimicry-based systems, tunneling approaches rely on potential collateral damage caused by blocking popular protocols to avoid filtering. However, these systems tunnel their data in the payload of real instances of a target protocol. The Freewave [63] system, for example, uses Skype's voice channel to encode data, while Facet [72] uses the Skype video channel, SWEET [139] uses the body of email messages, and JumpBox [78] uses web browsers and live web servers. CensorSpoofer [115] also tunnels data over existing protocols, but uses a low-capacity email channel for upstream messages and a high-capacity VoIP channel for downstream. CloudTransport [18] uses a slightly different approach by tunneling data over critical (and consequently unblockable) cloud storage services, like Amazon S3, rather than a particular protocol. The tunneling-based systems have the advantage of using real implementations of their target protocols that naturally replicate all protocol semantics and other distinctive behaviors, and so they are much harder to distinguish. Even with this advantage, there are still cases where the tunneled data causes tell-tale changes to the protocols behavior [53] or to the overall traffic mix through skewed bandwidth consumption. In general, tunneling approaches incur even more overhead than shallow mimicry systems since they are limited by the (low) capacity of the tunneling protocols.

6 Censorship Circumvention with Format-Transforming Encryption

Despite the growing use of DPI in many security-critical settings, we are unaware of any work that specifically studies the robustness of state-of-the-art DPI protocol-identification tools in the face of dedicated attacks. Thus, we address the following question: *are there practical attacks that will force* any *regular-expression-based DPI into misclassifying connections as protocols of the attacker's choosing?* And, if such attacks do exist, can these misclassified connections carry practically useful amounts of information?

Our conclusion is that, indeed, misclassification attacks exist against enterprise-grade DPI, and that these attacks can be mounted while carrying sufficient information to surf the web, transfer files, and use Tor [44]. Rather than building an array of ad-hoc schemes to trick specific DPI systems, we instead target the implicit premise underlying modern DPI: that regular expressions (regexes) are sufficient for identifying network protocols. To that end, we develop a generic approach for controlling the format of encrypted data, so that it will match whatever regex we desire to specify. With this ability, we can force protocol misidentification across a broad range of DPI systems.

The development of a generic approach to evasion of regex-based DPI implies that, for settings with adversarial network users, future protocol-identification systems will have to move to more expensive techniques based on machine learning [138, 70, 89, 88], active probing [81], or something else entirely. At the same time, the approach also suggests a promising way forward mimicry-based censorship circumvention.

Format-Transforming Encryption The foundation of our approach is a new cryptographic primitive called format-transforming encryption (FTE). It allows the user to input a regex of their choosing and output ciphertexts that are guaranteed to match it. This gives FTE a built-in mechanism for forcing misidentification by regex-based DPI. It will additionally achieve more traditional privacy and authenticity goals.

We consider a variety of methods to specify the regular expressions that are input to the FTE scheme. In those scenarios where we know which DPI systems are being used, the simplest method is lifting them directly from systems themselves, or manually creating them using knowledge of RFCs and the DPI code. When we do not have information about the DPI system, we provide a simple procedure for learning regexes from network traces of the application-protocols that we wish our traffic to match.

Under the hood, our FTE scheme relies heavily on well-known algorithms for ranking strings in a given regular language [54]. The algorithms were previously suggested for use in the related context of format-preserving encryption [14], however, as far as we are aware, our work gives the first implementation and performance analysis of the algorithms. To realize a working FTE proxy system capable of tunneling arbitrary network traffic, we specify and implement a full, FTE-powered record layer. By this we mean that we build, around the FTE core, logic to manage buffering and fragmentation of incoming plaintext streams on the sender's side, and ciphertext stream buffering, parsing and fragment reassembly on the receiver's side.

We use this FTE proxy system to explore the resistance of six state-of-the-art DPI systems to protocol misclassification attacks. We show that even expensive, proprietary systems can be forced to mistakenly identify FTE-protected traffic as any of a number of target protocols chosen by the user, including HTTP, SMB, and SSH. We stress that our approach works no matter what the underlying FTE-encapsulated application-layer protocol actually is. To the best of our knowledge, this is the first comprehensive analysis exposing how ineffectual modern DPI systems can be rendered.

One immediate implication is that our proxy system provides a ready-made mechanism for circumventing actual, deployed DPI tools. When used to surf the web, FTE imposes as little as 16% bandwidth overhead and no latency overhead compared to conventional encryption of traffic. By comparison, FTE is both more flexible and efficient than existing circumvention tools that also attempt to prevent proper protocol identification [134, 85, 64] As a practical matter, our FTE system works as a drop-in pluggable transport [7] for Tor, and has been integrated into the official Tor Browser Bundle. Since it's deployment in early 2014, more than 30,000 users have connected to the Tor Network using FTE. Furthermore, the FTE library source code has been released under the Apache License, and its source is freely available[10].

Acknowledgements This work in this section was performed in collaboration with Coull, Ristenpart and Shrimpton. It was presented at ACM Computer and Communications Security in 2013 [47]. The analysis and writing of the results was lead by Dyer with the help of Shrimpton, Ristenpart, and Coull. Dyer lead the engineering effort to produce the experimental results and was assisted by Coull.

6.1 Modern DPI Systems

In our evaluation, we focus on port-independent protocol identification as used by six modern DPI systems. These systems span a wide range of complexity, cost, and expected deployment environments. Here, we discuss details of the systems we evaluate, and present a summary in Figure 12. Unless otherwise mentioned, our discussion and later evaluations will use default configurations.

[10] `https://fteproxy.org/`

System	DPI Type	Multi-stage Pipeline	Classifier Complexity HTTP	 SSH	 SMB
			DFA States		
appid	regex-only	✗	15	8	104
l7-filter	regex-only	✗	55	8	6
YAF	regex-only	✓	29	10	5
			Lines of C/C++ Code		
bro	hybrid	✓	1593	30	1188
nProbe	hybrid	✓	807	89	24
DPI-X	?	?	?	?	?

Table 12: Summary of evaluated DPI systems. *Type* indicates the kind of DPI engine used. *Multi-stage pipelines* chain together several passes over packet contents. *Classifier complexity* is the number of DFA states used for regular expressions or total lines non-whitespace/non-comment C/C++ code.

appid The appid [9] library uses port-based pre-filtering to determine a set of protocol-identifying regexes, against which each TCP stream should be evaluated. It applies each regular expression in the set against the stream, and returns the first match as the protocol label. A match is attempted for bi-directional (i.e., client-server and server-client) streams. In our evaluation we used the latest available version of appid (as of April 2013), and instantiated it via its included Python module.

L7-filter The l7-filter [34] software also performs regex-only matching. However, it differs from appid in that it does not pre-filter based on port numbers. Moreover, l7-filter specifies only regexes to identify uni-directional server-to-client streams. In our evaluation we used version 2009-05-28 of the l7-filter userspace classification engine, and invoked it with its included test suite.

Yet Another Flowmeter YAF [68] is a network monitor that performs application labeling. The YAF protocol-classification engine is predominantly regex-only, although in a few cases (e.g., the classification of TLS) it employs C-based logic. We classify YAF as regex-only because the majority of protocols are classified using a regex-only strategy. Additionally, for some protocols, YAF performs its regex analysis in two stages; HTTP is one example. A first-pass match for HTTP (caused, say, by matching the string HTTP/) triggers a second-pass to extract message contents, such as the User-Agent field of the HTTP request. In our evaluation we used the latest stable version of YAF (2.3.3, January 2013) compiled with application identification support.

Bro Network Security Monitor bro's [94] Dynamic Protocol Detection (DPD) is implemented as a set of regexes and associated C/C++ based parsers, where a parser is triggered in the event that its partnering regular expression(s) match the input stream. The parser is used to extract additional information from the stream, when possible, and to determine when the regular expression match was actually a false positive. In its default configuration, bro parses individual messages and extracts per-message attributes, such as the URI of an HTTP request, or the version number of the SSH protocol being used. In our evaluation, we used the latest, stable version (2.1, Aug. 2012) of bro, and extracted the assigned stream label from the service field of the conn.log file.

For classification of SMB streams, which is not supported in version 2.1, we used an alpha-release SMB parser available in the bro git repository. We bootstrapped this parser into an SMB-classifier by labeling a flow as SMB only if the parser did not encounter parse errors, which is consistent with the strategy of other classifiers present in bro.

nProbe Pro nProbe [40] is an open-source network monitoring utility that costs €299.95 for commercial use. nProbe includes the ability to identify the encapsulated data within a protocol. As an example, it can identify a web request to YouTube or FaceBook. This means that nProbe can re-assemble TCP streams, identify the contents of a flow, and then parse individual message within a flow. nProbe uses hard-coded C-based logic, for the sake of efficiency, to identify attributes that could be captured by a regular expression. As an example, for HTTP it searches for a finite list of values (i.e. GET, POST, HTTP, etc.), in order to identify an HTTP stream. When such a match occurs, a C-based second-pass parser is triggered. In our evaluation we used version 6.9.5 of nProbe Pro, and the NetFlow output value `%L7_PROTO_NAME` to determine nProbe's stream label.

DPI-X We obtained access to an enterprise-grade security gateway device sold by a well-known network equipment manufacturer. The DPI capabilities of this device, as advertised, enable classifying over 900 applications and protocols, including nested and tunneled applications (e.g., Facebook over HTTP). We disclosed our results to the equipment manufacturer, but did not receive approval to release details about the device. Hence, we refer to this system as DPI-X. This device belongs to a class of commercial systems, ranging from lower-end systems capable of handling a maximum throughput of 90 Mbps ($600), up to carrier-grade products capable of 100 Gbps (over $25,000). The system used in our testing has a maximum throughput of 1.5 Gbps and a cost of $8,000. We believe DPI-X is representative of the DPI products employed in many enterprise and carrier networks. In fact, the maker of this product has been identified as one of the suppliers of censorship equipment for Iran [98].

Threat model A primary goal in this work is to experimentally ascertain the extent to which these representative DPI systems are vulnerable to *protocol-misidentification attacks.* (We will use misidentification and misclassification interchangeably.) To define this term, consider a setting in which a DPI system monitors all connections traversing a network. Two parties want to communicate via an application-layer protocol that uses connection(s) traversing the DPI-protected network. We will refer to these two parties as the "attacker" to emphasize that the DPI faces adversarially generated traffic. In a *white-box DPI* attack, the attacker knows the DPI classification algorithms that are used, while in a *black-box DPI* setting the attacker does not. The latter may be because the particular DPI being used is not known (even if the set of possible DPIs is known), or simply because the algorithms are proprietary.

We call the application-layer protocol used for communicating data the *source protocol.* In a misidentification attack, the attacker's goal is to have its connections, which use the source protocol, be (mis)identified by the DPI as a *target protocol* of the attacker's choosing. An example would be to have SSH as the source protocol, and HTTP as the target protocol. In a successful attack, the DPI will incorrectly label the actual (encrypted) SSH connections as (unencrypted) HTTP connections. To declare a practical success, we also require that the attack does not significantly degrade performance of the source protocol.

Despite the clear importance of protocol misidentification attacks, we are unaware of any prior work that evaluate the DPI systems in our test set (or other similar systems). See Section 5 for discussion of related settings and other potential approaches for forcing protocol misidentification that are different from ours.

6.2 Format-Transforming Encryption

All of the open-source DPI systems in our evaluation set use membership in a regular language — either explicitly with regexes or implicitly by logic coded in languages such as C/C++ — to inform application-layer protocol classification. We therefore target mechanisms which enable an attacker to force protocol misidentification against any DPI that relies on regex checks. Note that this goal is more aggressive than merely defeating the systems in our corpus (but we will do that as well!), and we expect our approach will work against any currently deployed regex-based DPI system.

Our main idea is to build DPI resistance into encryption schemes. The key technological enabler, which we explain in the remainder of this section, is augmenting the normal encryption interface to take a regex as an input. The purpose of this regex is to specify the format of ciphertexts: this means that ciphertexts, when taken as a string over the appropriate alphabet, are guaranteed to match against the specified regex. We call the resulting primitive format-transforming encryption (FTE). In turn, we show how to use FTE as a component within a record layer that handles streams of messages from an arbitrary source protocol. Through proper choice of regex, ciphertexts produced by this record layer –which are actually carrying the source protocol– will be classified as messages from another protocol (of our choosing) by the DPI.

Our FTE record layer will be used for two purposes. First, we will build an FTE (single-hop) proxy system by combining the FTE record layer with SOCKS in a straightforward way. We will then use it to support our hypothesis that the regex-based DPI used in practice is fundamentally vulnerable to misclassifcation attacks. We do this by showing how to use the FTE proxy to force protocol misidentification by the entire set of DPI systems and with respect to a variety of target protocols. We also show that doing so has essentially negligible overhead for most relevant regular languages. We believe these results suggest that DPI systems must move to more advanced mechanisms (discussed in Section 5) in settings with adversarial users.

Our second purpose, empowered by the ease with which it forces protocol misidentification and the high performance it obtains, will be to incorporate the FTE record layer into Tor for use in circumventing censorship. We discuss this more in Section 6.5.

6.2.1 FTE via Encrypt-then-Unrank

An FTE scheme is a triple of algorithms: key generation, encryption and decryption. Key generation works as in conventional encryption, outputting a randomly chosen

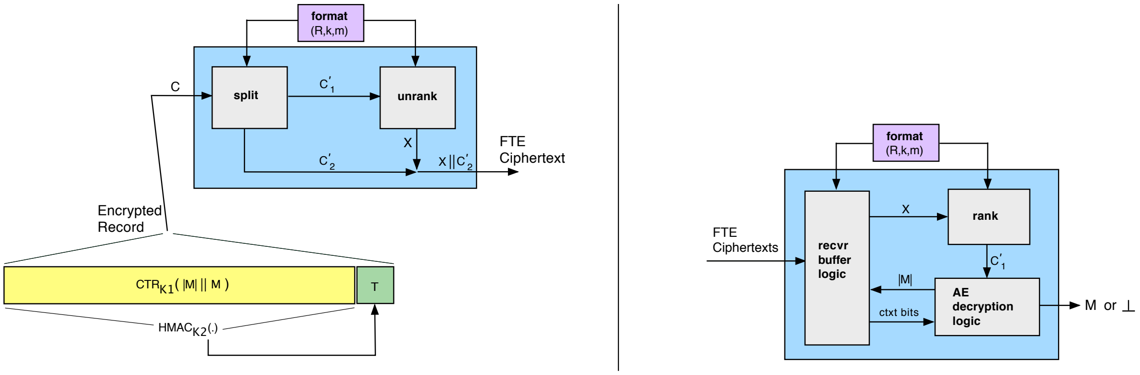


Figure 15: Sender-side (left) and receiver-side (right) record-layer flow. We discuss the various modules in the text.

symmetric key K. Encryption Enc takes inputs of a key K, a format $\mathcal{F}$, and a message M. It can be randomized, stateful, or deterministic, and always outputs a ciphertext C or a distinguished error symbol $\perp$. Decryption Dec takes inputs of a key K, a format $\mathcal{F}$, and a ciphertext C. It outputs a message or $\perp$. The format $\mathcal{F}$ specifies a set $L(\mathcal{F})$ called the language of $\mathcal{F}$. The requirement is that any C output by Enc must be a member of $L(\mathcal{F})$.

FTE is a related to format-preserving encryption (FPE), first formalized by Bellare, Ristenpart, Rogaway, and Stegers (BRRS) [14]. FPE is used in the context of in-place encryption of credit-card numbers (or other records) within databases. It likewise uses formats, but requires that both plaintext messages and ciphertexts be members of the same format-specified language.

We desire FTE schemes that support formats described by regexes. This will allow easy "programming" of formats and endows FTE with the same expressive power as regex-based DPI. To do so, we start by following an approach similar to one used by BRRS. Loosely speaking, to implement $\mathsf{Enc}(K, \mathcal{F}, M)$ for a regular expression $\mathcal{F}$ we: (1) encrypt M using a standard authenticated encryption scheme to obtain an intermediate ciphertext Y; (2) treat Y as an integer in $\mathbb{Z}_{|L(\mathcal{F})|}$ (the set of integers from 0 to the size of the language minus one); and (3) apply an encoding function

$\mathsf{unrank}\colon \mathbb{Z}_{|L(\mathcal{F})|} \to L(\mathcal{F})$. To be able to decrypt, we require that unrank is in fact a bijection with efficiently computable inverse $\mathsf{rank}\colon L(\mathcal{F}) \to \mathbb{Z}_{|L(\mathcal{F})|}$.

The key algorithmic challenge is implementing rank and unrank efficiently. These associate to each string in the language its rank, i.e., its position in a total ordering of the language. The notion of ranking was first explored by Goldberg and Sipser [54] in the context of language compression. Goldberg and Sipser also gave an efficient way to rank a regular language when that language is represented by a deterministic finite automaton (DFA). BRRS used this for an (unimplemented) FPE scheme for arbitrary regular languages encoded as DFAs, but they also emphasize that, asymptotically speaking, there is provably no way to give efficient rank and unrank functions starting just from a regex. Standard tools exist for converting from a regex to an nondeterministic finite automaton (NFA) and from there to a DFA (see Section 6.4), but the second step potentially incurs an exponential blow-up in state size. We observe that this worst-case behavior is not an issue for FTE, in part because the kinds of regexes used by DPIs are themselves designed explicitly to avoid the worst-case blowup.

Our implementation uses the time-space tradeoff of Goldberg and Sipser to support more efficient runtime performance by precomputing tables that allow (un)ranking of all strings $x \in L$ with $|x| \leq n$. (These algorithms appear in Appendix C.) The complexity of this precomputation is $\mathcal{O}(n{\cdot}|\Sigma|{\cdot}|Q|)$, where Σ is the underlying alphabet and Q is the state set for the DFA implementing the FTE regular expression. Given these tables, the complexity of rank_L and unrank_L are $\Omega(n)$ and $\mathcal{O}(n \cdot |\Sigma|)$, where n is the length of the output of rank_L or input of unrank_L, respectively. We can also formalize all of the above and prove that the Encrypt-then-Unrank approach preserves the message confidentiality and authenticity security of the underlying authenticated encryption scheme.

6.2.2 FTE Record Layer

In order to transform arbitrary TCP streams, we need additional "record layer" machinery to buffer, encode, unambiguously parse, and decode streams of FTE messages. In Figure 15 we give an overview of the processes by which plaintexts are transformed into FTE ciphertexts, and vice versa. We assume that sender and receiver share a pre-established set of keys, possibly derived from a single shared master key. Our record layer also assumes an underlying, reliable network transport protocol, e.g. TCP.

Implementing FTE formats We will find it useful to specify in our formats more than just a regular expression. Thus a format $\mathcal{F}$ for the record layer is as a tuple (R, k, m): a regular expression R, a number $k > 0$ that specifies the length of strings to use from the language $L(R)$, and an integer $m \geq 0$ that controls the number of unformatted ciphertext bits to append to the end of the FTE-encoded message. The reason to use a specific length k is that it is an expedient way of rendering easy-to-parse FTE ciphertexts, while the value m is used to cheaply enable more capacity in cases where the desired language is, in fact, any string with a prefix in $L(R)$. All told, for $\mathcal{F} = (R, k, m)$ the language becomes $L(\mathcal{F}) = (L(R) \cap \Sigma^k) \| \Sigma^{\leq m}$ where Σ is the alphabet underlying R. For simplicity we assume $\Sigma = \{0, 1\}$ in what follows, but in implementations one will typically use larger alphabets. In our implementation we use values $m = 2^{18}$ and $k = 2^{10}$, unless otherwise specified.

FTE Sender See the left diagram in Figure 15 and consider a format $\mathcal{F} = (R, k, M)$. Let $L_k(R) = L(R) \cap \{0, 1\}^k$ be the k-bit slice of $L(R)$ we will use and $t = \lfloor \log_2(|L_k(R)|) \rfloor$ be that slice's capacity (the number of bits one can encode using the slice). The first action on the sender's side is to prepare an encrypted record using a secret key K. From a plaintext message buffer, grab a plaintext message M of length at most

$|M| < m$. Then form a plaintext record, containing an encoding of $|M|$ followed by M. The record is then encrypted using a standard, stateful authenticated-encryption (AE) scheme with K to produce a ciphertext C. We assume that $|C|$ is defined solely by $|M|$, which is true for AE schemes used in practice. We also pad M, if required, to ensure that $|C| \geq t$. Our implementation uses CTR mode over AES and then authenticates the resulting ciphertext with HMAC-SHA256.

The encrypted record C is passed to the split module, and appended to an internal buffer maintained by split. The job of split is to produce two strings: one to be passed to unrank for formatting, and one that will be passed along as-is. In particular, split takes up to $t + m$ bits (and at least t bits) from the front of the buffer, which we refer to as C'. Note that C' may be a full AE ciphertext, part of one, or include bits from multiple ciphertexts. Then split partitions C' into C'_1 and C'_2 with $|C'_1| = t$ and $|C'_2| \leq m$. The first portion C'_1 is forwarded to unrank, which produces a formatted string $X \in L_k(R)$. Finally, the concatenation $X \| C'_2$ becomes the sender's FTE ciphertext which can then be transmitted.

FTE Receiver Referring to the right-hand diagram in Figure 15, the receiver buffer logic is responsible for managing the stream of incoming FTE ciphertexts. One issue here, is that there are no explicit markers to demarcate FTE-encoded ciphertext boundaries. We therefore must take care to ensure that the receiver can correctly decrypt given that its buffer may contain multiple contiguous ciphertexts. To do so, the receiver takes advantage of the fact that the sender used a fixed[11] slice $L_k(R)$. As soon as it has the first k FTE ciphertext bits in its buffer, it treats these as a string $X \in L_k(R)$ and applies $\mathsf{rank}(X)$ to recover C'_1. (Should $X \notin L_k(R)$ decryption should abort.) It then feeds C'_1 to the AE decryption algorithm in order to retrieve $\ell = |M|$

[11] One could instead rely on $L(R)$ being prefix-free, but we found using fixed slices simpler and sufficient.

and possibly some initial bits of a message M. (Those latter message bits should not yet be released to higher layers.) Note that this means that the AE scheme must be able to perform incremental decryption and that it should not be vulnerable to attacks (c.f., [4]) that abuse use of the length field before ensuring integrity. CTR mode plus HMAC provides these properties.

Given ℓ the receiver now knows how many more bits of AE ciphertext are expected. From here it can remove the next up to m bits of ciphertext from the input buffer, and then go back into a state where it treats the subsequent k bits in the buffer as a string in $L_k(R)$, applying rank as above, and so on. When it retrieves a full AE ciphertext, it finishes decryption, verifies integrity of the ciphertext, and only now releases the buffered message bits up to the application.

Regex negotiation One unique feature of FTE is the ability to quickly change regexes on the fly. We would like to be able to provide in-band negotiation of formats, but since all data sent on the wire must be formatted to pass DPI checks, it is not possible to negotiate regexes in the clear. We address this limitation by allowing the client/server to agree upon regexes for the duration of a TCP connection assuming they support some initial large set of possible regexes.

In more detail, we assume the client and server have a shared, ordered list of FTE formats $(\mathcal{F}_1, \ldots, \mathcal{F}_n)$, and still assume the client and server have negotiated cryptographic keys out-of-band. For each TCP connection, the client determines the FTE format $\mathcal{F}_i$ it wishes to use for client-to-server messages, and the $\mathcal{F}_j$ it wishes to use for server-to-client message. Then, the client constructs the message $M \leftarrow \langle i \rangle \| \langle j \rangle$, encrypts M as a distinguished `negotiate` message-type, designated by the value of a first (reserved) byte of the plaintext, encodes it with $\mathcal{F}_i$, then sends it to the server.

When the server receives an initial message from the client it iterates through its list of formats, attempting to decode the message with each of the FTE formats. Once it encounters a successful decryption, it evaluates the message and then uses for the session $\mathcal{F}_i$ for client-server messages and $\mathcal{F}_j$ for server-client messages. The server finalizes the negotiation by responding to the client with a distinguished `negotiate_acknowledge` message.

We can improve on the naïve receiver-side implementation of this procedure by having an implementation of rank that contains special checks that short-circuit evaluation to terminate early in cases when the string being ranked is not accepted by the DFA. This enables the server to quickly exclude certain formats, thereby making it possible to support dozens of formats.

6.3 Protocol Misclassification

In this section, we experiment with three strategies for providing regexes used with our FTE record layer: regexes extracted from open-source DPI systems (6.3.1); regexes programmed manually (6.3.2); and regexes automatically learned from samples of target protocol traffic (6.3.3). We will always use a pair of formats: one for upstream (client-to-server) and one for downstream (server-to-client) communications. We will then use our FTE record layer to assess the vulnerability of the DPI systems in our evaluation to protocol-misidentification attacks.

Recall that a source protocol is the application-layer protocol whose connections the attacker wants to have misclassified. We explored HTTP [50], HTTPS [42], secure copy (SCP) [137], and Tor [44] as source protocols, and found that the choice of source protocol does not influence our results. We therefore focus our discussion primarily on HTTP(S).

For each regex generation strategy we explore three target protocols: HTTP [50], SSH [137], and SMB [82]. The latter is a proprietary protocol designed by Microsoft to support sharing of resources, such as files and printers, over networks. We also explored other target protocols, including SIP [99] and RTSP [100], but limit our discussion here to the prior three as the results for the others were the same. Indeed, we suspect FTE can be successfully used for almost any target protocol.

Experimental testbed We implemented our FTE record layer as a library that can be used to relay arbitrary data streams. More implementation details appear in Section 6.4. In our evaluation we use it in the following configuration. The FTE client listens for incoming TCP connections on a local client-side port. Upon receipt of an incoming connection the FTE record layer encrypts the messages using the upstream format and relays them to a remote FTE server. The FTE server receives these FTE ciphertexts, decrypts them, and then relays the recovered plaintexts to a pre-configured destination TCP port. Downstream, the process is reversed, with the FTE server encrypting returned messages using the downstream format.

We use Mozilla Firefox version 17.0.3, controlled by version 2.28.0 of the Selenium browser automation framework to request the Alexa Top 50 (US) websites over the FTE record layer; this is repeated five times for each upstream-downstream format pair tested. The testing framework generates a mixture of HTTP, HTTPS and DNS source traffic and a total of roughly 12,000 TCP connections. On the server-side we relayed all HTTP(S) messages via a SOCKS proxy. We used the default Firefox configuration, with the following exceptions: we tunneled all DNS requests through the SOCKS connection, specified our start page as blank, and disabled caching to disk.

6.3.1 DPI-Extracted Regular Expressions

As our first method of programming the FTE record layer, we build FTE formats by extracting them directly from the source code of open-source regex-based DPI systems. This models a white-box DPI attack setting for the DPI systems from which we extract regexes, and a black-box DPI attack for others (e.g., DPI-X).

Extracting regular expressions Using regexes from DPI systems within our FTE record layer is straightforward. Given a regular expression R from the DPI, we specify a format $\mathcal{F} = (R', k, m)$ where R' is exactly R except without a ".*" (match any) prefix, should there be one. This makes $L(R') \subseteq L(R)$, while ensuring that FTE ciphertexts will include the formatting information found in R after the prefix. We set $k = 128$ and $m = 2^{15}$ for good performance (the misclassification results do not change with other reasonable settings).

We use the naming convention of $\langle$DPI system$\rangle$-$\langle$target protocol$\rangle$ to reference a pair of upstream-downstream regexes extracted from the given DPI system. (Some systems only have a regex for a single direction, in which case we use ".*" for the other direction.) YAF contains at least two regexes for each target protocol, and we indicate that using a 1 or 2 in the format name. The result is 12 different formats.

Misclassification evaluation The misclassification rates for all 12 formats against the 6 classifiers appears in Table 13. Here (and throughout this section) the rate is calculated as the number of TCP connections labeled as the target protocol by the classifier, divided by the total number of TCP connections generated when using that FTE format. Thus a rate of 1.0 means complete misclassification success, and 0.0 is complete failure. Throughout our evaluations, a connection that failed to be misclassified as the target protocol by a DPI system was always marked as an unknown

Format	Misclassification Rate					
	appid	l7-filter	YAF	bro	nProbe	DPI-X
appid-http	1.0	0.0	1.0	0.0	0.0	1.0
l7-http	0.0	1.0	0.16	0.0	0.0	1.0
yaf-http1	0.0	0.0	1.0	0.0	0.0	1.0
yaf-http2	0.0	0.0	1.0	0.57	0.0	1.0
appid-ssh	1.0	0.32	1.0	1.0	0.0	1.0
l7-ssh	0.16	1.0	0.16	0.13	0.0	1.0
yaf-ssh1	1.0	0.31	1.0	1.0	0.0	1.0
yaf-ssh2	1.0	0.21	1.0	1.0	0.0	1.0
appid-smb	1.0	1.0	1.0	0.08	0.0	1.0
l7-smb	0.0	1.0	0.38	0.0	0.0	1.0
yaf-smb1	0.0	0.04	1.0	0.0	0.0	1.0
yaf-smb2	0.0	0.04	1.0	0.0	0.0	1.0

Table 13: Misclassification rates for the twelve DPI-Extracted FTE formats against the six classifiers in our evaluation testbed.

protocol, regardless of the DPI system.

What Table 13 reveals is that DPI-extracted regexes always succeed against the DPI system from which they were extracted, and can even force misclassification by different DPI systems. As an example of the latter, we need only look at DPI-X, which is by far the easiest system to force misclassification against despite having no information about its operation. We confirmed the proper operation of the device by running a variety of control traffic, such as Facebook, Gmail, and SSH, through the device, and found that our results were indeed correct. While we still do not know the exact DPI strategy used, our best guess is that DPI-X performs minimalistic analyses, favoring performance and optimistic labeling of protocols. These regexes were not effective against nProbe, and had varied success against bro. This can be attributed to the latter DPI systems requiring slightly more stringent protocol conformance.

Intersection formats We also consider formats whose regex R is the explicit intersection of multiple DPI regexes. FTE based on such an intersection format will produce ciphertexts matching all of the individual regexes used to form R. Using the intersection of the four DPI-extracted formats for each target protocol resulted in formats with perfect 1.0 misclassification rates for appid, l7-filter, YAF, and DPI-X. The rates against bro and nProbe are comparable to those for the individual regexes.

6.3.2 Manually-Generated Regular Expressions

As our next strategy for producing regexes, we will code them manually. Our FTE record layer makes this easy since most developers are already familiar with regexes due to their use in other programming contexts. Moreover, it turns out to be very simple to build fast, simple regexes that achieve perfect misclassification for *all* classifier/target protocol combinations in our evaluation set.

Coding regexes for FTE We started by inspecting the open-source DPI systems, in particular for cases from the last section where the extracted regexes failed, in order to educate regex design. For HTTP regular expressions, we observed the following requirements. l7-filter requires that responses have an HTTP version of 0.9, 1.0, or 1.1; an HTTP status code in the range of 100-599; and Connection, Content-Type, Content-Length, or Date fields. appid requires that responses have a string of length greater than zero following the status code, and that the status line is terminated with `\r\n`. YAF requires that we have a valid HTTP method verb for requests (i.e. GET, POST, etc.). nProbe requires that we terminate HTTP messages with `\r\n\r\n`. Finally, bro require that we have no payload for HTTP requests, or a payload and a valid Content-Length field — we accommodate this requirement by not allowing a payload for requests and specifying an FTE format parameter of $m = 0$. (Section 6.2.1)

For SSH all classifiers require that the first downstream, and in some cases upstream, messages start with `SSH-`. Next, l7-filter demands that the first two messages in a stream start with `SSH-1.x` or `SSH-1.y`. nProbe requires the first messages in an

SSH stream to be less than or equal to 99 bytes long, and we achieve this by setting our FTE format parameter to $k = 99$ for both directions of traffic, which constricts message lengths.

All classifiers in our evaluation require that SMB messages have a valid SMB fingerprint, which is the byte `\xFF`, followed by `SMB` encoded in ASCII. In addition, nProbe requires that message have a valid length that matches the length of the message payload, that the length field is located as the first 32-bit word in the message, and that `SMB` is encoded as ASCII in the second 32-bit word. Here we encounter a check that is not easily encoded as a regular language (or avoided as above for nProbe's checking of HTTP Content-Length fields), at least if one wants to support all 2^{32} possible lengths. However, we can simply use a specific value for the length field and provide an equivalently sized payload. For the regexes in our experiments, we set the FTE format parameter m to zero, meaning that we do not append any raw AE ciphertext bytes.

Misclassification evaluation We specified regexes that met the above requirements for each of the target protocols. It took less than 30 minutes for one of the authors to specify each regex and debug it by testing it against known DPI engines. The resulting regexes achieved perfect misclassification for all classifier/target protocol combinations, as shown in Table 14. Every TCP connection was tagged as the target protocol of our choosing.

6.3.3 Automatically-Generated Regular Expressions

When we know the DPI systems and their classification methods, the DPI-extracted and manually-generated regexes provide guaranteed evasion and optimal capacity. Unfortunately, there are many cases where it is not possible to know this information, like when the DPI classification strategy abruptly changes or when proprietary systems are used. In these situations, we can use a simple but effective process to automatically generate regexes from network traffic samples using widely-available protocol parsers. This allows us to implicitly learn FTE formats from data that is assumed (or known) to pass DPI scrutiny without raising alerts. Although other, more complex, methods of format discovery are available [19, 20, 126, 107], we focus on well-known network message formats to avoid unnecessary complexity while still providing robust regex generation capabilities. The regex generation process proceeds as follows.

1. Collect packet trace data for target protocol message.
2. Apply a parser to label message fields.
3. Create a set containing observed values for each field, called a *dictionary*.
4. Create a *template* for each message type by replacing the values with placeholders for the associated dictionaries.
5. Convert each dictionary into a regex by concatenating the values with an “or" operator between them.
6. For each template, replace the placeholders with the associated dictionary regexes.
7. Choose one or more of the resultant template regexes as the language(s) used by the FTE system.

There are a number of ways this method can be tuned to adjust the quality and capacity of the resultant language(s). First, we can control the properties of the messages that are collected in the packet trace data, such as their message types or payload lengths. Data containing a single, consistent message type will produce more coherent regexes at the cost of smaller dictionaries, while a mix of message types will produce much larger dictionaries with more capacity but with potentially inconsistent, low-quality regexes. We can also control the regexes through the granularity of the

Format	**Misclassification Rate**					
	appid	l7-filter	YAF	bro	nProbe	DPI-X
manual-http	1.0	1.0	1.0	1.0	1.0	1.0
manual-ssh	1.0	1.0	1.0	1.0	1.0	1.0
manual-smb	1.0	1.0	1.0	1.0	1.0	1.0
auto-http	1.0	1.0	1.0	1.0	1.0	1.0
auto-ssh	1.0	1.0	1.0	1.0	0.0	1.0
auto-smb	1.0	1.0	1.0	1.0	1.0	1.0

Table 14: Misclassification rates for the manually-generated and automatically-generated FTE formats against all six classifiers.

parsers used to break the data into fields. Fine-grained parsing produces a greater number of dictionaries within each template and, consequently, an increase in the number of possible combinations among their values. Conversely, coarse parsers will create templates that are more likely to produce valid outputs, but with less overall capacity.

Regex generation The packet trace data used to evaluate the security of the generated regexes was produced by agents that randomly logged into and crawled HTTP, SMB, and SSH servers. For HTTP, we used the *wget* utility to download the front page of a random selection of web sites on the Alexa Top 1000. SMB and SSH data was generated by scripts that logged into local Linux and Windows servers, randomly crawled the directory structure, accessed files, and logged out several times over the course of a one-hour period. We partitioned the captured data into groups based on their message types and payload lengths. From these partitions, we extracted client headers for HTTP POST requests, SMB transaction requests, and SSH handshake banner messages. The server messages that we use include HTTP 200 OK responses, SMB transaction responses, and SSH handshake banner messages. Each of these message types was parsed using their respective Wireshark dissectors, and the highest-capacity template regular expression was chosen for the FTE format. Any remaining payload bytes not included in the parsed message header were automatically replaced with a regular expression that produced the appropriate number of random bytes.

Misclassification evaluation Table 14 presents the results of our evaluation, and illustrates that the generated regexes achieved perfect misclassification rates except for the nProbe SSH classifier, which had a misclassification rate of zero. The nProbe SSH classifier requires that the first message in each direction be an appropriately formatted banner message with an arbitrary length limitation of 100 bytes. Since our regex generation method is limited to only using what it observes (in this case a single client banner and less than five server banners), the generated regex would have only two bits of capacity. To enhance the capacity, we artificially generated RFC-compliant banner messages with the optional comment field used to carry random bytes up to the specified maximum length of 255 bytes, which provided sufficient capacity and the ability to evade all classifiers but nProbe. This highlights a natural limitation of the simple generation process, though avenues for improvement are possible through more advanced generalization procedures or the use of multiple FTE formats within a single connection (e.g., zero-capacity banner messages followed by high-capacity key exchange messages).

6.4 Performance

Our FTE prototype was developed in C/C++ and Python. Performance-critical algorithms such as rank, unrank, and BuildTable are implemented in C/C++. We use a customized version of the re2 library for regular expression to DFA conversion, and OpenFST for DFA minimization. Cryptographic algorithms are implemented using PyCrypto. Multiple precision arithmetic is performed using GMP. Logic for

FTE format	DFA states	avg. unrank (ms)	avg. rank (ms)
intersection-http-downstream	70	0.52	0.48
intersection-smb-downstream	104	0.55	0.54
intersection-ssh-downstream	11	0.55	0.52
manual-http-downstream	38	0.43	0.43
manual-smb-downstream	130	0.53	0.5
manual-ssh-downstream	9	0.42	0.42
auto-http-downstream	13,815	1.6	1.5
auto-smb-downstream	222	0.82	0.79
auto-ssh-downstream	237	0.52	0.49

Table 15: Average rank and unrank performance for our downstream FTE formats.

the record layer and networking is multi-threaded and implemented in Python.

For performance benchmarks our client machine was an AMD Opteron 8220 SE @ 2.80GHz running CentOS 6.4 and server was an Intel Core i5-2400 @ 3.4GHz running Ubuntu 12.04. Each machine was connected to the Internet at an academic institution.

Rank/Unrank Recall that for FTE we must perform unranking starting from regexes, which can require exponential time in the worst case (see Section 6.2.1). In practice, however, the regexes we require for misidentification attacks admit fast (un)ranking. In Table 15, we present the DFA size and average time to perform ranking and unranking for our intersection, manual, and automated formats over 100k trials using random integers as input. We present performance results for downstream formats only. In all cases the upstream formats perform better than their downstream counterpart in (un)ranking benchmarks and have smaller DFA state-spaces. The FTE format parameters used in these benchmarks are described in Section 7.5. The DPI-extracted intersection formats and manual formats resulted in very compact DFAs, with no more than 130 states. We note that the automatically generated regexes resulted in the largest DFAs, but were still very fast even for $>10,000$ state DFAs. This all evidences that the worst-case blow-up in state sizes when converting

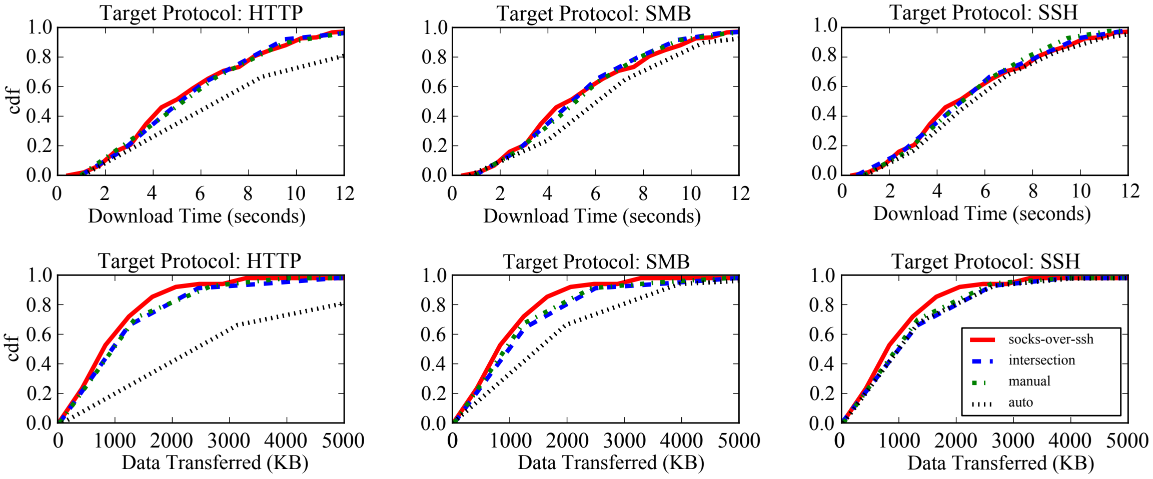


Figure 16: Distribution of webpage (Alexa top fifty) download times (top row) and data transferred (bottom row) for our intersection, manually-generated and automatically-generated FTE formats, compared to using our socks-over-ssh configuration.

from an NFA to DFA does not greatly impede performance.

Web-browsing performance We setup our FTE record layer to proxy HTTP(S) traffic as described in Section 6.3. As a baseline for comparison, we use a conventional encrypted tunnel. Using OpenSSH's integrated functionality we established a SOCKS proxy which listened client-side, such that all connections were routed through the SSH connection to the remote server, and then ultimately on to the destination IP address. We call this our *socks-over-ssh* configuration.

We download the Alexa Top 50 websites five times each using the socks-over-ssh setup. We do the same with the FTE proxy for each of the intersection, manual, and automatically generated regexes. Each of the four separate runs (inclusive of socks-over-ssh) therefore resulted in 250 data points. In our socks-over-ssh configuration, websites in the Alexa Top 50 took an average of 5.5 seconds to render the webpage and all dependencies, and required an average of 1,164KB of data transfer including all TCP/IP overhead. In Figure 16, we show the cumulative distribution of the latency and bandwidth data points for the four different runs.

The lowest average download time of the FTE formats was the intersection-ssh format, which incurred no increase (5.5s avg.) in latency compared to socks-over-ssh, and 1,348KB (16%) increase in data transfer overhead. The highest average download time of all the formats was the automatic-http format, with an average page-render time of 7.1s (29%) and an average of 3,279KB (181%) transferred per website. The increase in data transferred is due to (1) ciphertext expansion and (2) Firefox generating persistent TCP connections that cause FTE/SOCKS negotiation, but do not result in data transfer. These latter empty connections do not use much bandwidth in the socks-over-ssh case.

In our testing we note that engineering issues often overshadow the overheads of FTE. For example, the socks-over-ssh system performed better on sites with low bandwidth requirements and a small number of TCP connections, while our FTE prototype actually performs *better* than socks-over-ssh for websites that use a large number of TCP connections. This is because our FTE implementation uses multi-threading more aggressively than OpenSSH, and this is better aligned with the Firefox's use of multiple TCP connections in parallel.

Goodput As a goodput baseline, we performed a direct copy of a 100MB file with SCP from our server to our client and achieved 58Mbps on average over 100 trials. Average round-trip latency between the client and server was 70ms. When using SCP with the FTE record layer, our best performing format was intersection-ssh and it achieved 42Mbps. All other FTE formats with $m > 0$ exhibited similar performance. For our worst performing format, automated-http, we achieved 1.9Mbps and other formats with $m = 0$ had similar goodput. The slower performance for the latter formats stems from their not allowing raw AE ciphertext bytes to follow unrank output.

In our goodput tests, the FTE implementation was never CPU bound. Hence, the performance of our (un)ranking algorithms was not the bottleneck. Profiling of our FTE prototype indicates that future performance gains could be had by optimizing the buffer management and networking logic.

Memory utilization To determine the memory utilization of our FTE prototype, we first measured the memory requirements of the BuildTable algorithm, which is the largest consumer of memory in our prototype. For all formats, except auto-http, the BuildTable algorithm required at most 2 MB of memory. The auto-http upstream format requires 15 MB and the auto-http downstream format requires 184 MB. As expected, memory utilization increases linearly with respect to DFA state space.

6.5 Censorship Circumvention

In the previous sections, we focused on using FTE to evaluate the efficacy of modern enterprise-grade DPI. We showed that not only can our FTE record layer easily force DPI misclassification, but it can do so while incurring negligible performance impact. This suggests that FTE can be a useful tool for settings where one wants to circumvent DPI-enabled censorship. Here, we experimentally investigate integrating our FTE record layer into the Tor anonymity network as a as a pluggable transport [7].

Integration A pluggable transport is a record-layer mechanism that processes Tor messages before being transmitted on the wire. The only currently deployed transport is obfsproxy [125], which applies a stream cipher to every bit output by Tor using a shared key. Originally, this shared key was a hard-coded, but a newer version (not yet deployed) replaces this with an in-band, anonymous Diffie-Hellman key exchange. The result of this latter approach is a cryptographic guarantee that all the bitstrings seen by a (passive) DPI are indistinguishable from random strings, so that the obfuscated

Tor messages will not have fixed fingerprints. This does not, however, force protocol misclassification, and therefore would fail to bypass DPI that use a whitelist of allowed protocols.

We can do better using our FTE record layer as the pluggable transport. Integrating it into Tor with a hard-coded key is immediate, and it is straightforward to add key exchange to our record layer. Specifically, one could just initiate sessions using the existing obfs3 [125] Diffie-Hellman key exchange, but running the key-exchange messages through our unranking mechanisms before being sent on the wire, since the messages in this exchange are indistinguishable from uniformly random bit strings, they behave like the AE ciphertext bits in our record layer. Together with our existing library of regex formats, we arrive at a version of Tor that can easily force misclassification for the DPI systems currently used in practice. Indeed we verified that misclassification rates for all the six systems in our corpus are as seen in Section 6.3, but now using Tor with FTE as the pluggable transport. We believe that FTE is an attractive pluggable transport option for several reasons:

- *Flexibility*: The FTE record layer already supports a variety of target protocols, and adding new ones requires only specifying new regexes. Extending prior steganographic systems to support many targets (see Section 5), on the other hand, would be very labor intensive.
- *Sufficiency*: The FTE record layer forces misclassification by all evaluated DPIs, even DPI-X whose proprietary classification strategy is unknown to us and is similar to systems used in censorship settings [98].
- *Speed*: The FTE record layer has essentially negligible overhead and, when used with Tor, its overhead is lost in the noise of the Tor network's performance variability.

FTE through the GFC We set up an FTE client on a Virtual Private Server (VPS) located within China and an FTE server in the United States. The server was configured to accept incoming connections on port 80. To set a censorship baseline, we first attempted to browse several websites that are known to be censored, including YouTube and Facebook, *without* using FTE to tunnel the traffic, and found that these sites were blocked. We then attempted to browse the same websites through the FTE tunnel, using our intersection, manual, and automatic HTTP formats. In every case, the FTE tunneled traffic successfully traversed the GFC, and we were able to browse the censored websites.

Next, we considered using FTE to tunnel Tor traffic. Again, to set a baseline, we attempted to connect to a private Tor bridge listening on port 443 of our server, using the default Tor distribution. We observed behavior consistent with the recent analysis of the Great Firewall of China (GFC) by Winter et al. [124]: initial Tor connections to our private bridge were successful, and they were followed by an active probe from a Chinese IP address after roughly 15 minutes. The probe performed a handshake with the bridge, then blacklisted the (IP,port)-combination used by the bridge. We validated the blacklisting by observing that subsequent attempts to connect to our Tor bridge (IP,port)-combinations resulted in a successful SYN packet from the VPS to our bridge, followed by spoofed TCP RSTs transmitted to the client and bridge to terminate the TCP connection.

Having established that Tor was indeed being censored, we then attempted the same tasks through our FTE tunnel, again using each of our intersection, manual, and automatic HTTP formats. Despite port 443 being blacklisted from our previous Tor tests, using FTE on port 80 was successful, and we were able to circumvent the GFC with this FTE-tunneled Tor circuit. After these initial tests we established a persistent FTE tunnel between our FTE client and server. Every five minutes we selected a censored URL and downloaded it through our FTE-powered tunnel. This tunnel remained active for one month, and successfully subverted the GFC until the termination of our VPS account.

On detecting FTE Censors have been aggressive at rolling out new DPI-based mechanisms for detecting and blocking circumvention tools. How will FTE fare in this kind of arms race? The first idea would be for DPI systems to obtain the FTE regex formats and then use them to mark any traffic exactly matching the regex as FTE. For most of the regexes we consider, this would lead to prohibitively high false positive rates (e.g., 100% of HTTP traffic). The one exception is the automatically generated regexes, which may not match against other traffic, because of (for example) time stamps or unique hash values that were learned from collected traffic traces.

A second approach might be for DPI to perform more non-regular checks, such as verifying correctness of length fields for protocols that include them, e.g., the Content-Length field of HTTP responses. Note that actually this example would not work for DPI in practice, as one-third of the response messages generated when downloading the Alexa Top 50 did not include a valid Content-Length field despite it being strongly recommended in the HTTP RFC. Elsewhere this kind of check can be addressed by, for example, developing formats that encode a large number of lengths that are frequently observed in legitimate traffic, and for each length ensuring appropriate length of ciphertexts. In theory, one could also use FTE for more powerful language classes (i.e., an algorithm for ranking unambiguous CFGs appears in [54]).

More generally, DPI is faced with finding checks for protocol semantics or formatting with fidelity beyond what is captured by the FTE record layer. Since the latter takes a minimalist approach, there are innumerable ways in which FTE communications differ from real target protocol runs. The recent work of Houmansadr et al. [63], for example, shows how to exploit discrepancies in other circumvention systems that do much more than FTE in terms of attempting to mimic a target protocol [120, 85, 115]. However, finding such discrepancies is easy, and the hard open question (not addressed in [63]) is how to make such checks effective—fast, scalable, and with negligible errors—in the messy deployment environments faced by DPI and for all of the essentially arbitrary target protocols FTE supports.

The GFC, as discussed above, also engages in active probing. For example, attempting to connect to destination systems suspected of undesirable behavior. Determining how to resist such active attacks in practice is an ongoing research topic (c.f., [103]). Use of FTE, however, can hope to force active probing for all legitimate connections using the target protocol, vastly increasing the complexity of such censorship techniques.

6.6 Concluding Thoughts

Since the publication of this work there has been a number of follow on works that consider new FTE constructions. In 2014, Luchaup et al. [75] proposed a novel algorithm to perform ranking of regular languages, directly from the NFA representation of a language. Initial tests show a marked reduction in memory usage for some languages, compared to the DFA-based ranking presented in this section.

In another work [76], Luchaup et al. proposes a novel ranking scheme for context-free languages, this can be used as a drop-in replacement for the ranking algorithms presented in this section. This work opens the door for a whole new class of languages that can be used to bypass DPI systems that employ context-free grammars for protocol analysis.

7 Marionette: A Unified Framework for Censorship Circumvention

To combat application-layer filtering, several systems have been proposed to obfuscate packet payloads, and generally hide the true protocol being transported. As we discussed in Section 5, these methods fall into one of three categories: those that use encryption to fully randomize the messages sent (e.g., obfs4 [110], ScrambleSuit [125], Dust [123]), those that use encryption in combination with some lightweight ciphertext formatting (e.g. FTE [48], StegoTorus [120]), and those that piggyback off of existing software artifacts (e.g. FreeWave [65], Facet [72], SkypeMorph [85]). A few of these systems have been deployed and are currently used by more comprehensive circumvention systems, such as Lantern [1], uProxy [2], and Tor [44].

Motivated by their initial, positive impact, we set out to address challenges that these obfuscation schemes do not (and cannot). For example, none of the obfuscation methods just mentioned can provide protocol-compliant HTTP traffic to traverse a simple HTTP-proxy like Squid [121]. Methods like FTE and StegoTorus fail because they make no effort to enforce protocol compliance across the (ciphertext) HTTP-messages that they create; nor can they tolerate message-content changes that HTTP-proxies routinely make. Randomizing methods, such as obfs4, will be rejected by an HTTP-proxy for obvious reasons.

This example surfaces two important observations. The first is relatively specific: forward-looking systems must support stateful behaviors. Minimally, messages exchanged by the two ends of the obfuscated tunnel should respect protocol semantics, at least to the level required by the expected monitoring apparatus. The second observation is related, but more general. To date, obfuscation schemes each seem to adopt a particular, fairly narrow design target, and this makes them brittle in practice. Consider randomizing schemes, whose design target is, effectively, to destroy all structure in the messages they create. These are not useful in settings where the monitor employs whitelisting, where only specific application-layer protocols are allowed. Iran's recent elections [38] showcased such a setting. Going in the exact opposite direction, systems like FreeWave and Facet target delivery of ciphertexts that, as a collection, respect one specific protocol. But when that protocol is blacklisted, as Skype was in Ethiopia [86], the scheme becomes useless. Currently, the only system that allows any type of dynamic updating of its behavior is FTE [48]. But it, too, relies on an (arguably) narrow design goal of mimicking protocols at the level of a single message.

The main contribution of this section is that we give a system-building framework that supports a rich design space: a broad range of fidelity in respecting any particular protocol, choice of which protocol to target, the ability to change and schedule target protocols during a communication session, and mechanisms to admit shaping of traffic-related distributions.

Models. The conceptual core of our framework are a powerful kind of probabilistic automata, loosely inspired by probabilistic input/output automata [133]. These provide an intuitive method for modeling, and controlling, features of the messages our system will produce. For example, we use them to model the (potentially) probabilistic sequencing of message types, and whole protocols. Each transition between states in a model may have an associated block of actions. If the model is capturing, say, the scheduling of protocol messages on a channel (see Figure 17), examples actions might be to encrypt/obfuscate the next block of user data, and postprocess it to have appropriate, protocol-specific fields. If the model is capturing a simple web-object retrieval the states might be *which* protocol to mimic, and the actions then spawn models for those protocols. (Which, in turn, produce messages according

Case Study	MC	SB	MLC	ID	TS	Protocol(s)	(Down/Up)
Regex-Based DPI	✓	-	-	-	-	HTTP, SSH SMB	52.0 / 52.0 Mbps
Proxy Traversal	✓	✓	-	-	-	HTTP	5.5 / 0.5 Mbps
Protocol Compliance	✓	✓	✓	✓	-	FTP, POP3	2.0 / 2.0 Mbps
Traffic Analysis	✓	✓	✓	✓	✓	HTTP	0.8 / 0.4 Mbps

Table 16: Summary of Marionette case studies illustrating breadth of protocols, depth of feature control, and high throughput. MC = Message Content, SB = Stateful Behavior, MLC = Multi-Layer Control, ID = Interconnection Dependencies, TS = Traffic Statistics

to that model.) Hierarchical models support intuitive traffic-feature control stacks, and support system design.

The Marionette System. We provide a system architecture (see Table 18) that puts puts models to use, in relaying arbitrary datastreams between a client and server, the latter typically providing a proxying service. To support turning abstract models into system-usable data structures, we develop a simple domain-specific language (DSL). We will give multiple examples of models and their DSL representation in later sections. Specific model actions are instantiated by plugins, which are functions that the developer supplies; in some cases, these will be calls to existing libraries (e.g. FTE [48]). To support transmission of arbitrary datastreams, we develop a record layer that allows for a variety of practical circumstances (e.g. proxies that multiplex incoming TCP streams) and provides the system with mechanisms to support state-sharing between models across connections.

We evaluate a prototype of Marionette through a series of case studies, which we summarize in Table 16. The Marionette system can navigates a range of conditions, including: bypassing regex-based DPI, generating RFC-compliant FTP traffic, bypassing a protocol-enforcing HTTP proxy, and controlling for traffic features such as message lengths. This far exceeds the capabilities of any previous obfuscation system. Performance results show that our prototype can achieve peak throughput of 52Mbps for formats that bypass DPI, and maintain a throughput of 0.8Mbps even with the most complex specifications in our evaluation.

Template Grammars. In our particular realization of the Marionette system, we make use of another novel abstraction that enables fine-grained control of individual ciphertext formats. A *template grammar* is a probabilistic context-free grammar that compactly describes a language of ciphertext *templates*. These templates are strings that contains placeholder tokens, marking the positions where ciphertext bits may be embedded. Each placeholder has an associated handler, whose job is to do the embedding. For example, handlers may implement direct embedding of a given data string, perform encodings (e.g. the "unrank" functionality of FTE), or retrieve appropriate dates, content lengths, etc.

Acknowledgements The work in this section was performed in collaboration with Coull and Shrimpton, and was published at USENIX Security 2015 [49]. The engineering, analysis and writing was lead by Dyer, with the help of Shrimpton and Coull.

7.1 Models and actions

We aim for a system that admits broad controls over traffic properties, not just those of individual application-layer protocol messages. As one example, it should support sending a (ciphertext) message that is correct, with respect to what has been set so far, for the expected semantics of the mimicked protocol. For another, it should respect the observed distributions of message timings and sizes. A natural approach to modeling this sort of control is a probabilistic state machine. We follow this path, introducing a kind of state machine that is well-suited to our needs, and flexible enough to support a wide range of design approaches.

Marionette models. A *marionette model* (or just model, for short) is a tuple $M = (Q, Q_{\text{nrm}}, Q_{\text{err}}, C, \Delta)$. The *state* set $Q = Q_{\text{nrm}} \cup Q_{\text{err}}$, where Q_{nrm} is the set of *normal states*, Q_{err} is the set of *error states*, and $Q_{\text{nrm}} \cap Q_{\text{err}} = \emptyset$. We assume that Q_{nrm} contains a distinguished start state, and that at least one of $Q_{\text{nrm}}, Q_{\text{err}}$ contains a distinguished finish state. The set C is the set of *actions*, which are (potentially) randomized algorithms. A string $\mathcal{B} = f_1 f_2 \cdots f_n \in C^*$ is called an *action-block*, and it defines a sequence of actions. Finally, Δ is a transition relation $\Delta \subseteq Q \times C^* \times (\mathsf{dist}(Q_{\text{nrm}}) \cup \emptyset) \times \mathbb{P}(Q_{\text{err}})$ where $\mathsf{dist}(X)$ the set of distributions over a set X, and $\mathbb{P}(X)$ is the powerset of X. The roles of Q_{nrm} and Q_{err} will be made clear shortly.

A tuple $(s, \mathcal{B}, (\mu_{\text{nrm}}, S)) \in \Delta$ is interpreted as follows. When M is in state s, the action-block $\mathcal{B}$ may be executed and, upon completion, one samples a state $s'_{\text{nrm}} \in Q_{\text{nrm}}$ (according to μ_{nrm}), and the $\{s'_{\text{nrm}}\} \cup S$ is the set of valid next states. In this way, our models have both proper probabilistic and nondeterministic choice, a la probabilistic automata [133]. When $(s, \mathcal{B}, (\mu_{\text{nrm}}, \emptyset)) \in \Delta$, then only transitions to states in Q_{nrm} are possible, and similarly for $(s, \mathcal{B}, (\emptyset, S))$.

In practice, normal states will be states of the model that are reached under normal, correct operation of the system. Error states are reached with the system detects an operational error, which may or may not be caused by an active adversary. For us, it will typically be the case that the results of the action-block $\mathcal{B}$ determine whether or not the system is operating normally or is in error, thus which of the possible next states is correct.

Discussion. Marionette models support a broad variety of uses. One is to capture the intended state of a channel between two communicating parties, i.e. what message the channel should be holding at a given point in time. Such a model serves at least

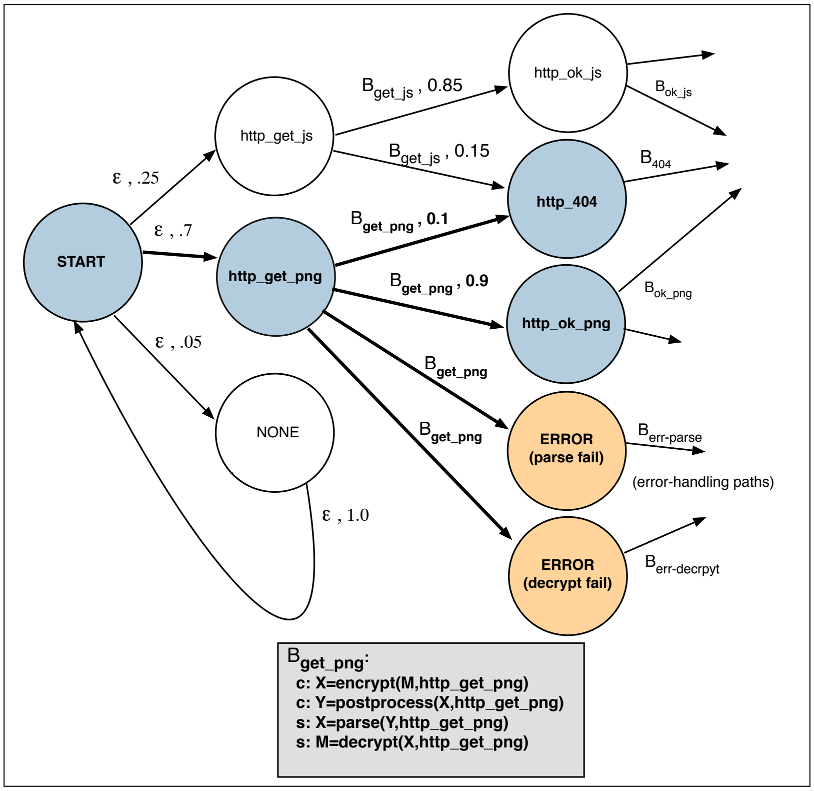


Figure 17: A (partial) graphical representation of a marionette model for an HTTP exchange. The text discusses paths marked with bold arrows; normal states on these are blue, error states are orange.

two related purposes. First, it serves to drive the implementation of procedures for either side of the channel. Second, it describes what a passive adversary would see (given implementations that realize the model), and gives the communicating parties some defense against active adversaries. The model tells a receiving party exactly what types of messages may be received next; receiving any other type of message (i.e. observing an invalid next channel state) provides a signal to commence error handling, or defensive measures.

Consider the partial model in Figure 17 for an exchange of ciphertexts that mimic various types of HTTP messages. The states of this model represent effective states of the shared channel, i.e. what message type is to appear next on the channel. Let us refer to the first-sender as the client, and the first-receiver as the server. In the beginning, both client and server are in the start state. The client moves to state http_get_js with probability 0.25, state http_get_png with probability 0.7, and state NONE with probability 0.05. In transitioning to any of these states, the empty action-block is executed (denoted by ε), meaning there are no actions on the transition. Note that, at this point, the server knows only the set $\{\text{http\_get\_js}, \text{http\_get\_png}, \text{NONE}\}$ of valid states and the probabilities with which they are selected.

Say that the client moves to state http_get_png, thus the message that should be placed on the channel is to be of the http_get_png type. The action-block $\mathcal{B}_{\text{get\_png}}$ gives the set of actions to be carried out in order to affect this; we have annotated the actions with "c:" and "s:" to make it clear which meant to be executed by the client, and which are meant to be executed by the server. The client is to encrypt a message M using the parameters associated to the handle http_get_png, and then apply any necessary post-processing in order to produce the (ciphertext) message Y for sending. The server, is meant to parse the received Y (e.g. to undo whatever was done by the post-processing), and then to decrypt the result.

If parsing and decrypting succeed at the server, then it knows that the state selected by the client was http_get_png and, hence, that it should enter http_404 with probability 0.1, or http_ok_png with probability 0.9. If parsing fails at the server (i.e. the server action parse(Y,http_get_png) in action block $\mathcal{B}_{\text{get\_png}}$ fails) then the server must enter state ERROR (parse fail). If parsing succeeds but decryption fails (i.e., the server action decrypt(X,http_get_png) in action block $\mathcal{B}_{\text{get\_png}}$ fails) then the server must enter state ERROR (decrypt fail). At this point, it is the client who must keep alive a front of potential next states, namely the four just mentioned (error states are shaded orange in the figure). Whichever state the server chooses, the associated action-block is executed –intuitively, that block will have the server acting first to create a ciphertext, and the client second– and progress through the model continues until it reaches the specified finish state.

Models provide a useful design abstraction, specifying (for example) allowable sequencings of ciphertext messages, as well as the particular actions that the communicating parties should realize in moving from message to message, e.g. encrypt or decrypt according to a particular ciphertext format. In practice, we do not expect sender and receiver instantiations of a given model will be identical. Indeed, probabilistic/nondeterministic choices made by the sender will need to be "determinized" by the receiver, and the way that the two sides respond to errors may differ. In Section 7.8 we will consider concrete specifications of models.

7.2 Templates and Template Grammars

In an effort to allow fined-grained control over the format of individual ciphertexts on the wire, we introduce the ideas of ciphertext-format templates, and grammars for creating them. *Templates* are, essentially, partially specified ciphertext strings. The unspecified portions are marked by special placeholders, and each placeholder will be ultimately be replaced by an appropriate string, e.g. an ascii string representing a date, a hexadecimal value representing a color, a URL of a certain depth. To compactly represent a large set of these templates, we will use a probabilistic context-free grammar. Typically, a grammar will create templates sharing a common motif, for example HTTP request messages, or CSS files.

Template Grammars. A template grammar $G = (V, \Sigma, R, S, p)$ is a probabilisitic CFG, and we refer to strings $T \in L(G)$ as templates. The set V is the set of non-terminals, and $S \in V$ is the starting non-terminal. The set $\Sigma = \Sigma \cup P$ consists of two disjoint sets of symbols: Σ are the *base terminals*, and P is a set of *placeholder terminals* (or just placeholders). Collectively, we refer to Σ as template terminals.

The set of rules R consists of pairs $(v, \beta) \in V \times (V \cup \Sigma)^*$, and we will sometimes adopt the standard notation $v \to \beta$ for these. Finally, the mapping $p\colon R \to (0, 1]$ associates to each rule a probability. We require that the sum of values $p(v, \cdot)$ for a fixed $v \in V$ and any second component is equal to one. For simplicity, we have assumed all probabilities are non-zero.

The mapping p supports a method for sampling templates from $L(G)$. Namely, beginning with S, carry out a leftmost derivation and sample among the possible productions for a given rule according to the specified distribution.

Template grammars produce templates, but it is not templates that we place on the wire. Instead, a template T serves to define a set of strings in Σ^*, all of which share the same template-enforced structure. To produce these strings, each placeholder $\gamma \in P$ has associated to it a *handler*. Formally, a handler is a algorithm that takes as inputs a template $T \in \Sigma^*$ and (optionally) a bit string $c \in \{0, 1\}^*$ and outputs a string in Σ^*, or the distinguished symbol $\perp$, which denotes error. A handler for γ scans T and, upon reading γ, computes a string in $s \in \Sigma^*$ and replaces γ with s. The handler halts upon reaching the end of T, and returns the new string T' that is T but will all occurrences of γ replaced. If a placeholder γ is to be replaced with a string from a particular set (say a dictionary of fixed strings, or an element of a regular language described by some regular expression), we assume the restrictions are built into the handler.

As an example, consider the following (overly simple) production rules that could be a subset of a context-free grammar for HTTP requests/responses.

$$
\begin{aligned}
\langle\text{header}\rangle &\rightarrow \langle\text{date\_prop}\rangle\text{: }\langle\text{date\_val}\rangle\texttt{\textbackslash r\textbackslash n}\\
&\qquad \mid \langle\text{cookie\_prop}\rangle\text{: }\langle\text{cookie\_val}\rangle\texttt{\textbackslash r\textbackslash n}\\
\langle\text{date\_prop}\rangle &\rightarrow \texttt{Date}
\end{aligned}
$$

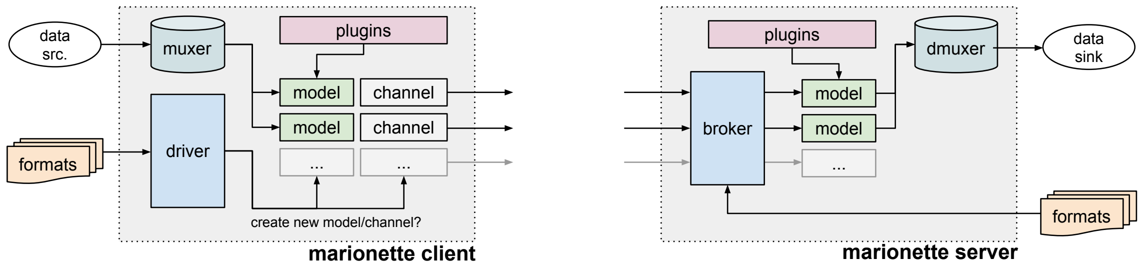


Figure 18: A high-level diagram of the Marionette client-server architecture and its major components for the client-server stream of communications in the Marionette system.

$$\langle \text{cookie\_prop} \rangle \rightarrow \texttt{Cookie}$$

$$\langle \text{date\_val} \rangle \rightarrow \gamma_{\text{date}}$$

$$\langle \text{cookie\_val} \rangle \rightarrow \gamma_{\text{cookie}}$$

To handle our placeholders γ_{date} and γ_{cookie}, we might replace the former with the result of FTE[”(Jan|Feb|...”)], and the latter with the result of running FTE[”([a-zA-Z...)”]. In this example our FTE-based handlers are responsible for replacing the placeholder with a ciphertext that is in the language of its input regular expression. To recover the data we parse the string (according to the the template grammar rules), processing terminals in the resultant parse tree that correspond to placeholders.

7.3 Proxy Architecture

In Section 7.1 we described how a Marionette model can be used to capture stateful and probabilistic communications between two parties. The notion of abstract actions (and action-blocks) gives us a way to use models generatively, too. In this section, we give a high-level description of an architecture that supports this use, so that we may transport arbitrary datastreams via ciphertexts that adhere to our models. We will discuss certain aspects of our design in detail in subsequent sections.

Figure 18 provides a diagram of this client-server proxy architecture. In addition to models, this architecture consists of the following components:

- The client-side *driver* runs the main event loop, instantiates models (form their model-specifications), and destructs them when they have reached the end of their execution. The complimentary receiver-side *broker* is responsible for listening to incoming connections and constructing and destructing models.

- *Plugins* are the mechanism that allow uesr-specified actions to be invoked in action-blocks. We discuss plugins in greater detail in Section 7.6.

- The client-side *multiplexer* is an interface that allows plugins to serialize incoming datastreams into bitstrings of precise lengths, to be encoded into messages via plugins. The receiver-side *demultiplexer* parses and deserializes streams of cells to recover the underlying datastream. We discuss the implementation details of our (de)multiplexer in Section 7.5.

- A *channel* is a logical construct that connects Marionette models to real-world (e.g. TCP) data connections, and represents the communications between a specific pair of Marionette models. We note that, over the course of a channel's lifetime, it may be associated with multiple real-world data connections.

Let's discuss how data traverses the components of a Marionette system. A datastream's first point of contact with the system is the incoming multiplexer, where it enters a FIFO buffer. Then a driver invokes a model that, in turn, invokes a plugin that wants to encode n bits of data into a message. (Note that if the FIFO buffer is empty, the multiplexer returns a string that contains no payload data and is padded to n bits.) The resultant message produced by the plugin is then relayed to the server. Server-side, the broker attempts to dispatch the received message to a model. There are three possible outcomes when the broker dispatches the message: (1) an active model is able to process it, (2) a new model needs to be spawned, or (3) an error has occurred and the message can't be processed. In case 1 or 2, the cell is forwarded to the demultiplexer, and onward to its ultimate destination. In case 3, the server enters an error state for that message and responds accordingly.

We note that the Marionette system will, in fact, operate with some of its components disabled. As an example, by disabling the multiplexer/demultiplexer we effectively have a traffic generation system that doesn't carry actual data payloads, but generates traffic that abides by our model(s). This shows that there's a clear decoupling of two system features: control over traffic features and relaying of data.

7.4 Implementation

Our implementation of Marionette consists of two component modules, called client and server, which share a common codebase. Each component can be invoked as a command line application or directly instantiated as a thread. Our initial prototype of Marionette is written in roughly three thousand lines of Python code. The Marionette source code and engineering details will be available as free and open-source software[12]. In this section we'll overview some of the major engineering obstacles we overcame to realize Marionette.

In our implementation, the client is always invoked with one associated model, while the server can be invoked with an arbitrary number of associated models. The server must provide a mapping between its models and its connections (e.g., binding this particular model to that particular TCP port). In practice, support for multiple models may be useful for a number of reasons. As one example, a server may want to invoke two models to a single port, where one model is dedicated to serving Marionette clients and a second model is dedicated to error handling. The latter may be used in

[12] `https://github.com/kpdyer`

cases where we wish to resist active probing attacks.

7.5 Record Layer

First, we'll briefly describe the Marionette record layer and its objectives and design. The following record layer is our concrete implementation choices for the multiplexer/demultiplexer described in Section 7.3. Note that the following design choices target settings where a user employs marionette for the purpose of web browsing and may encounter active adversaries, such as protocol enforcing proxies, intercepting communications. Hence, we target features that may not be required in all settings.

Our record layer aims to achieve three goals: it (1) enables multiplexing and reliability of multiple datastreams, (2) aids Marionette in negotiating and initializing models, and (3) provides privacy and authenticity of payload data. Let's walk through each of these goals in turn.

Multiplexing of datastreams. Our goal is to enable datastreams that we tunnel through the Marionette system to not rely upon any single connection for the reliability and in-order delivery of data. What's more, oftentimes Marionette is responsible for tunneling multiple independent datastreams. For this reason, the Marionette record layer provides the ability to reliably multiplex multiple datastreams.

We achieve this by including a *datastream ID* and *datastream sequence number* in each cell, as depicted in Figure 19. Sender side, these values are populated at the time of the cell creation. Receiver side, these values used to reassemble streams and delegate them to the appropriate data sink. The *datastream flags* field may have the value of OPEN, RELAY or CLOSE, to indicate the state of the datastream.

Negotiation and initialization of Marionette models. Upon accepting an incoming message, a Marionette receiver iterates through all transitions to all start states for all models mapped to the listening channel. If one of the action blocks for a transition is successful, the underlying record layer (Figure 19) is recovered and then processed. The *marionette flags* field, in Figure 19, may have three values: START, RUNNING, or END. A START value is set when if this is the first cell transmitted by this model, otherwise the value is set to RELAY until the final transmission of the model where a CLOSE is sent. The *marionette model UUID* field is a global identifier that uniquely identifies the model that transmitted the message. The *marionette instance ID* is used to unique identify the instance of the model that relayed the cell.

For practical purposes, in our proof of concept, we assume that a marionette instance ID is created by either the client or server, but not both. By convention we define the “first sender" (i.e., the party that action block that has message-carrying capacity) as the party that initiates the instance ID. Once established, the marionette instance ID has two potential uses. In settings where we have a proxy between the Marionette client and server the instance ID can be used to determine the model that originated a message. In other settings, the instance ID can be used to enhance performance and seed a random number generator for shared randomness between the client and server.

Encryption of the cell. We encrypt each record-layer cell M using a slightly modified encrypt-then-mac authenticated encryption scheme, namely

$$C = \mathsf{AES}_{K1}(\mathrm{IV}_1 \| \langle |M| \rangle) \| \mathsf{CTR}[\mathsf{AES}]_{K_1}^{\mathrm{IV}_2}(M) \| T.$$

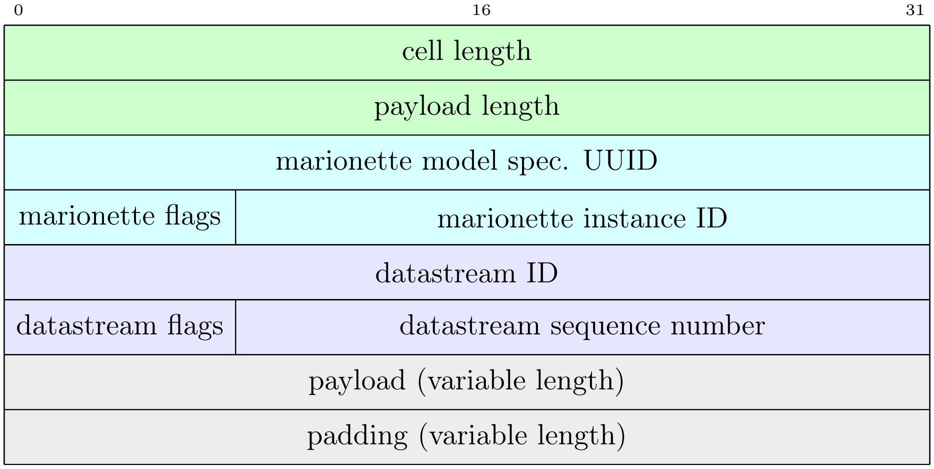

0
16
31
cell length
payload length
marionette model spec. UUID
marionette flags
marionette instance ID
datastream ID
datastream flags
datastream sequence number
payload (variable length)
padding (variable length)

Figure 19: The format of the plaintetxt marionette record layer cell.

The first component of the encrypted record is a header. Here we use AES with key $K1$ to encrypt an encoding of the length of M, along with an initial value IV_1.[13] The second component is the record body, which is the counter-mode encryption of M under IV_2, using E_{K1}. (This is why we put IV_1 in the encrypted record header, to force domain separation between the uses of E.) Finally, we append to this an authentication tag T by running $\mathsf{HMAC\text{-}SHA256}_{K2}$ over the record header and the record body.

7.6 Plugins

The full power of the Marionette system is enabled by it's ability to execute arbitrary user-specified plugins. A plugin is called by the marionette system with four parameters: the current channel, global variables shared across all active models, local variables scoped to our specific model, and the input parameters for this specific plugin. (e.g., the FTE regex or the template grammar name) By use of global and local dictionaries this allows plugins to be stateful and even enable message passing

[13] One could use the cell-length field from the cell in place of $\langle |M| \rangle$. We do not, to make easier the reuse of libraries from the reference implementation of FTE [48].

plugin name	**description**
spawn(S)	spawns models S, blocks until completion
puts(S)	transmits S
fte.send(R)	transmits a record-layer cell FTE-encoded with regex R
tg.send(S)	sends a cell with template grammar S

Table 17: A selection of plugins from our Marionette implementation. Some plugins, such as *spawn*, *fte.send* and *fte.recv* have can also have asynchronous implementations that immediately return success and do not block until completion.

between models. We place very few restrictions on plugins. However, we require that if a plugin fails, it must indicate so and revert any changes it made in attempting to perform the action. In Table 17, we list some of the plugins that we implemented for our proof of concept.

To enable multi-level models, we provide a *spawn* plugin that can be used to spawn new instances of models. In addition, we provide *puts* and *gets* for the purpose of transmitting static strings. As one example, this can be used to transmit a static, non-information carrying banner to emulate, say, an FTP server. As our primary message-level plugins that enable data encoding, we implemented FTE and template grammars (Section 7.2.) Each plugin has a synchronous (blocking until completion) and asynchronous (nonblocking, returns immediately) implementation. The FTE plugin is a wrapper around the FTE[14] and regex2dfa[15] libraries used by the Tor Project for FTE [48]. For the initial version of our template grammar plugin we implemented parsers, template generators manually.

7.7 The Marionette DSL

In order to expose the full potential of the Marionette framework we present a domain-specific language that can be used to compactly describe models. We refer to the

[14] `https://github.com/kpdyer/libfte`

[15] `https://github.com/kpdyer/regex2dfa`

formats that are created using this language as *marionette model-specifications* or *model-specifications* for short. In Figure 20 we have the Marionette language specification.

We have two primary, logical blocks in the model-specification. The `connection` block is responsible for establishing model states and their transition probabilities. The `action_block` is responsible for defining a set of actions, which is a line for each party (client or server) and the plugin the party should execute.

Example: A model-specification for HTTP. Recall the model in Figure 17, which (partially) captures an HTTP connection where the first client-server message is an HTTP GET for a JS or PNG file. Translating the diagram into our Marionette language is a straightforward process. First, we establish our connection block and specify `tcp` and port `80` — this tells the server on which port to listen and client which port to connect. For each transition we create an entry in our connection block. As an example, we added a transition between the `http_get_png` and `http_404` state with probability 0.1. For this transition we execute the `get_png` action block. We repeat this process for all transitions in the model ensuring that we have the appropriate action block for each transition.

For each action block we use synchronous FTE. One party is sending, one is receiving, and neither party can advance to the next state until the action successfully completes. The user is not responsible for creating or destroying the underlying TCP connection. This is transparently handled by Marionette.

7.8 Case Studies

We evaluate the Marionette implementation described in Section 7.4 by building model-specifications for a breadth of scenarios: protocol misidentification against regex-

```
connection([connection_type]):
    start [block_name] [dst] [prob]
    [src] [block_name] [dst] [prob]
    ...
    [src] [block_name]  end  [prob]

action_block [block_name]:
    [client | server] plugin(arg1, arg2, ...)
    ...
```

```
connection(tcp, 80):
    start        NULL    http_get_js  0.25
    start        NULL    http_get_png 0.75
    http_get_png get_png http_404     0.1
    http_get_png get_png http_ok_png  0.9
    http_ok_png  ok_png  ...

action_block get_png:
    client fte.send("GET /\w+ HTTP/1\.1...")

action_block ok_png:
    server fte.send("HTTP/1\.1 200 OK...")

...
```

Figure 20: Top: The Marionette DSL. The connection block is responsible for establishing the Marionette model, its states and transitions probabilities. Optionally, the `connection_type` parameter specifies that type of channel that will be used for the model. **Bottom:** The partial model-specification that implements the model from Figure 17.

based DPI, protocol compliance for complex stateful protocols, traversal of proxy systems that actively manipulate Marionette messages, and controlling statistical features of traffic, such as message lengths.

For each case study we analyze the performance of Marionette for the given model-specification using our testbed. In our testbed, we deployed our Marionette client and server on Amazon Web Services m3.2xlarge instances, in the us-west (Oregon) and us-east (N. Virginia), respectively. These instances include 8 virtual CPUs based on the Xeon E5-2670 v2 (Ivy Bridge) processor at 2.5GHz and 30GB of memory. As our performance metric we report the *goodput*, which is the rate in Mbps at which Marionette can relay datastreams. All performance numbers are the average performance over 30 trials for the transmission of a 10MB file, by repeatedly invoking the chosen Marionette model until the file transfer is complete.

7.8.1 Regex-Based DPI

As our first case study, we confirm that Marionette is able to generate traffic that is misclassified by regex-based DPI as a target protocol of our choosing. Effectively, reproducing results from Section 6, by using the regular expressions referred to as *manual-http*, *manual-ssh* and *manual-smb*. Using these regular expressions, we engineering a Marionette model that invokes the non-blocking implementation of our FTE plugins.

For each configuration we generated 100 datastreams in our testbed and classified this traffic using bro [94] (version 2.3.2) and YAF [68] (version 2.7.1.) We considered it “success" if the classifier reported the *manual-http* datastreams as HTTP, the *manual-ssh* datastreams as SSH, and so on. In all six cases (two classifiers, three protocols) we achieved 100% success. All three formats exhibited similar performance characteristics, which is consistent with the results from [48]. On average, we achieved 52Mbps goodput for both the upstream and downstream directions.

7.8.2 Protocol-Compliance

As our next test, we aim to achieve protocol compliance for scenarios that require a greater degree of inter-message and inter-connection state. In our testing we created model-specifications for HTTP, POP3, and FTP that generate protocol-compliant network traffic. The FTP format was the most challenging of the three, so we'll use it as our case study.

Let's start by analyzing what the control channel for a PASV FTP session looks like and the network traffic it generates. We denote client messages with `C:` and server messages with `S:`.

```
S: 220 Welcome to My FTP Server!

C: USER username

S: 331 Password required for username

C: PASS password

S: 230 Logged in as username.

C: PASV

S: 227 Entering Passive Mode (a,b,c,d,x,y)

C: get myfile.mp3

S: 150 Data connection starting...

S: 226 Transfer Complete.

C: quit

S: 221 Goodbye.
```

An FTP session in passive mode uses two data connections: a control channel and a data channel. In order to enter passive mode a client issues the `PASV` command, and the server responds with a server address of `(a,b,c,d,x,y)`. As defined by the FTP protocol [97], the client then connects to TCP port `a.b.c.d:(256*x+y)` to retrieve the file requested in the `GET` command. Hence, there is a one to many relationship between this conceptual FTP session and the network connections it generates.

Building our FTP model-specification. In building our FTP model we encounter three unique challenges, compared to other protocols such as HTTP:

1. FTP has a sequence of back-and-forth messages between the client and server. In order to maximize potential encoding capacity, we must utilize multiple encoding strategies (e.g., FTE, template grammars, etc.)

2. The FTP protocol is stateful (i.e., message order matters) and has a range of message types (e.g., `USER`, `PASV`, etc,) some of which do not have the ability to

encode information.

3. Performing either an active or passive FTP file transfer requires establishing a new connection and maintaining appropriate inter-connection state.

To address the first challenge, we utilize Marionette's plugin architecture, including FTE, template grammars, model spawning (to spawn the fte_pasv_get model), and the ability to send/receive static strings. To resolve the second, we rely on Marionette's ability to model stateful transitions and block until, say, a specific static string (e.g., the FTP server banner) has been sent/received. For the third, we rely not only on Marionette's ability to spawn a new model, but we also rely on inter-model communications. In fact, we can generate the listening port server-side on the the fly and communicate it in-band to the client via the `227 Entering Passive Mode (a,b,c,d,x,y)` command, which is processed by a client-side template-grammar handler to populate a client-side global variable. This global variable value is then used to inform the spawned model as to which server-side TCP port it should connect. The full model-specification appears in the appendix (Figures 28 and **??**)

Our FTP model-specification relies upon the upstream password field and downstream file transfer in order to encode capacity. In our testbed the FTP model achieved 2 Mbps downstream and 10Kbps upstream goodput. We also created a high-throughput version of the FTP model that operates similarly to the one described here, however it also implements the "put" command and a second file transfer model that switches the roles of the server and client, thereby mimicking an FTP upload and providing 2 Mbps of goodput in both directions.

7.8.3 Proxy Traversal

As our next case, we evaluate Marionette in a setting where a protocol-enforcing proxy is positioned between the client and server. Given the prevalence of the HTTP protocol, and breadth of proxy systems available and deployed, we will focus our attention on engineering Marionette model-specifications that are able to traverse an HTTP proxy system.

In considering an HTTP proxy, there are at least five actions it could do to disrupt marionette communications, which need not be considered in the non-proxied setting. A proxy could (1) add HTTP headers, (2) remove HTTP headers, (3) modify header or payload contents, (4) re-order/multiplex messages, or (5) drop messages. Marionette is able to handle each of these cases with only slight enhancements to the system we've already described.

We first considered using FTE to generate ciphertexts that are valid HTTP messages. However, FTE is sensitive to modifications to its ciphertexts. As an example, changing the case of a single character of an FTE ciphertext would result in FTE decryption failure. We could have compensated for this by anticipating changes a proxy introduces to ciphertexts, which may work in certain settings. However, we want a solution that is general and robust.

Fortunately, there are two primary tools that Marionette provides us that make proxy traversal an easy task: template grammars (Section 7.2), which give us fine-grained control over ciphertexts and allows us to tolerate ciphertext modification, and our record layer (Section 7.5) which allows to tolerate multiplexing and message re-ordering.

Building our HTTP template grammar. As a proof of concept we developed four HTTP template grammars. Two languages that are HTTP-GET requests, one with a header field of `Connection: keep-alive` and one with `Connection: close`. We then created analogous HTTP-OK languages that have keep-alive and close headers. Our model oscillates between the keep-alive GET and OK states with probability 0.9, until it transitions from the keep-alive OK state to the GET close state, with probability 0.1

In all upstream messages we encode data into the URL and cookie fields using the FTE template grammar handler. Downstream we encode data in the payload body using the FTE and follow this with a handler to correctly populate the Content-Length field.

We provide receiver-side HTTP parsers that validate incoming HTTP messages (e.g., ensure content length is correct) and then extract the URL, cookie and payload fields. These parsers, are, of course, able tolerate to significant message modifications including insertion of headers, header re-ordering, or even header deletion of non-information carrying fields, such as, say, the `User-Agent` field.

Coping with multiplexing and re-ordering. The template grammar plugin resolves the majority of issues that we could encounter. However, it doesn't allow us to cope with cases where the proxy might re-order or multiplex messages. By multiplex, we mean that a proxy may interleave two or more Marionette TCP channels into a single TCP stream between the proxy and server. In such a case, we can no longer assume that two messages from the same incoming datastream are, in fact, two sequential messages from the same client model.

In the non-proxy setting there is a one-to-one mapping between channels and server-side Marionette model instances. In the proxied setting, the channel to model instance mapping may be one-to-many. We are able to cope with this scenario by relying upon the non-determinism of our Marionette models, and our record layer.

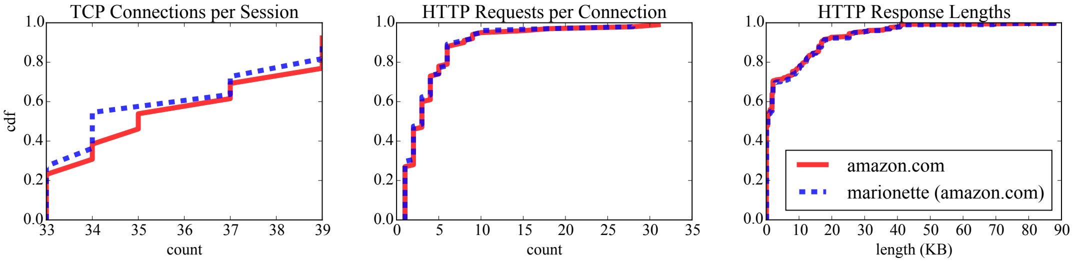


Figure 21: A comparison of the aggregate traffic features for ten downloads of amazon.com using Firefox 35, compared to the traffic generated by ten executions of the Marionette model mimicking amazon.com.

We use non-determinism and modify the server-side broker to attempt to execute all active model instances on each active model. If, for a given channel, no model was successful then the broker attempts to instantiate a new model on that instance. In our plugins we must rely upon our record layer to determine success for each of these operations. As an example, in some cases a message may successfully decode and decrypt, but the model instance ID field doesn't match the current model.

Testing with Squid HTTP proxy. We validated our HTTP model-specification and broker/plugin enhancements against Squid [121] caching proxy version 3.4.9. The squid caching proxy adds headers, removes header, alters headers and payload contents, and re-orders/multiplexes datastreams. In all cases, Marionette was able to successful cope with the changes Squid introduced.

We were able to instigate cases where the Squid proxy dropped messages. However, this only occurred when our HTTP messages were malformed. If a proxy unpredictably dropped messages, we could deal with this by enhancing our record layer to include re-transmit functionality.

In our testbed, our HTTP model-specification for use with Squid proxy achieved 5.5Mbps downstream and 500Kbps upstream goodput.

7.8.4 Traffic Analysis Resistance

In our final case study, we aim to control statistical attributes of HTTP traffic. As our baseline, we visited `Amazon.com` with Firefox 35 and independently captured all resultant network traffic ten times[16]. We then post-processed the packet captures and recorded the following values: the lengths of HTTP response payloads, the number of HTTP request-response pairs per TCP connection, and the number of TCP connections generated as a result of each page visit. Our goal in this section is to utilize Marionette to model the traffic characteristics of these observed traffic patterns to make network sessions that "look like" a visit to `Amazon.com`. We'll discuss each traffic characteristic individually, then combine them in a single model to mimic all characteristics simultaneously.

Message lengths. To model message lengths, we started with the HTTP response template grammar described in Section 7.8.3. We adapted this handler such that it takes an additional, integer value as input. This integer dictates the output length of the HTTP response body. On input n, the handler must return an HTTP response payload of exactly length n bytes.

From our packet captures of `Amazon.com` we recorded the message length for each observed HTTP response payload. Each time our modified HTTP response template grammar was invoked by Marionette, we sampled from the observed distribution of message lengths and used this value as as input to the HTTP response template grammar. With this, we generate HTTP response payloads with lengths that match those observed during our downloads of `Amazon.com`.

[16] Retrieval performed on February 21, 2015.

Messages per TCP connection. We model the number of HTTP request-response pairs per TCP connection using the following strategy, which employs hierarchical modeling. Let's start with the case where we want to model a single TCP connection that has n HTTP request-response pairs. We can achieve this by creating a model M_n with $n+2$ states, and $n+1$ transitions. From the start state of this model there is only one path, and each transition results in an action block that performs one HTTP request-response. This is the most straightforward approach to achieving a connection which contains exactly n request-response pair with probability 1.

Then, we can employ Marionette's hierarchical model structure to have fine-grained control over the number of HTTP request-response pairs per connection. Let's say that we want to have n_1 request-response pairs with probability p_1, n_2 with probability p_2, and so on. For simplicity, we assume that all values n_i are unique, all values p_i are greater than 0, and $\Sigma_{i=0}^{m} p_i = 1$. For each possible value of n_i we create a model M_{n_i}, as described above. Then, we create a single parent model which has a start state with a transition to M_{n_1} with probability p_1, M_{n_2} with probability p_2, and so on. This enables us to create a single, hierarchical model that that models the number of request-response pairs for arbitrary distributions.

Simultaneously active connections. Finally, we aim to control the total number of connections generated by a model during an HTTP session. That is, we want our model to spawn n_i connections with probability p_i, according to some distribution dictated by our target. We can achieve this by taking as the same hierarchical approach used to model request-response pairs, with the distinction that each child model spawns n_i connections.

Building the model and its performance. For each statistical traffic feature, we analyzed the distribution of values in the packet captures from our `Amazon.com` visits.

We then used the strategies in this section to construct a three-level hierarchical model that controls all of the traffic features simultaneously: message lengths, number of request-response pairs per connection, and the number of simultaneously active TCP connections. With this new model we deployed Marionette in our testbed and captured all network traffic it generated. In Figure 21 we have a comparison of the traffic features of the `Amazon.com` traffic, compared to the traffic generated by our Marionette model.

In our testbed, this model achieved 800Kbps downstream and 500Kbps upstream goodput. Compared to Section 7.8.3 this decrease in performance can be explained, in part, by the fact that `Amazon.com` has many connections with only a single HTTP request-response, and very short messages. Hence, it introductions additional overhead to create and manage these connections.

7.9 Conclusion

The Marionette system is the first programmable obfuscation system to offer users the ability to control traffic features ranging from the format of individual application-layer messages to statistical features of connections to dependencies among multiple connections. In doing so, the user can choose the strategy that best suits their network environment and usage requirements. More importantly, Marionette achieves this flexibility without sacrificing performance beyond what is required to maintain the constraints of the model. This provides the user with an acceptable trade-off between depth of control over traffic features and network throughput. Our evaluation highlights the power of Marionette through a variety of case studies motivated by censorship techniques found in practice and the research literature. Here, we conclude by putting those experimental results into context by explicitly comparing them to the state of the art in application identification techniques.

DPI. The most widely used method for application identification available to censors is DPI, which can search for content matching specified keywords or regular expressions. DPI technology is now available in a variety of networking products with support for traffic volumes reaching 30Gbps [32], and has been demonstrated in real-world censorship events by China [124] and Iran [10]. The Marionette system uses a novel template grammar system, along with a flexible plugin system, to control the format of the messages produced and how data is embedded into those messages. As such, the system can be programmed to produce messages that meet the requirements for a range of DPI signatures, as demonstrated in Section 7.8.1

Proxies and Application Firewalls. Many large enterprise networks implement more advanced proxy and application-layer firewall devices that are capable of deeper analysis of particular protocols, such as FTP, HTTP, and SMTP [121]. These devices can cache data to improve performance, apply protocol-specific content controls, and examine entire protocol sessions for indications of attacks targeted at the application. In many cases, the proxies and firewalls will rewrite headers to ensure compliance with protocol semantics, multiplex connections for improved efficiency, change protocol versions, and even alter content (e.g., HTTP chunking). Although these devices are not known to be used by nation-states, they are certainly capable of large traffic volumes (e.g., 400TB/day [6]) and could be used to block most current obfuscation and mimicry systems due to the changes they make to communications sessions. Marionette avoids these problems by using template grammars and a resilient record layer to combine several independent data-carrying fields into a message that is robust to reordering, changes to protocol headers, and connection multiplexing. The protocol compliance and proxy traversal capabilities of Marionette were demonstrated in Sections 7.8.2 and 7.8.3, respectively.

Advanced Techniques. Recent papers by Houmansadr et al. [63] and Geddes et al. [53] have presented a number of passive and active tests that a censor could use to identify mimicry systems. The passive tests include examination of dependent communication channels that are not present in many mimicry systems, such as a TCP control channel in the Skype protocol. Active tests include dropping packets or preemptively closing connections to elicit an expected action that the mimicked systems do not perform. Additionally, the networking community have been developing methods to tackle the problem of traffic identification for well over a decade [25], and specific methods have even been developed to target encrypted network traffic [131].

To this point, there has been no evidence that these more advanced methods have been applied in practice. This is likely due to two very difficult challenges. First, many of the traffic analysis techniques proposed in the literature require non-trivial amounts of state to be kept on every connection (e.g., packet size bi-gram distributions), as well as the use of machine learning algorithms that do not scale to the multi-gigabit traffic volumes of enterprise and backbone networks. As a point of comparison, the Bro IDS system [94], which uses DPI technology, has been known to have difficulties scaling to enterprise-level networks [112]. The second issue stems from the challenge of identifying rare events in large volumes of traffic, commonly referred to as the base-rate fallacy. That is, even a tiny false positive rate can generate an overwhelming amount of collateral damage when we consider traffic volumes in the 1 Gbps range. Sommer and Paxson [104] present an analysis of the issue in the context of network intrusion detection and Perry [95] for the case of website fingerprinting attacks.

Regardless of the current state of practice, there may be some cases where technological developments or a carefully controlled network environment enables the censor to apply these techniques. Since Marionette seeks to support a large range of possible use cases, we must consider how the system could address these issues. As we have shown in Section 7.8.4, the Marionette system is capable of controlling multiple statistical features not just within a single connection, but also across many simultaneous connections. We also demonstrate how our system can be programmed to spawn interdependent models across multiple connections in Section 7.8.2.

8 Concluding Thoughts

This dissertation advances the state-of-the-art in understanding the risks of traffic analysis attacks, and new strategies for circumventing Internet censorship.

In Section 3 and Section 4 we evaluate a range of cryptographic protocols and show that they leak sensitive information about their users. In the context of encrypted messaging applications, such as Apple's iMessage and WhatsApp, we show that a passive network observer can learn sensitive information, such as: the language used for communications and user operating systems. When encryption is used to conceal web-browsing habits it fails to conceal the websites requested, if we assume the attacker has *a priori* information about the sites the user may visit. What's more, we show that per-packet padding is prohibitively expensive (cf. Section 3) or fails completely (cf. Section 4) in these settings.

Our evaluations in Section 3 and Section 4 are compelling. Yet, they do not conclusively show that such attacks are feasible or practical at Internet-scale. As argued by Jaurez et al. [3], real-world considerations such as: caching, localization, and user web-browsing habits can increase the complexities of these attacks for the adversary. Nevertheless, the work of Jaurez et al. [3] is also inconclusive, and motivates the need for future work to reconcile it with the abundance of literature [16, 73, 60, 92, 24, 83] supporting the contrary.

In Section 6 we present a new cryptographic primitive, Format-Transforming Encryption (FTE), that enables users to control output ciphertexts formats with a regular expression. We use FTE to build a proxy system that is able to proxy arbitrary datastreams by transmitting messages that match a user-specified regular expression. Our evaluation shows that this system is able to effortlessly bypass modern DPI systems by coercing them into classifying traffic as a protocol of our choosing. Then, in Section 7 we build a proxy system, called *Marionette*, that is purposely built towards censorship cicrumvention and addresses some of the limitations of the system we present in Section 6. Marionette is powered by a domain-specific language that allows its users to control network traffic features such as format of messages on the wire, using FTE, and the order in which these messages are transmitted. As one example, we show that Marionette can be used to create an RFC-compliant FTP file transfer as cover-traffic, while being able to tunnel arbitrary datastreams.

As we note in Section 5, modern attempts at censorship circumvention systems can be divided into three categories: obfuscation, mimicry and tunneling. It is argued by Houmansadr et al. [63] that developers of a mimcry-based circumvention system are at a fundamental disadvantage because they are required to mimic every single feature of quirk or the system they target. As an example, if one is trying to mimic Skype, they are, ostensibly, required to identify and emulate every single esoteric behavior of the system — even bugs. However, what is not addressed by Houmansadr is the practical considerations that network monitors must overcome, such as: background traffic, NATs, and clients that may be running many simultaneous applications. Indeed, there is open questions about the potential of identify these mimcry systems at Internet scale.

What we can conclude from this work on traffic analysis (Section 3 and Section 4) and censorship circumvention (Section 6 and Section 7) is that many open research questions remain. Therefore, it is critical that we continue this effort to understand the potential for privacy-compromising attacks, and tools we can use to resist them.

References

[1] Lantern. https://getlantern.org/.

[2] uproxy. https://uproxy.org/.

[3] *A critical evaluation of website fingerprinting attacks*, 2014.

[4] Martin R Albrecht, Kenneth G Paterson, and Gaven J Watson. Plaintext recovery attacks against ssh. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 16–26. IEEE, 2009.

[5] Collin Anderson. The hidden internet of iran: Private address allocations on a national network. *CoRR*, abs/1209.6398, 2012.

[6] Apache traffic server. Avaliable at: http://trafficserver.apache.org/.

[7] Jacob Appelbaum and Nick Mathewson. Pluggable transports for circumvention. Available at: https://gitweb.torproject.org/torspec.git/HEAD:/proposals/180-pluggable-transport.txt, 2010.

[8] Inc. Apple. iOS Security. http://images.apple.com/iphone/business/docs/iOS_Security_Feb14.pdf, February 2014.

[9] Arbor Networks, Inc. appid. Available at: https://code.google.com/p/appid/.

[10] Simurgh Aryan, Homa Aryan, and J Alex Halderman. Internet censorship in iran: A first look.

[11] Michael Backes, Goran Doychev, and Boris Köpf. Preventing side-channel leaks in web traffic: A formal approach. In *20th Annual Network and Distributed Sys-*

tem Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013, 2013.

[12] Agathe Battestini, Vidya Setlur, and Timothy Sohn. A Large Scale Study of Text-Messaging Use. In *Proceedings of the* 12^{th} *Conference on Human Computer Interaction with Mobile Devices and Services*, pages 229–238, 2010.

[13] M. Becchi, M. Franklin, and P. Crowley. A workload for evaluating deep packet inspection architectures. In *IEEE International Symposium on Workload Characterization, 2008.*, pages 79–89, 2008.

[14] Mihir Bellare, Thomas Ristenpart, Phillip Rogaway, and Till Stegers. Format-preserving encryption. In Michael Jacobson, Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*. 2009.

[15] Laurent Bernaille and Renata Teixeira. Early recognition of encrypted applications. In Steve Uhlig, Konstantina Papagiannaki, and Olivier Bonaventure, editors, *Passive and Active Network Measurement*, volume 4427 of *Lecture Notes in Computer Science*. 2007.

[16] George Bissias, Marc Liberatore, David Jensen, and Brian Neil Levine. Privacy Vulnerabilities in Encrypted HTTP Streams. In *Proceedings of the Privacy Enhancing Technologies Workshop*, pages 1–11, May 2005.

[17] Reporters Without Borders. Enimies of the internet 2014: Entities at the heart of censorship and surveillance. Avaliable at: `http://12mars.rsf.org/2014-en/`, 2014.

[18] Chad Brubaker, Amir Houmansadr, and Vitaly Shmatikov. Cloudtransport: Using cloud storage for censorship-resistant networking. July 2014.

[19] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[20] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Theory and techniques for automatic generation of vulnerability-based signatures. *Dependable and Secure Computing, IEEE Transactions on*, 5(4):224–241, 2008.

[21] Sam Burnett, Nick Feamster, and Santosh Vempala. Chipping away at censorship firewalls with user-generated content. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, 2010.

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

[23] Xiang Cai, Rishab Nithyanand, Tao Wang, Rob Johnson, and Ian Goldberg. A systematic approach to developing and evaluating website fingerprinting defenses. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 227–238. ACM, 2014.

[24] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 605–616, New York, NY, USA, 2012. ACM.

[25] A. Callado, C. Kamienski, G. Szabo, B. Gero, J. Kelner, S. Fernandes, and D. Sadok. A survey on internet traffic identification. *Communications Surveys Tutorials, IEEE*, 11(3):37–52, rd 2009.

[26] Jin Cao, William S. Cleveland, Yuan Gao, Kevin Jeffay, F. Donelson Smith, and Michele C. Weigle. Stochastic models for generating synthetic http source traffic. In *INFOCOM*, 2004.

[27] Abdelberi Chaabane, Mathieu Cunche, Terence Chen, Arik Friedman, Emiliano De Cristofaro, and Mohammed-Ali Kafaar. Censorship in the wild: Analyzing web filtering in syria. *arXiv preprint arXiv:1402.3401*, 2014.

[28] Peter Chapman and David Evans. Automated Black-Box Detection of Side-Channel Vulnerabilities in Web Applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 263–274, November 2011.

[29] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-Channel Leaks in Web Applications: a Reality Today, a Challenge Tomorrow. In *Proceedings of the* 31^{nd} *IEEE Symposium on Security and Privacy*, pages 191–206, May 2010.

[30] Guang Cheng and Song Wang. Traffic classification based on port connection pattern. In *Computer Science and Service System (CSSS), 2011 International Conference on*, 2011.

[31] Heyning Cheng and Ron Avnur. Traffic Analysis of SSL Encrypted Web Browsing, December 1998. Available at: `http://www.cs.berkeley.edu/~daw/teaching/cs261-f98/projects/final-reports/ronathan-heyning.ps`.

[32] Cisco sce 8000 service control engine. Avaliable at: `http://www.cisco.com/c/en/us/products/collateral/service-exchange/sce-8000-series-service-control-engine/data_sheet_c78-492987.html`, February 2015.

[33] Richard Clayton, Steven J. Murdoch, and Robert N. M. Watson. Ignoring the great firewall of china. In *in G. Danezis & P. Golle, eds, Privacy Enhancing Technologies workshop (PET 2006), LNCS*, 2006.

[34] Clear Foundation. l7-filter. Available at: `http://l7-filter.clearfoundation.com/`.

[35] Marjorie Cohn. NSA Metadata Collection: Fourth Amendment Violation. `http://www.huffingtonpost.com/marjorie-cohn/nsa-metadata-collection-f_b_4611211.html`, January 2014.

[36] Scott Coull and Kevin Dyer. Privacy failures in encrypted messaging services: Apple imessage and beyond. Cryptology ePrint Archive, Report 2014/168, 2014. `http://eprint.iacr.org/`.

[37] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H Katz. Protocol-independent adaptive replay of application dialog. In *NDSS*, 2006.

[38] Holly Dagres. Iran induces internet 'coma' ahead of elections. ALMONITOR, 2013.

[39] Alberto Dainotti, Antonio Pescapè, and Kimberly C Claffy. Issues and future directions in traffic classification. *Network, IEEE*, 26(1):35–40, 2012.

[40] Luca Deri. nprobe: an open source netflow probe for gigabit networks. In *In Proc. of Terena TNC 2003*, 2003.

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

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

[43] Roger Dingledine. Iran blocks Tor; Tor releases same-day fix. Available at: `https://blog.torproject.org/blog/iran-blocks-tor-tor-releases-same-day-fix`, 2011.

[44] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *In Proceedings of the 13th USENIX Security Symposium*, 2004.

[45] Peter Dorfinger, Georg Panholzer, and Wolfgang John. Entropy estimation for real-time encrypted traffic identification (short paper). In Jordi Domingo-Pascual, Yuval Shavitt, and Steve Uhlig, editors, *Traffic Monitoring and Analysis*, volume 6613 of *Lecture Notes in Computer Science*. 2011.

[46] Peter Dorfinger, Georg Panholzer, Brian Trammell, and Teresa Pepe. Entropy-based traffic filtering to support real-time skype detection. In *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, IWCMC '10, 2010.

[47] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, May 2012.

[48] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 20th ACM Conference on Computer and Communications Secuirty (CCS 2013)*, November 2013.

[49] Kevin P. Dyer, Scott E. Coull, and Thomas Shrimpton. Marionette: A programmable network-traffic obfuscation system. In *The Proceedings of USENIX Security 2015*, August 2015.

[50] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June.

[51] Michael Frister and Martin Kreichgauer. PushProxy: A Man-in-the-Middle Proxy for iOS and OS X Device Push Connections. `https://github.com/meeee/pushproxy`, May 2013.

[52] Xinwen Fu, Bryan Graham, Riccardo Bettati, Wei Zhao, and Dong Xuan. Analytical and Empirical Analysis of Countermeasures to Traffic Analysis Attacks. In *Proceedings of the International Conference on Parallel Processing*, pages 483–492, October 2003.

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

[54] A Goldberg and M Sipser. Compression and ranking. In *Proceedings of the 17^{th} Annual ACM symposium on Theory of computing*, STOC '85, 1985.

[55] Dan Goodin. Can Apple Read Your iMessages? Ars Deciphers End-to-End Crypto Claims, June 2013.

[56] Matthew Green. Can Apple read your iMessages? `http://blog.cryptographyengineering.com/2013/06/can-apple-read-your-imessages.html`, June 2013.

[57] Andy Greenberg. Apple Claims It Encrypts iMessages And Facetime So That Even It Can't Decipher Them, June 2013.

[58] Danhua Guo, Guangdeng Liao, Laxmi N. Bhuyan, Bin Liu, and Jianxun Jason Ding. A scalable multithreaded l7-filter design for multi-core servers. In *Proceedings of the 4^{th} ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, 2008.

[59] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11(1), 2009.

[60] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naive-bayes classifier. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, 2009.

[61] Andrew Hintz. Fingerprinting Websites Using Traffic Analysis. In *Proceedings of the Privacy Enhancing Technologies Workshop*, pages 171–178, April 2002.

[62] Seung-Sun Hong and S. Felix Wu. On interactive internet traffic replay. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, pages 247–264, Berlin, Heidelberg, 2006. Springer-Verlag.

[63] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The Parrot is Dead: Observing Unobservable Network Communications. In *The* 34^{th} *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[64] Amir Houmansadr, Giang T.K. Nguyen, Matthew Caesar, and Nikita Borisov. Cirripede: circumvention infrastructure using router redirection with plausi-

ble deniability. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, 2011.

[65] Amir Houmansadr, Thomas Riedl, Nikita Borisov, and Andrew Singer. I Want my Voice to be Heard: IP over Voice-over-IP for Unobservable Censorship Circumvention. In *The* 20th *Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

[66] Marios Iliofotou, Hyun-chul Kim, Michalis Faloutsos, Michael Mitzenmacher, Prashanth Pappu, and George Varghese. Graph-based p2p traffic classification at the internet backbone. In *INFOCOM Workshops 2009, IEEE*, pages 1–6. IEEE, 2009.

[67] Marios Iliofotou, Prashanth Pappu, Michalis Faloutsos, Michael Mitzenmacher, Sumeet Singh, and George Varghese. Network monitoring using traffic dispersion graphs (tdgs). In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, 2007.

[68] Christopher M. Inacio and Brian Trammell. Yaf: yet another flowmeter. In *Proceedings of the 24th international conference on Large installation system administration*, LISA'10, 2010.

[69] Thomas Karagiannis, Andre Broido, Michalis Faloutsos, and Kc claffy. Transport layer identification of P2P traffic. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.

[70] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. Blinc: Multilevel traffic classification in the dark. In *In Proceedings of ACM SIGCOMM*, 2005.

[71] Jeffrey Knockel, Jedidiah R Crandall, and Jared Saia. Three researchers, five conjectures: An empirical analysis of tom-skype censorship and surveillance.

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

[73] Marc Liberatore and Brian Neil Levine. Inferring the source of encrypted http connections. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, 2006.

[74] Ben Lovejoy. Massive Growth in Apple's Cloud-Based Services Eclipsed by Debate on Financials, January 2013.

[75] Daniel Luchaup, Kevin P. Dyer, Somesh Jha, Thomas Ristenpart, and Thomas Shrimpton. Libfte: A toolkit for constructing practical, format-abiding encryption schemes). In *To appear in the proceedings of USENIX Security 2014*, August 2014.

[76] Daniel Luchaup, Thomas Shrimpton, Thomas Ristenpart, and Somesh Jha. Formatted encryption beyond regular languages. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1292–1303. ACM, 2014.

[77] Xiapu Luo, Peng Zhou, Edmond W. W. Chan, Wenke Lee, Rocky K. C. Chang, and Roberto Perdisci. HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. In *Proceedings of the Network and Distributed Security Symposium*, February 2011.

[78] Jeroen Massar, Ian Mason, Linda Briesemeister, and Vinod Yegneswaran. Jumpbox–a seamless browser proxy for tor pluggable transports. *Security and Privacy in Communication Networks. Springer*, page 116, 2014.

[79] Luke Mather and Elisabeth Oswald. Pinpointing side-channel information leaks in web applications. *Journal of Cryptographic Engineering*, 2(3):161–177, 2012.

[80] Luke Mather and Elisabeth Oswald. Quantifying side-channel information leakage from web applications. *IACR Cryptology ePrint Archive*, 2012:269, 2012.

[81] Eitan Menahem, Gabi Nakibly, and Yuval Elovici. Network-based intrusion detection systems go active! In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 1004–1006. ACM, 2012.

[82] Microsoft. [MS-SMB2]: Server Message Block (SMB) Protocol Versions 2 and 3. Available at: `http://msdn.microsoft.com/en-us/library/cc246482.aspx`, 2013.

[83] Brad Miller, Ling Huang, Anthony D Joseph, and J Doug Tygar. I know why you went to the clinic: Risks and realization of https traffic analysis. In *Privacy Enhancing Technologies*, pages 143–163. Springer, 2014.

[84] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[85] 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, 2012.

[86] Katia Moskvitch. Ethiopia clamps down on skype and other internet use on tor. BBC News, 2012.

[87] Zubair Nabi. The anatomy of web censorship in pakistan. *arXiv preprint arXiv:1307.1144*, 2013.

[88] Thuy T. T. Nguyen, Grenville Armitage, Philip Branch, and Sebastian Zander. Timely and continuous machine-learning-based classification for interactive IP traffic. *IEEE/ACM Trans. Netw.*, 20(6), December 2012.

[89] Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys & Tutorials, IEEE*, 10(4):56–76, 2008.

[90] Rishab Nithyanand, Xiang Cai, and Rob Johnson. Glove: A bespoke website fingerprinting defense. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 131–134. ACM, 2014.

[91] Parmy Olson. Watch Out, Facebook: WhatsApp Climbs Past 400 Million Active Users. `http://www.forbes.com/sites/parmyolson/2013/12/19/watch-out-facebook-whatsapp-climbs-past-400-million-active-users/`, December 2013.

[92] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website Fingerprinting in Onion Routing-based Anonymization Networks. In *Proceedings of the Workshop on Privacy in the Electronic Society*, pages 103–114, October 2011.

[93] Jong Chun Park and J.R. Crandall. Empirical study of a national-scale distributed intrusion detection system: Backbone-level filtering of html responses in china. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 315–326, June 2010.

[94] Vern Paxson. Bro: a system for detecting network intruders in real-time. In *Proceedings of the 7^{th} conference on USENIX Security Symposium - Volume 7*, SSYM'98, 1998.

[95] Mike Perry. A critique of website traffic fingerprinting attacks. Avaliable at: `https://blog.torproject.org/`, November 2013.

[96] Phobos. Iran partially blocks encrypted network traffic. Available at: `https://blog.torproject.org/blog/iran-partially-blocks-encrypted-network-traffic`, 2012.

[97] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), October 1985. Updated by RFCs 2228, 2640, 2773, 3659.

[98] Christopher Rhoads and Farnaz Fassihi. Iran vows to unplug internet. Available at: `http://online.wsj.com/article/SB10001424052748704889404576277391449002016.html`, December 2011.

[99] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002.

[100] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard), April 1998.

[101] Yan Shi and Subir Biswas. Website fingerprinting using traffic analysis of dynamic webpages. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 557–563. IEEE, 2014.

[102] Sam Small, Joshua Mason, Fabian Monrose, Niels Provos, and Adam Stubblefield. To catch a predator: A natural language approach for eliciting malicious

payloads. In Paul C. van Oorschot, editor, *USENIX Security Symposium*, pages 171–184. USENIX Association, 2008.

[103] Rob Smits, Divam Jain, Sarah Pidcock, Ian Goldberg, and Urs Hengartner. Bridgespa: improving tor bridges with single packet authorization. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*. ACM, 2011.

[104] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010.

[105] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, 2001.

[106] Qixiang Sun, Daniel R. Simon, Yi-Min Wang, Wilf Russell, Venkata N. Padmanabhan, and Lili Qiu. Statistical identification of encrypted web browsing traffic. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.

[107] Géza Szabó, Zoltán Turányi, László Toka, Sándor Molnár, and Alysson Santos. Automatic protocol signature generation framework for deep packet inspection. In *Proceedings of the 5th International ICST Conference on Performance Evaluation Methodologies and Tools*, pages 291–299. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011.

[108] Tcpreplay. Avaliable at: `http://tcpreplay.synfin.net/`.

[109] Jörg Tiedemann. Parallel Data, Tools and Interfaces in OPUS. In *Proceedings of the* 8^{th} *International Conference on Language Resources and Evaluation*, May 2012.

[110] Tor Project. Obfsproxy. Available at: `https://www.torproject.org/projects/obfsproxy.html.en`, 2013.

[111] International Telecommunications Union. Percentage of individuals using the internet 2000-2012. Avaliable at: `http://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2013/Individuals_Internet_2000-2012.xls`, 2012.

[112] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The nids cluster: Scalable, stateful network intrusion detection on commodity hardware. In Christopher Kruegel, Richard Lippmann, and Andrew Clark, editors, *Recent Advances in Intrusion Detection*, volume 4637 of *Lecture Notes in Computer Science*. 2007.

[113] John-Paul Verkamp and Minaxi Gupta. Inferring mechanics of web censorship around the world.

[114] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 Protocol. In *Proceedings of the USENIX Workshop on Electronic Commerce*, pages 29–40, November 1996.

[115] Qiyan Wang, Xun Gong, Giang Nguyen, Amir Houmansadr, and Nikita Borisov. CensorSpoofer: Asymmetric Communication using IP Spoofing for Censorship-Resistant Web Browsing. In *The* 19^{th} *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[116] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective attacks and provable defenses for website fingerprinting. In *Proc. 23th USENIX Security Symposium (USENIX)*, 2014.

[117] Tao Wang and Ian Goldberg. Improved website fingerprinting on tor. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pages 201–212. ACM, 2013.

[118] Wei Wang, Mehul Motani, and Vikram Srinivasan. Dependent Link Padding Algorithms for Low Latency Anonymity Systems. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 323–332, November 2008.

[119] Michele C. Weigle, Prashanth Adurthi, Félix Hernández-Campos, Kevin Jeffay, and F. Donelson Smith. Tmix: A tool for generating realistic tcp application workloads in ns-2. *SIGCOMM Comput. Commun. Rev.*, 36(3):65–76, July 2006.

[120] 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 *ACM Conference on Computer and Communications Security*, 2012.

[121] D. Wessels and k. claffy. ICP and the Squid web cache. *IEEE Journal on Selected Areas in Communications*, 16(3):345–57, Mar 1998.

[122] Andrew M. White, Austin R. Matthews, Kevin Z. Snow, and Fabian Monrose. Phonotactic Reconstruction of Encrypted VoIP Conversations: Hookt on fon-iks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 3–18, May 2011.

[123] Brandon Wiley. Dust: A blocking-resistant internet transport protocol. Technical report, School of Information, University of Texas at Austin, 2011.

[124] Philipp Winter and Stefan Lindskog. How the Great Firewall of China is Blocking Tor. In *Free and Open Communications on the Internet*, 2012.

[125] Philipp Winter, Tobias Pulls, and Juergen Fuss. Scramblesuit: a polymorphic network protocol to circumvent censorship. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pages 213–224. ACM, 2013.

[126] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, and Scuola Superiore S Anna. Automatic network protocol analysis. In *15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[127] Charles Wright, Scott Coull, and Fabian Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *Proceedings of the 16th Network and Distributed Security Symposium*, February 2009.

[128] Charles V Wright, Lucas Ballard, Scott E. Coull, Fabian Monrose, and Gerald M Masson. Spot Me if You Can: Uncovering Spoken Phrases in Encrypted VoIP Conversations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 35–49, May 2008.

[129] Charles V. Wright, Lucas Ballard, Scott E. Coull, Fabian Monrose, and Gerald M. Masson. Uncovering spoken phrases in encrypted voice over IP conversations. *ACM Trans. Inf. Syst. Secur.*, 13, December 2010.

[130] Charles V. Wright, Lucas Ballard, Fabian Monrose, and Gerald M. Masson. Language identification of encrypted VoIP traffic: Alejandra y roberto or alice

and bob? In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.

[131] Charles V. Wright, Fabian Monrose, and Gerald M. Masson. On inferring application protocol behaviors in encrypted network traffic. *Journal on Machine Learning Research*, 7, December 2006.

[132] CharlesV. Wright, Christopher Connelly, Timothy Braje, JesseC. Rabek, LeeM. Rossey, and RobertK. Cunningham. Generating client workloads and high-fidelity network traffic for controllable, repeatable experiments in computer security. In Somesh Jha, Robin Sommer, and Christian Kreibich, editors, *Recent Advances in Intrusion Detection*, volume 6307 of *Lecture Notes in Computer Science*, pages 218–237. Springer Berlin Heidelberg, 2010.

[133] Sue-Hwey Wu, Scott A Smolka, and Eugene W Stark. Composition and behaviors of probabilistic i/o automata. *Theoretical Computer Science*, 176(1):1–38, 1997.

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

[135] Xueyang Xu, Z Morley Mao, and J Alex Halderman. Internet censorship in china: Where does the filtering occur? In *Passive and Active Measurement*, pages 133–142. Springer, 2011.

[136] Ai-min Yang, Sheng-yi Jiang, and He Deng. A p2p network traffic classification method using svm. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 398–403. IEEE, 2008.

[137] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), January 2006.

[138] Sebastian Zander, Thuy Nguyen, and Grenville Armitage. Automated traffic classification and application identification using machine learning. In *Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on*, pages 250–257. IEEE, 2005.

[139] Wenxuan Zhou, Amir Houmansadr, Matthew Caesar, and Nikita Borisov. Sweet: Serving the web by exploiting email tunnels. *HotPETS. Springer*, 2013.

[140] Ye Zhu, Xinwen Fu, Bryan Graham, Riccardo Bettati, and Wei Zhao. On Flow Correlation Attacks and Countermeasures in Mix Networks. In *Proceedings of the Privacy Enhancing Technologies Workshop*, volume 3424 of *Lecture Notes in Computer Science*, pages 207–225, May 2004.

Appendix A BuFLO Countermeasure

Parameters			Overhead		Classifier Accuracy (%)				
τ	ρ	d	Bandwidth (%)	Latency (s)	LL	H	P	VNG++	P-NB
0	40	1000	93.5	6.0	18.4 ± 2.9	0.8 ± 0.0	27.3 ± 1.8	22.0 ± 2.1	21.4 ± 1.0
0	40	1500	120.0	3.6	16.2 ± 1.6	0.8 ± 0.0	23.3 ± 3.3	18.3 ± 1.0	18.8 ± 1.4
0	20	1000	140.5	2.4	16.3 ± 1.2	0.8 ± 0.0	20.9 ± 1.6	15.6 ± 1.2	17.9 ± 1.7
0	20	1500	201.3	1.2	13.0 ± 0.8	0.8 ± 0.0	24.1 ± 1.8	18.4 ± 0.9	18.7 ± 1.0
10000	40	1000	129.2	6.0	12.7 ± 0.9	0.8 ± 0.0	14.1 ± 0.9	12.5 ± 0.8	13.2 ± 0.7
10000	40	1500	197.5	3.6	8.9 ± 1.0	0.8 ± 0.0	9.4 ± 1.3	8.2 ± 0.8	9.3 ± 1.3
10000	20	1000	364.5	2.4	5.4 ± 0.8	0.8 ± 0.0	7.3 ± 1.0	5.9 ± 1.0	6.8 ± 0.9
10000	20	1500	418.8	1.2	4.4 ± 0.2	0.8 ± 0.0	5.1 ± 0.7	4.1 ± 0.8	5.3 ± 0.5

Table 18: Overhead and accuracy results for the BuFLO countermeasure at $k = 128$.

Appendix B Experimental Results

	Classifier						
Countermeasure	LL	H	P	BW	TIME	VNG	VNG++
None	98.1 ± 0.1	98.9 ± 0.1	97.2 ± 0.2	80.1 ± 0.6	9.7 ± 0.1	93.7 ± 0.2	93.9 ± 0.3
Session Random 255	40.7 ± 0.3	13.1 ± 0.2	90.6 ± 0.3	54.9 ± 0.4	9.5 ± 0.1	87.8 ± 0.3	91.6 ± 0.3
Packet Random 255	80.6 ± 0.4	40.1 ± 0.3	94.9 ± 0.3	77.4 ± 0.6	9.4 ± 0.1	91.6 ± 0.2	93.5 ± 0.3
Pad to MTU	63.1 ± 0.5	4.7 ± 0.1	89.8 ± 0.4	62.7 ± 0.6	9.6 ± 0.2	82.6 ± 0.4	88.2 ± 0.4
Packet Random MTU	45.8 ± 0.4	11.2 ± 0.2	92.1 ± 0.3	64.6 ± 0.5	9.5 ± 0.1	77.8 ± 0.3	87.6 ± 0.3
Exponential	95.4 ± 0.2	72.0 ± 0.4	96.6 ± 0.3	77.1 ± 0.6	9.6 ± 0.1	95.1 ± 0.2	94.8 ± 0.3
Linear	96.6 ± 0.2	89.4 ± 0.2	96.8 ± 0.2	79.5 ± 0.6	9.6 ± 0.2	93.5 ± 0.2	94.3 ± 0.3
Mice-Elephants	84.8 ± 0.4	20.9 ± 0.3	94.5 ± 0.3	72.3 ± 0.6	9.6 ± 0.1	89.4 ± 0.3	91.7 ± 0.4
Direct Target Sampling	25.1 ± 0.6	2.7 ± 0.1	81.8 ± 0.5	41.2 ± 0.9	9.7 ± 0.2	69.4 ± 0.6	80.2 ± 0.5
Traffic Morphing	31.0 ± 0.7	6.3 ± 0.3	88.7 ± 0.4	43.0 ± 0.9	9.8 ± 0.2	81.0 ± 0.5	86.0 ± 0.4

Figure 22: Classifier performance for $k = 128$, using the Herrmann dataset.

	Classifier						
Countermeasure	LL	H	P	BW	TIME	VNG	VNG++
None	87.1 ± 0.6	87.4 ± 0.3	87.5 ± 0.6	55.7 ± 0.7	11.6 ± 0.7	72.0 ± 1.1	76.3 ± 1.0
Session Random 255	25.3 ± 0.4	9.5 ± 0.1	66.1 ± 0.6	38.6 ± 0.5	12.1 ± 0.6	60.5 ± 1.1	68.7 ± 1.2
Packet Random 255	43.6 ± 0.7	13.1 ± 0.3	74.0 ± 0.7	51.5 ± 0.7	11.8 ± 0.6	65.6 ± 1.3	71.8 ± 1.0
Pad to MTU	41.3 ± 0.6	5.0 ± 0.1	69.2 ± 0.7	41.8 ± 0.6	11.6 ± 0.7	56.8 ± 1.2	65.7 ± 1.1
Packet Random MTU	21.8 ± 0.5	7.5 ± 0.1	69.1 ± 0.7	40.2 ± 0.6	11.4 ± 0.8	47.1 ± 1.0	60.3 ± 1.0
Exponential	72.9 ± 0.6	61.2 ± 0.4	82.1 ± 0.8	54.8 ± 0.8	10.5 ± 0.7	74.1 ± 0.8	78.0 ± 0.9
Linear	79.2 ± 0.7	73.9 ± 0.3	84.2 ± 0.6	54.4 ± 0.9	12.0 ± 0.7	70.3 ± 0.9	74.3 ± 1.4
Mice-Elephants	55.9 ± 0.9	25.6 ± 0.3	75.6 ± 0.7	49.3 ± 0.6	11.7 ± 0.5	65.9 ± 1.1	71.2 ± 1.0
Direct Target Sampling	19.4 ± 1.0	2.5 ± 0.3	47.4 ± 1.4	26.8 ± 1.1	11.1 ± 0.7	35.7 ± 3.0	49.7 ± 1.9
Traffic Morphing	20.1 ± 1.2	4.1 ± 0.5	55.3 ± 1.3	25.6 ± 1.1	12.3 ± 0.7	45.4 ± 2.1	56.7 ± 2.0

Figure 23: Classifier performance for $k = 128$, using the Liberatore dataset.

	Privacy Set Size						
Classifier	$k = 16$	$k = 32$	$k = 64$	$k = 128$	$k = 256$	$k = 512$	$k = 775$
None	96.9 ± 0.1	95.9 ± 0.2	95.1 ± 0.2	93.9 ± 0.3	93.3 ± 0.4	91.6 ± 0.6	90.6 ± 0.9
Session Random 255	96.0 ± 0.1	94.7 ± 0.2	93.4 ± 0.2	91.6 ± 0.3	89.4 ± 0.4	85.6 ± 0.6	84.1 ± 0.5
Packet Random 255	96.6 ± 0.1	95.7 ± 0.2	94.5 ± 0.2	93.5 ± 0.3	92.2 ± 0.5	89.8 ± 0.7	88.5 ± 0.8
Pad to MTU	95.2 ± 0.1	93.2 ± 0.2	91.4 ± 0.2	88.2 ± 0.4	84.2 ± 0.5	79.5 ± 0.8	77.3 ± 0.7
Packet Random MTU	95.0 ± 0.1	93.1 ± 0.2	90.8 ± 0.2	87.6 ± 0.3	83.3 ± 0.5	78.6 ± 0.6	74.8 ± 0.8
Exponential	97.1 ± 0.1	96.5 ± 0.1	95.6 ± 0.2	94.8 ± 0.3	93.7 ± 0.4	92.5 ± 0.6	90.8 ± 1.2
Linear	96.9 ± 0.1	96.0 ± 0.2	95.1 ± 0.2	94.3 ± 0.3	92.8 ± 0.5	91.8 ± 0.6	89.5 ± 1.1
Mice-Elephants	96.2 ± 0.1	95.1 ± 0.2	93.6 ± 0.2	91.7 ± 0.4	90.0 ± 0.5	86.5 ± 0.8	84.0 ± 0.8
Direct Target Sampling	93.1 ± 0.2	89.8 ± 0.2	85.3 ± 0.3	80.2 ± 0.5	74.3 ± 0.8	65.8 ± 1.8	61.0 ± 4.1
Traffic Morphing	94.8 ± 0.2	92.8 ± 0.2	90.2 ± 0.3	85.6 ± 0.7	83.3 ± 0.7	77.7 ± 2.3	75.1 ± 3.1

Figure 24: Performance of the VNG++ classifier, for varying values of k, using the Herrmann dataset.

	Privacy Set Size						
Classifier	$k = 16$	$k = 32$	$k = 64$	$k = 128$	$k = 256$	$k = 512$	$k = 775$
None	98.4 ± 0.1	98.0 ± 0.1	97.7 ± 0.2	97.2 ± 0.2	97.2 ± 0.3	96.4 ± 0.5	96.4 ± 0.4
Session Random 255	96.8 ± 0.1	95.5 ± 0.2	93.6 ± 0.2	90.6 ± 0.3	87.2 ± 0.5	83.2 ± 0.5	78.7 ± 0.9
Packet Random 255	97.6 ± 0.1	96.9 ± 0.2	96.2 ± 0.2	94.9 ± 0.3	93.9 ± 0.3	91.2 ± 0.8	90.3 ± 0.7
Pad to MTU	96.4 ± 0.1	94.8 ± 0.2	92.7 ± 0.3	89.8 ± 0.4	86.6 ± 0.5	82.4 ± 0.8	79.2 ± 0.9
Packet Random MTU	97.0 ± 0.1	95.8 ± 0.2	94.4 ± 0.2	92.1 ± 0.3	89.3 ± 0.5	85.6 ± 0.7	83.2 ± 0.6
Exponential	98.1 ± 0.1	97.9 ± 0.1	97.2 ± 0.2	96.6 ± 0.3	95.6 ± 0.4	95.2 ± 0.3	94.6 ± 0.4
Linear	98.1 ± 0.1	97.7 ± 0.1	97.6 ± 0.2	96.8 ± 0.2	95.8 ± 0.5	95.0 ± 0.6	94.2 ± 0.7
Mice-Elephants	97.5 ± 0.1	97.0 ± 0.1	95.6 ± 0.2	94.5 ± 0.3	93.2 ± 0.4	89.9 ± 0.9	88.7 ± 1.0
Direct Target Sampling	94.7 ± 0.2	91.7 ± 0.2	87.2 ± 0.3	81.8 ± 0.5	75.9 ± 0.7	68.7 ± 0.9	62.5 ± 1.3
Traffic Morphing	95.9 ± 0.1	94.2 ± 0.2	91.6 ± 0.3	88.7 ± 0.4	85.6 ± 0.6	81.0 ± 0.9	77.8 ± 1.3

Figure 25: Performance of the Panchenko classifier, for varying values of k, using the Herrmann dataset.

Appendix C Algorithms for Ranking and Unranking a Regular Language

In Figure 26 we have the core algorithms BuildTable, rank, and unrank, used in our FTE record layer. The ordinality of symbol $\alpha \in \Sigma$, written $\text{ord}(\alpha)$, is its position (starting from 1) in the lexicographical ordering of the elements of Σ. $T[q, i]$ is the number of strings of length i that end in an accepting state when starting from state q; thus $T[q_0, i]$ is the number of $X \in L$ such that $|X| = i$. $S[i]$ is the number of strings in L of length at most $i - 1$. Unspecified algorithm FindSlice finds the largest ℓ such that $S[\ell] < c$, and returns $n = \ell+1$ and $c' = c-S[\ell]$. This can be done in $O(\log_2(|S|))$ time via binary search.

```
alg. BuildTable($N$):
for $q \in Q$ do
  if $q \in F$ then $T[q,0] \leftarrow 1$
for $i = 1$ to $N$ do
  for $q \in Q$ do
   for $a \in \Sigma$ do
      $T[q,i] \overset{+}{\leftarrow} T[\delta(q,a), i-1]$
$S[0] \leftarrow 0$
for $i = 1$ to $N$ do
  $S[i] \leftarrow S[i-1]+T[q_0,i-1]$
```

```
alg. rank($X$):
$n \leftarrow |X|$; $c \leftarrow S[n]$
$q \leftarrow q_0$
for $i = 1$ to $n$ do
  for $j = 1$ to $\text{ord}(X[i]) - 1$
do
   $c \overset{+}{\leftarrow} T[\delta(q,a_j), n-i]$
  $q \leftarrow \delta(q, X[i])$
ret $c$
```

```
alg. unrank($c$):
$(n,c') \leftarrow$ FindSlice(c)
$X \leftarrow \varepsilon$; $q \leftarrow q_0$; $j \leftarrow 1$
for $i = 1$ to $n$ do
  while $c' \geq T[\delta(q,a_j), n-i]$
do
      $c' \overset{-}{\leftarrow} T[\delta(q,a_j), n - i]$;
$j \overset{+}{\leftarrow} 1$
  $X[i] \leftarrow a_j$; $q \leftarrow \delta(q, X[i])$;
$j \leftarrow 1$
ret $X$
```

Figure 26: Algorithms for ranking and unranking strings in the regular language L of a DFA $M = (Q, \Sigma, \delta, q_0, F)$.

Appendix D Marionette POP3 Format

In Figure 27 we have the format-specification for languages that mimicks POP3.

```
connection(tcp, 1110):
    start NULL ready 1
    ready B_ready username 1
    username B_uname username_ok 1
    username_ok B_uname_ok password 1
    password B_passwd password_ok 1
    password_ok  B_passwd_ok get_delete_message_retr 1
    get_delete_message_retr B_gdm_retr get_delete_message_msg 1
    get_delete_message_msg B_gdm_msg  get_delete_message_del 1
    get_delete_message_del B_gdm_del get_delete_message_ok 1
    get_delete_message_ok B_gdm_ok get_delete_message_retr 0.5
    get_delete_message_ok B_gdm_ok sign_off_quit 0.5
    sign_off_quit B_sign_off_quit sign_off_ok 1
    sign_off_ok B_sign_off_ok end 1



action_block B_ready:
    server io.puts("+OK My POP3 server ready.\n")

action_block B_uname:
    client io.puts("USER MyUsername\n")

action_block B_uname_ok:
    server io.puts("+OK please send PASS command\n")

action_block B_passwd:
    client tg.send("pop3_password")

action_block B_passwd_ok:
    server io.puts("+OK MyUsername is welcome here\n")

action_block B_gdm_retr:
    client io.puts("RETR 1\n")

action_block B_gdm_msg:
    server tg.send("pop3_message_response")

action_block B_gdm_del:
    client io.puts("DELE 1\n")

action_block B_gdm_ok:
    server io.puts("+OK\n")

action_block B_sign_off_quit:
    client io.puts("quit\n")

action_block B_sign_off_ok:
    server io.puts("+OK MyUsername My POP3 server signing off.\n")
```

Figure 27: The “pop3_simple_blocking" format.

Appendix E Marionette FTP Format

In Figure 28 and Figure 29 we have the format-specification for POP3.

```
connection(tcp, 2121):
    start NULL ready 1
    ready B_ready B_uname username 1
    username B_uname username_ok 1
    username_ok B_uname_ok password 1
    password B_passwd password_ok 1
    password_ok B_passwd_ok pasv_mode 1
    pasv_mode B_pasv pasv_mode_ok 1
    pasv_mode_ok B_pasv_ok ftp_get_file_request 1
    ftp_get_file_request B_gf_req ftp_get_file_response_started 1
    ftp_get_file_response_started B_gf_resp ftp_pasv_transfer 1
    ftp_pasv_transfer B_pasv_xfer ftp_pasv_transfer_ok 1
    ftp_pasv_transfer_ok B_pasv_xfer_ok sign_off_quit 1
    sign_off_quit B_sign_off_quit sign_off_ok 1
    sign_off_ok B_sign_off_ok end 1

action_block B_ready:
    server io.puts("220 My FTP Server.\n")
action_block B_uname:
    client io.puts("USER MyUsername\n")

action_block B_uname_ok:
    server io.puts("Password required for MyUsername\n")
action_block B_passwd:
    client tg.send("pop3_password")

action_block B_passwd_ok:
    server io.puts("230 User MyUsername logged in.\n")

action_block B_pasv:
    client io.puts("PASV\n")
action_block B_pasv_ok:
    server io.puts("227 Entering Passive Mode (127,0,0,1,195,86).\n")

action_block B_gt_req:
    client io.puts("get MyFile.mp3\n")

action_block B_gf_resp:
    server io.puts("150 Data connection starting for MyFile.mp3.\n")

action_block B_pasv_xfer:
    server model.spawn("ftp_file_transfer")

action_block B_pasv_xfer_ok:
    server io.puts("226 Transfer Complete.\n")

action_block B_sign_off_quit:
    client io.puts("quit\n")

action_block B_sign_off_ok:
    server io.puts("221 Goodbye.\n")
```

Figure 28: The “ftp_passive_transfer" format. Spawned by the “ftp_session" format. Needed when a PASV transfer is spawned.

```
connection(tcp, 50006):
    start NULL ftp_pasv_transfer 1
    ftp_pasv_transfer B_xfer end 1

action_block B_xfer:
    server fte.send("ID3.*", 512)
```

Figure 29: The “ftp_file_transfer" format. Spawned by the “ftp_session" format. Needed when a PASV transfer is spawned.