Seeing through Network-Protocol Obfuscation

Liang Wang
University of Wisconsin
liangw@cs.wisc.edu

Kevin P. Dyer
Portland State University
kdyer@cs.pdx.edu

Aditya Akella
University of Wisconsin
akella@cs.wisc.edu

Thomas Ristenpart
Cornell Tech
ristenpart@cornell.edu

Thomas Shrimpton
University of Florida
teshrim@cise.ufl.edu

ABSTRACT

Censorship-circumvention systems are designed to help users bypass Internet censorship. As more sophisticated deep-packet-inspection (DPI) mechanisms have been deployed by censors to detect circumvention tools, activists and researchers have responded by developing network protocol obfuscation tools. These have proved to be effective in practice against existing DPI and are now distributed with systems such as Tor.

In this work, we provide the first in-depth investigation of the detectability of in-use protocol obfuscators by DPI. We build a framework for evaluation that uses real network traffic captures to evaluate detectability, based on metrics such as the false-positive rate against background (i.e., non obfuscated) traffic. We first exercise our framework to show that some previously proposed attacks from the literature are not as effective as a censor might like. We go on to develop new attacks against five obfuscation tools as they are configured in Tor, including: two variants of obfsproxy, FTE, and two variants of meek. We conclude by using our framework to show that all of these obfuscation mechanisms could be reliably detected by a determined censor with sufficiently low false-positive rates for use in many censorship settings.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*

Keywords

Censorship-resistance; network obfuscation; Tor

1. INTRODUCTION

Nation-states and other Internet censors use deep-packet inspection (DPI) to detect and block use of circumvention tools. They do so by recognizing the tools' protocol headers or other telltale fingerprints contained in application-layer content of network packets. In response, researchers and activists have proposed a large number of approaches for obfuscating the network protocol being used. These obfuscation tools can be loosely categorized as either attempting to randomize all bytes sent on the wire [39, 40, 48, 52], attempting to look like (or mimic) an unblocked protocol such as HTTP [12, 26, 45, 46], or tunneling traffic over an implementation of an unblocked protocol [42]. Examples of network obfuscators from each of these three classes are now deployed as Tor pluggable transports [3] and with other anti-censorship tools [44, 48]. Currently, the available evidence indicates that existing DPI systems are easily subverted by these tools [12], and that nation-state censors are not currently blocking their use via DPI [43]. Thus these systems provide significant value against today's censors.

Can censors easily adapt and deploy new DPI algorithms that accurately detect these protocol obfuscators? Houmansadr et al. [16] proposed a number of attacks for detection of mimicry obfuscators, but they do not measure false-positive rates. It may be that their attacks are undeployable in practical settings, due to labeling too many "legitimate" connections as emanating from an anti-censorship tool. What's more, the randomizing and tunneling obfuscators, which are the most widely used at present [43], have not been evaluated for detectability at all. (Despite folklore concerns about possible approaches [42].) In short, no one knows whether these obfuscators will work against tomorrow's censors.

In this work, we provide the first in-depth, empirical investigation of the detectability of modern network protocol obfuscators, and of the collateral damage to real network traffic due to false positives. **Our results suggest that all of the in-use protocol obfuscation mechanisms can be reliably detected by methods that require surprisingly little in the way of payload parsing or DPI state**. See Figure 1 for a summary of our best-performing attacks against Tor's pluggable transport obfuscators.

To obtain these results, we built a trace analysis framework and exercised it with a variety of datasets. Prior attack evaluations [12, 16] have focused primarily on small, synthetic datasets generated by a researcher running a target tool in a specific environment, and evaluating true-positive and false-negative rates of a certain attack. As mentioned above, this may lead to over-estimation of tool efficacy in practice: actual false-positive rates may be prohibitively large, and the synthetic traces used in the lab evaluations may be unlike the network traces seen in real environments. We address both of these issues by employing an experimental methodology that more closely reflects the setting of nation-state level DPI.

We collected nearly a terabyte of packet traces from routers associated to various networks at our university campus. Together these have about 14 million TCP flows. Given the size of the covered networks (five /16 networks and three /24 networks), this represents a large, diverse dataset suitable for assessing false positives. We supplement with researcher-driven traces of obfuscation tools (which do not appear in the traces already), collected across a number of client environments.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/2810103.2813715.

Obfuscator	Type	Attack	TPR	FPR
obfsproxy3	Randomizer	entropy + length	1.0	0.002
obfsproxy4	Randomizer	entropy + length	1.0	0.002
FTE	Mimicry	URI entropy/length	1.0	0.00003
meek-amazon	Tunneling	decision tree	0.98	0.0002
meek-google	Tunneling	decision tree	0.98	0.00006

Figure 1: Summary of best-found attacks against the network obfuscators deployed as Tor pluggable transports. TPR is the true-positive rate; FPR is the false-positive rate as measured using real network traces from a university campus.

Using these datasets, we explore previously proposed DPI-based attacks against obfuscation systems and develop new ones. To begin, we evaluate a collection of *semantics-based attacks*, which attempt to detect a mimicry obfuscator by looking for deviations from expected behavior of the cover protocol. This type of attack was recently introduced by Housmansadr et al. [16]. Next, we explore *entropy-based attacks*, which seek to detect when network packet contents (in whole, or in part) appear to be encrypted. encryption may be in whole, or in specific parts where non-obfuscated flows would not be. Finally, we examine *machine-learning-based attacks*. These use decision trees that are trained from traces of both obfuscated and non-obfuscated traffic. Some of the features used were inspired by suggestions by the meek designers [42], but we are the first to build full attacks and evaluate them. Our investigations reveal that:

- Semantics-based attacks can sometimes have prohibitive false-positive rates (up to 37% in the case discussed in §4.1). False positives here arise because many non-censorship tools deviate from standards in ways similar to mimicry protocols. In these cases they are unlikely to be useful by censors. We also show that other semantics attacks, including one suggested in [16], have relatively low false-positive rates (0.03% as in §4.2).
- Randomizers such as obfsproxy [40], which only emit random payload bytes, are reliably detected by a combination of entropy-based tests and simple heuristics (e.g. length checks). In particular, these tests are applied only to the beginning of the first packet payload. This attack abuses the distinction between conventional protocols' plaintext headers and the lack of same in randomizing obfuscators. In short, having "no fingerprint" is itself a fingerprint (§5.1).
- Format-transforming encryption (FTE), as currently deployed in Tor, is reliably detected by simple tests on the entropy and length of the URI appearing in the first FTE-produced packet (§5.2).
- Tunneling protocols such as meek [42], broadly considered the most secure current proposal for protocol obfuscation, are reliably detected by classifiers trained on traffic-analysis and entropy features (§6). The trained classifiers are simple decision trees, and do require the DPI to maintain state for the first part of a flow.

In summary, our analyses show how to reliably detect all of the currently deployed Tor pluggable transports. For example, our testing reveals that we can detect 100% of obfsproxy3/obfsproxy4 traces using just the first packet in the flow, and with false-positive rate of 0.2%.

A false-positive rate of 0.2% is likely to still be considered prohibitively high for some censors. In particular, since the base-rate of obfuscator-produced traffic is dominated by that of non-obfuscated traffic, the vast majority of flows that are tagged as obfuscated will in fact be false positives. In the near term, false positive rates could be improved via white-listing of specific protocols. In addition, more robust and expensive tests on tagged traffic such as the active probing techniques used by the Great Firewall [51] could be used to achieve a false positive rate of essentially zero. Using our network trace data sets, we give a preliminary analysis of the expected burden such secondary checks would impose on a censor's infrastructure (§6.2). Nevertheless, evidencing negligible false positive rates in realistic deployment settings remains an important challenge for future work.

Summary and discussion. We are the first to provide an in-depth analysis of attacks against modern network protocol obfuscation. We quantify, for the first time, the false-positive rates of previously proposed attacks, as well as ones that we introduce. To do so we use real network traffic like what censors would see in practice. The framework for this analysis, while built mostly off existing open-source tools (e.g., Bro, MySQL, etc.) required significant engineering effort to handle the scale of our analyses. We will open source and make public the framework for other researchers to use. The university network captures cannot be released.

Our results suggest that censors can easily adapt their current tools to efficiently detect state-of-the-art network obfuscators. New obfuscators could, in turn, easily defeat the specific tests that we propose. Under current knowledge, whichever of the censor or protocol obfuscator can adapt to the other will have the upper hand in protocol (mis)classification. Developing practical, network obfuscation mechanisms that are robust to adaptation by the DPI remains a compelling open problem of practical import.

2. BACKGROUND

Censorship. Nation-states and other organizations assert control over Internet communications by blocking connections to websites based on IP address, application-layer content of packets, active probing of remote servers, or some combination of the preceding. For a more fine-grained taxonomy of attacks see [16].

IP filtering is a commonly-used censorship technique. Here the censors monitor a list of IPs and block all communications whose source or destination IP that belongs to the list. IP filtering can be deployed in border ASes or local ISPs by nation-states censors [53]. To bypass IP filtering, a simple method is to use anonymous proxies. By deploying proxies outside the censored network, users within the censored network can submit traffic past censors to the (unfiltered) proxy IP. Various proxy-based censorship circumvention systems have been developed such as freegate, Ultrasurf and JonDo. Tor [11], an onion routing system, helps circumvent IP-based filtering due to its use of a large number of proxies (and bridges). A more recent proposal is to use domain fronting, in which one sends traffic through an unwitting reverse proxy [42].

In addition to filtering by IPs, and partly because of the burden of maintaining an exhaustive and accurate list of proxy IP addresses, censors have increasingly also deployed deep packet inspection (DPI) techniques. This enables censors to attempt *protocol identification*: inspecting the application-layer content of packets (e.g., application-layer headers) as well as packet size or timing information, in order to classify the traffic as generated by communication protocols associated with an anti-censorship tool. In the past, Tor traffic itself contained unique application-layer fingerprints patterns that can be recognized by DPIs [51]. In this work we focus primarily on DPI-based censorship.

A final class of protocol identification attacks uses active probing of a remote server. The Great Firewall of China, for example, is known to attempt Tor handshakes with destination IP addresses

should a DPI test flag a flow to that IP as possibly emanating from a Tor client [51]. We will consider an active attack briefly in §4.2, and also in consideration of filtering out a (passive) DPI test's false positives (§6.2).

Network protocol obfuscation. In response censors' efforts to carry out protocol identification, researchers have developed a number of approaches to protocol obfuscation. In large part, the goals have been to force DPI to misidentify flows of a censored protocol as those of a protocol that is not blocked, or to prevent DPI from recognizing the flows' protocol at all. The latter helps in the case that censors do not block unidentified flows. Suggested obfuscation techniques roughly fall into three categories:

- *Randomizers*: A randomizing obfuscator aims to hide all application-layer static fingerprints, usually by post-processing traffic with an obfuscation step that emits only bits that are indistinguishable from random ones. Examples are Dust [48], ScrambleSuit [52], and the various versions of obfsproxy [40, 54]. The last are currently deployed with Tor.
- *Protocol mimicry*: A mimicry obfuscator attempts to produce traffic that looks to DPI as if it were generated by some "benign" protocol, also called the cover protocol. One example of light-weight mimicry is format-transforming encryption (FTE) [12, 24], now deployed with Tor and implemented elsewhere [44]. It encrypts messages to produce ciphertexts that match regular expressions commonly used by DPI for identifying protocols. Less efficient obfuscators like Stegotorus [46], SkypeMorph [26], CensorSpoofer [45] and Marionette [13] use heavier steganographic techniques to produce messages that look like a cover protocol. Marionette also provides mechanisms to mimic higher-level protocol behaviors, to perform traffic shaping, and to protect against some forms of active attacks.
- *Tunneling*: A logical extreme of mimicry is to simply tunnel data over a (typically encrypted) cover protocol. Intuitively this should provide best-possible mimicry as one is, in fact, using an existing implementation of the cover protocol. An example now deployed with Tor is meek, which uses domain fronting and tunnels traffic over HTTPS connections to popular cloud load balancers such as Google (we will refer to this as meekG) and Amazon (meekA).

As mentioned, several of these are now in-use with Tor as pluggable transports (PTs). A PT is just Tor's terminology for an obfuscator that works in their framework to obfuscate the traffic between Tor clients and bridges [3].

Available evidence indicates that the currently deployed obfuscators are able to circumvent deployed DPI systems [12, 43]. The main question we address is: *Can censors easily adapt their DPI to detect and block obfuscators, without also blocking a significant fraction of non-obfuscator traffic?*

Unobservability. Houmansadr et al. [16] suggest that protocol-mimicry obfuscators will not foil future censors because they do not provide (what they refer to as) complete unobservability. They informally define the latter to be achieved only when a mimicry obfuscator faithfully follows the standards of the target protocol, in addition to imitating all aspects of common implementations. They give a number of DPI-based detection techniques for mimicry obfuscators and show that these have few false negatives (missed obfuscated flows) and high true positives (correctly identified obfuscated flows) using synthetic traffic generated by the researchers. Their attacks target semantic mismatches between obfuscated traffic and the cover protocol, for example checking for valid application-level headers for files, that header values such as length fields are correct, etc.

These semantics-based attacks have not, however, been assessed in terms of false positives: a flow that is labeled as having been obfuscated but was actually not generated by the targeted anti-censorship tool. A high false-positive rate could make such attacks less useful in practice, since it may lead to blocking too many "legitimate" connections or overwhelm systems performing additional checks after the DPI first labels a flow as obfuscated, as in the case of the Great Firewall's secondary active probing mentioned above. In fact, as we mentioned earlier, there are realistic settings in which a false-positive rate that seems small (e.g. 0.2%) may be troublesome for censors.

Thus a question left open by this prior work is: *Do the semantics-based attacks proposed in [16] have prohibitively high false-positive rates?* Clearly a negative answer would help us answer our main question above, but, as we will see, some semantics-based attacks do not work as well as a censor might like.

Threat model, scope, and approach. In the rest of this paper, we focus on answering the two questions just posed. We will adopt the viewpoint of a censor, and attempt to build efficient algorithms that reliably and accurately detect network flows that have been produced by obfuscators. Our primary consideration will be for obfuscators deployed as Tor pluggable transports, since these are in wide use. These are: obfsproxy3 and its successor obfsproxy4, the Tor pluggable transport version of FTE, meek using Google AppEngine proxies (called *meekG*), and meek using Amazon CloudFront proxies (called *meekA*).

A handful of other obfuscators will be considered, although less deeply, in order to address the open question from [16]. For example, we will investigate false-positive rates for the Houmansadr et al. detection attacks for Stegotorus, even though Stegotorus currently is unsupported and unusable with Tor. We are aware that we evaluate only a very small fraction of proposed attacks for other obfuscators, and we would like to investigate more attacks in the future. However, the few attacks we considered give useful insights into developing efficient attacks.

Our study is restricted to the efficacy of existing obfuscators. As such, we assume that the IP address, port number, and traffic features that existing obfuscators do not attempt to hide are out of scope. Network-visible information that one or more existing obfuscators do attempt to hide, including application layer content, packet timing, and packet lengths may be leveraged in attacks.

3. ANALYSIS FRAMEWORK AND DATA

We implement a framework for empirical analysis of obfuscator detection techniques. It will leverage two groups of datasets: the first is a collection of network packet traces collected at various locations at our university at different points in time, and the second is synthetic traffic generated by target obfuscators.

We use our packet traces to examine false positives in existing proposals for attacking censorship circumvention systems, as well as in the new attacks we propose in this paper. We use the synthetic packet traces to study true-positive rates of the new attacks.

We start by providing details of the two sets of data and then discuss the analysis framework.

3.1 Datasets

We use two major types of datasets: (1) packet-level traffic traces collected at various locations in a campus network, and (2) packet-level traces for Tor Pluggable Transport traffic collected in controlled environments.

	OfficeDataset	*CloudDataset*	*WifiDataset*
Collection year	2014	2012	2010
Deployed PTs	Obfs3/FTE/meek	-	-
Size (GB)	389	239	446
Total flow No. (M)	1.89	9.34	13.17
TCP flow No. (M)	1.22	7.48	5.32
TCP-HTTP (%)	37.0	73.6	76.6
TCP-SSL/TLS (%)	45.3	5.4	12.9
TCP-other (%)	0.2	0.2	0.1
TCP-unknown (%)	17.5	20.8	10.4

Table 2: A summary of campus network datasets and breakdowns of TCP flows by services. "TCP-other" are flows with non-HTTP/SSL/TLS protocols. "TCP-unknown" are flows of which protocols are failed to identified by Bro. "Deployed PTs" shows the Tor pluggable transports that had been deployed by the time we collected the traces.

Campus network traces. Over a period of 43 hours between Sep. 6, 2014 to Sep. 8, 2014, we monitored all packets entering or leaving a /24 IPv4 prefix and a /64 IPv6 prefix belonging to our university. In all, this resulted in 389 GB of network traffic with full packet payloads. The networks correspond to two different academic departments within our campus. We call this dataset *OfficeDataset*.

In addition, we employed two other campus network traces, which we call *CloudDataset* and *WifiDataset*. These were collected at earlier points in time. In particular, *CloudDataset* was collected between June 26, 2013 and to June 27, 2013 (over a period of 24 hours), and contains all traffic recorded between our entire campus network and the public IP address ranges published by EC2 and Azure. The dataset *WifiDataset* constitutes all packets captured over a period of 12 contiguous hours in April 2010 from roughly 1,920 WiFi access points belonging to our campus. It contains data exchanged between all wireless clients (e.g., laptops and smartphones) connected to the campus wireless networks and other (internal or external) networks. A summary of these three datasets is shown in Table 2.

The most recent trace could, hypothetically, contain flows corresponding to actual use of Tor with the obfuscators turned on. In our analyses, we ignore this possibility and assume that all traffic is non-obfuscated. Given that these flows are only used to assess false positives (FPs), our assumption is conservative when we argue the FP rates are low.

These traces contain potentially sensitive information of network users. We obtained an IRB exemption for these analyses. We performed analysis on an isolated cluster with no connectivity to the Internet, and with suitable access controls so that only approved members of the research team were able to use the systems. Only the bare minimum of research team members were approved to access the machines.

Tor traces. A Tor traffic trace captures the network traffic exchanged when a client visits a website over Tor configured to use a specific obfuscator. To collect a trace, we follow the procedures for collecting traces for website fingerprint attacks as described in [19]. We built a framework to automate these procedures. Our framework uses the Stem Python controller library for Tor, and the Selenium plugin for automating control over a Firefox Browser when visiting websites [37, 41].[1] We record all traffic using *tcpdump* (or *WinDump* on Windows) at the same time.

[1]Also, our Firefox browser uses the exact same profiles as the default browser in the Tor Browser bundle (TBB) 4.06. The versions of obfuscators and Tor we used are also the same as those being used in TBB 4.06.

Before and after visiting a website, our framework visits the "about:blank" webpage and dwells there for 5–15 seconds. The first time this is done is to ensure that the obfuscator connection is fully built; in our experiments, we found that most obfuscators forced a few seconds (usually less than 5 seconds) delay when building connections after Tor starts successfully. The second visit to "about:blank" is for making sure we can capture any lingering packets.

We collected three sets of Tor traces under different combinations of network links (with different capacities), end-host hardware, and operating systems, and labeled them as *TorA*, *TorB* and *TorC*. We collected *TorA* and *TorB* on Ubuntu 12.04 (32-bit) virtual machines (VMs) and *TorC* on a Windows 7 (32-bit) virtual machine. The Ubuntu VMs are built on the same image. All VMs run on VirtualBox 4.3.26 and are configured with 4G RAM and 2 virtual processors. The VMs for *TorA* run on a workstation and are connected to a campus wired network, whereas the VMs for *TorB* and *TorC* are run on a laptop and connect to a home wired network,

Each of these three datasets contains 30,000 traces collected as follows: (1) For each target obfuscator, we used our trace collection framework to visit Alexa Top 5,000 websites to collect 5,000 traces (labeled as *obfs3*, *obfs4*, *fte*, *meekG*, and *meekA*, corresponding to obfsproxy3, obfsproxy4, FTE, meek-google, and meek-amazon respectively); (2) In addition, we visited the same set of websites without Tor and obfuscators to collect 5,000 traces and labeled them as *nonTor*.

A handshake message of a flow is the application-layer content of the first client-to-server packet in the flow. We extract 5,000 handshake messages from each of obsproxy3, obfsproxy4, SSL/TLS, HTTP, and SSH flows to construct a new dataset. The first two types of flows are sampled randomly from Tor datasets and the other types of flows are from unused campus network traces (recall that we only use a part of the collected campus network traces to construct the campus datasets). We call this dataset *HandShakeDataset* and use it when examining attacks based (only) on handshake messages.

3.2 Trace Analysis

We use Bro 2.3.2 [31] with the "-r" option to analyze the collected network traces, and format and store the results into MySQL tables. Each table corresponds to a ".log" file generated by Bro, and it stores the information for flows of a given type (UDP, HTTP, SSL, etc.). Each flow is assigned an unique flow ID, which is also generated by Bro. Also, for each trace packet, we compute an MD5 hash to generate a packet ID, and store the packet ID, the flow ID of the associated flow, the raw packet content (in hexadecimal) and the packet timestamp into an Apache Hive database [38]. The usage of Hive facilitates the management and processing of terabytes of data. Users only need to query the MySQL database to get the basic statistics of a trace, while using Hive for more time-consuming, sophisticated analysis such as analyzing packet payloads.

We develop a set of APIs to analyze the above data. These APIs are encapsulations of Hive or MySQL queries. For some simple analyses (e.g, counting the packets with a given keyword in the payload), pure Hive queries are enough. To facilitate more complex analysis and provide more flexibility, we leverage User-Defined Functions (UDFs) in Hive to allow users to provide their own mapper/reducer scripts [2]. We plan to release all our scripts publicly for other researchers or activists to use.

	Standard	Malformed	Partial	Other	Total
OfficeDataset	26 (89.7)	0	3 (10.3)	0	29
CloudDataset	4,293 (62.8)	1,313 (19.2)	338 (4.9)	895 (13.1)	6,839
WifiDataset	1,860 (46.7)	1,252 (31.5)	572 (14.4)	295 (7.4)	3,979
Total	6,182 (57.0)	2,565 (23.6)	913 (8.4)	1,190 (11.0)	10,847

Table 3: Breakdown of PDFs by their categories. The percentages of all PDFs found are shown in parentheses.

4. SEMANTICS-BASED ATTACKS

We seek to determine whether in-use obfuscators can be reliably detected by censors. The starting point is previously proposed attacks. As described in §2, Houmansadr et al. suggest a variety of attacks against mimicry obfuscators. For example, a Stegotorus client may generate invalid PDF documents, whereas legitimate traffic presumably does not. Their attacks therefore use the deviations of a target system from expected behavior as evidence for detecting mimicry obfuscators. We call these attacks *semantics-based attacks*.

In this section, we evaluate three semantics based attacks, two proposed by Houmansadr et al. against Stegotorus and one suggested by Dyer et al. [12] for detecting FTE. None of these attacks have been evaluated in terms of false positives, and the latter has not received any analysis at all. Looking ahead, we find that the first attack is unlikely to work well in practice due to the high false-positive rates we discover. The other two work better, but have deficiencies that our later attacks avoid.

4.1 Stegotorus PDF attack

Description. The Stegotorus HTTP obfuscator attempts to hide Tor traffic in commonly-seen documents, such as PDF and JavaScript. However, StegoTorus-HTTP does not guarantee the semantic correctness of the generated file. The authors in [16] proposed an attack that can detect files (more specifically, PDF files) generated by StegoTorus at a low cost and line speed. The key idea is to check the validness of the *xref* table in a PDF file.

Clearly the efficacy of this attack relies on the assumption that PDFs generated by non-obfuscated traffic, e.g., normal HTTP, indeed has valid *xref* tables. This was not evaluated in [16], presumably due to a lack of access to real traffic.

For an HTTP flow, we use a Python library pyndis to reassemble HTTP sessions. If the *Content-Type* in the response of a session is "application/pdf", we assume the response body is a PDF file. We extract the content from the response, store the content as a PDF file, and use the PyPDF2 library to test the semantic correctness of the file. We define four categories of PDFs:

- Standard: According to [17], a standard PDF file should start with "%PDF" and end with "%%EOF", and have a *xref* keyword in the content. The PDFs in this category have these keywords and also pass the PyPDF2 semantic check.
- Malformed: The PDFs have "%PDF", "%%EOF" or *xref* keywords (at least one keyword), but do not pass the PyPDF2 semantic check.
- Partial: A request may have a *Range* field that specifies the parts of a PDF file the client wants (e.g., only request the first 200 bytes of a file). The PDFs carried by the HTTP response of which status code is "206 Partial Content" is "partial PDF". The PDFs in this category will not pass the semantic check because they are only parts of the original PDFs.
- Other: All PDFs that are not in the aforementioned three categories. The PDFs in this category do not pass the semantic check.

Request	None	200	3xx	4xx	5xx	Other
GET long	3	12	< 1	**83**	2	0
GET non-existing	2	18	< 1	**79**	1	< 1
HEAD existing	2	**95**	< 1	2	1	< 1
OPTIONS common	3	**82**	< 1	13	2	< 1
DELETE existing	3	72	< 1	**21**	**4**	< 1
TEST method	4	66	< 1	**19**	**10**	< 1
GET wrong protocol	24	10	33	**28**	**5**	< 1

Table 4: Percentage of Alexa top 10 K servers that return a given type of response for each type of request (rounded to the nearest whole percent). The bolded entries indicate the standard response(s) for a given request type.

We find in our tests these four categories are mutually exclusive. Only the PDFs in *Standard* can pass the PyPDF2 semantic check, and the PDFs in other categories could not pass the test for different reasons.

Results. We performed the test described above on the *OfficeDataset*, *CloudDataset* and *WifiDataset* university capture datasets. As shown in Table 3, the false-positive rate can be as high as 43% across all the datasets, should the censor mark any non-standard PDF flow as Stegotorus, as per the suggestion in [16]. More restrictive checks would still have high false-positive rates.

To explain these results, we observe that there are many reasons that non-obfuscator PDFs fall into *Malformed* or *Other* category. For instance, the file is encoded or encrypted, or there are bugs in the PDF generation software [21]. Partial content is also widely used by browsers and applications. As an example usage, FireFox will build several connections to fetch different parts of a single resource with range request in parallel, and reconstruct the resource by itself. The PDF content in each connection is incomplete, so a censor could not directly do semantics checks. A similar technique is called *Byte Serving* [22]. An application can retrieve the necessary portion of a file instead of the entire file. For example, a byte-serving-enabled PDF viewer can request and load a large PDF file from a file server page by page ("page on demand"). If a user only reads a few pages, the censorship system will never see the whole file.

4.2 HTTP response fingerprinting attack

Description. In [16], the authors suggest that a censor can fingerprint a StegoTorus server by observing the server's reactions to different types of HTTP requests, since the StegoTorus server would respond differently from a genuine HTTP server. Of course HTTP servers in practice may have various implementations and it's possible that the HTTP servers used by real websites have the same HTTP-response-based fingerprint as the StegoTorus server. This would lead to false positives.

We reverse-engineered *httprecon*, the tool used for fingerprinting HTTP servers in [16]. We wrote our own version, a Python script that sends the same set of requests. One exception to this is that we omitted a particular request that is often viewed as an attack by server operators, as we used the scanner with public servers and therefore could only send non-malicious requests.[2] We used our implementation to scan each of the Alexa top 10 K domains to quantify false-positive rates. For each target server, we stored its response header for each request into a MySQL database for analysis. We removed the servers that fail to respond to the *GET existing* request or respond with non-200 code, which may indicate the corresponding websites are down, and got 9,320 servers.

[2]This was not an issue in [16] as they did not apply their scanner to public servers.

Results. Table 4 shows the percentages of servers that return a given type of response for each type of request. The responses are put into six categories, with boldface indicating the *standard* response (as described in [16]) for a given type of request. Since *GET existing* requests always receive a "200 OK" for the servers examined, we remove the corresponding row from the table. We can see that for some types of requests, a majority but not all of the target servers return standard responses. Only 73 servers (0.8%) respond like a "standard" server.

Examining the response headers more closely, we observe that the target servers do not follow standards in various aspects (other than the response code):

- For a *GET existing* request, a server should set the *Connection* field in the response to *keep-alive*. We find of the response headers of all the servers, 1,547 (17%) don't have the *Connection* field, 6,503 (71%) are *keep-alive* and 1,126 (12%) are *Close*.
- For a *TEST method* request, a server should set *Connection* field to *Close*. However, we find 1,408 (15%) responses have no *Connection* field, 5,572 (61%) are *keep-alive*, 1,823 (20%) are *Close*, and 378 (4%) fail to respond.
- For an *OPTIONS common* request, a standard server should set the supported HTTP methods in the *Allow* line. We find 8,026 (87%) the responses don't have this field, only 844 (9%) return the standard supported methods (as defined in RFC), 45 (0.5%) return non-standard methods, and 266 (3%) fail to respond. We observe a total of 36 unique non-standard methods.

We find only three servers that have the same HTTP-response fingerprint as Stegotorus, suggesting that the false-positive rate of this active attack is quite low (or 0.03%).

We note that there are a total of 1,447 unique HTTP-response fingerprints. About 40% of these fingerprints are shared by more than 2 servers. The most-seen fingerprint is shared by 845 servers (9% of all the servers examined). Looking ahead, one might therefore update the Stegotorus server to have the same fingerprint as these servers, which will cause a false-positive rate of close to 10% and, moreover, flag network flows associated with 5 of the top 100 domains and 74 of the top 1 K domains.

4.3 FTE content-length attack

Description. One possible approach for detecting FTE, as discussed by the authors of [12], is to check the correctness of the *Content-Length* fields of an HTTP message (HTTP request or HTTP response), since the HTTP message generated by the currently used version of FTE has an invalid *Content-Length* field that is mismatched with the real length of the content.

Results. For an HTTP flow, we use the same technique as we used in the PDF attack to reassemble HTTP sessions. Then, we check if the message has the *Content-Length* field in a message, calculate the length of the message body if it has the field, and compare the calculated length with the length specified in the *Content-Length* field. The false-positive rate is defined as the number of sessions with incorrect content length over the total number of sessions (in percentage). The false-positive rates are 1.86%, 1.95% and 3.5% for *OfficeDataset*, *CloudDataset*, and *WifiDataset* respectively.

Further examining the false positives, we find 34.2% of them are caused by early terminated connections, and 6.8% are caused by Transfer-Encoding fields (which trumps any existing Content-Length fields). More sophisticated checks could remove these false positives. There are other reasons for such mismatches, including extra control bytes added in the message body by some versions of web browsers; non-ASCII characters in the content-body, and bugs in web applications [15, 25].

4.4 Discussion

The semantics-based attacks we analyzed are relatively costly in terms of performance, because they require flow reconstruction (in the first and last case) or active probing (in the second case). The active probing and FTE attacks have arguably low false-positive rates, whereas the first PDF analysis has a clearly prohibitively large one. Based on our analysis of the three semantics-based attacks, we suggest that semantics-based attacks should account for the noisy, non-standards-compliant nature of the web.

5. ENTROPY-BASED ATTACKS

Entropy-based analyses have been used for various purposes such as randomness testing, network anomaly detection, and traffic classification [29,35,36,55]. As traffic generated by the obfuscators are encrypted, one expects them to have noticeably higher entropy than conventional, unencrypted protocols (e.g., HTTP). Prior works show that entropy tests can be effectively used to detect and cull encrypted or compressed packets from network streams [47]. Their goal was to speed up DPI analyses by avoiding such opaque packets. One might expect that similar techniques could work here, but we will need to adapt them to our setting of obfuscator detection.

The obfsproxy methods directly apply encryption to every transmitted message, thereby "randomizing" even the first messages sent. On the other hand, conventional encryption protocols like TLS (or HTTPS) use handshake messages that typically contain unencrypted data from small, fixed sets of strings. Thus, the entropy of initial messages may provide a reliable way to distinguish obfsproxy messages from normal traffic. FTE also employs encryption from the start, although ciphertexts are designed not to look randomized. Nonetheless, the initial messages produced by the FTE obfuscators (as currently deployed in the TBB) are HTTP GET messages containing URIs that directly surface random-looking bytes from an underlying encryption scheme. So both the obfsproxy methods and FTE may admit detection by entropy-based tests on their initial messages. We explore this conjecture in a moment. First, let us explain the statistics we will use.

Shannon-entropy estimator. Let $X = x_1 x_2 \cdots x_L$ be a string of L bytes, i.e., each $x_i \in \{0, 1, \ldots, 255\}$ (where we map between bytes and integers in the natural way). Let n_j be the number of times that the value j appears in X, and let $p_j = n_j/L$. Then our estimate of the (byte-oriented) Shannon entropy is computed as $H(X) = -\sum_{i=0}^{255} p_i \log_2 p_i$. Notice that the maximum value of $H(X)$ is $8 = \log_2 256$, and this occurs when $p_i = 1/256$ for all i. That means that the payload string X contains every symbol in $\{0, 1, \ldots, 255\}$ an equal (positive) number of times. Furthermore, we note that a string with only printable ASCII characters will never have an entropy more than $6.6 = \log_2 95$.

The entropy-distribution test. The traditional Kolmogorov-Smirnov (KS) two-sample test provides a tool for deciding whether or not two sets of samples were drawn from the same distribution. We use it to help us detect high-entropy byte sequences. As above, let X be a string of L bytes. Fix an integer *block size* $k > 0$, define $B = \lfloor L/k \rfloor$, and write $X = S_0 \,||\, S_1 \,||\, \cdots \,||\, S_{B-1}$, where each block S_i is length k bytes (or possibly less for S_{B-1}). We will treat the sequence $H(S_0), H(S_1), \ldots, H(S_{B-1})$ as B samples from some unknown distribution D_k over entropy values. The second distribution we consider is the distribution of entropy values

when the underlying strings are uniformly random. We denote this distribution by $\tilde{D}_k$, and a method for calculating $\tilde{D}_k$ is given in [47]. As shown, the distributions under consideration are parameterized by the block size k, whose value we will address later.

For our *entropy-distribution test*, we compute the two-sample KS-test[3] for $H(S_0), H(S_1), \ldots, H(S_{B-1})$ and an equal-length sequence of samples from $\tilde{D}_k$, and observe the p-value of the test. If the p-value is smaller than our chosen significance-level of $\alpha = 0.1$, then the null hypothesis that $D_k = \tilde{D}_k$ is rejected, and we conclude that the bytes are not uniformly distributed. Otherwise, the test does not provide statistically significant evidence for rejecting the null hypothesis, and we conclude that the bytes are uniformly distributed.[4] Our choice of $\alpha = 0.1$ was based on it being commonly used as a less restrictive (than 0.5) significance level in hypothesis testing in other settings. Higher α would label flows as uniform less often, lower α the opposite. Looking ahead, our experiments show that many choices of α around 0.1 would achieve the same obfuscator detection accuracy.

5.1 Detecting Obfsproxy

We start by showing how to use the entropy-distribution test to detect that a network flow corresponds to one of the two obfsproxy methods. (Our tests currently make no effort to distinguish between obfsproxy3 and obfsproxy4.) As discussed above, we will make use of the observation that randomizers like obfsproxy3 and obfsproxy4 encrypt all bytes in the first message of a flow. Conventional encryption protocols such as TLS and SSH do not: they have fixed plaintext headers. We therefore apply the entropy-distribution test to the first 2,048 bytes of the first client-to-server packet's application-layer contents. If the length of a first-seen content is less than 2,048 bytes, we will just test on all the bytes of the content. If the entropy-distribution test applied to these bytes indicates that they are uniformly distributed we flag the flow as obfsproxy.

First, based on the protocol specifications, the minimum payload length for obfsproxy3 is 193 bytes. For obfsproxy4 it is 149 bytes. Thus, we will check payload lengths for the first packet and require that the length be at least 149 bytes to be labeled as obfuscated. We will couple this check with an entropy-based test.

We experiment with the entropy distribution test described above for block sizes $k \in \{4, 8, 16, 32, 64\}$. We also considered two kinds of alternative tests. First is the byte uniformity test, in which we use a KS test with null hypothesis that the payload byte values are uniformly distributed (which is different than the entropy distribution test described above, where the null hypothesis is that the distribution of entropy values match closely those that one would get form a uniform distribution). We also consider the truncated sequential probability ratio test (SPRT) from [47] applied to payload byte values. This was used by [47] to detect "opaque" traffic (encrypted or compressed), and was shown to be more powerful than the (plain) entropy-test in their setting. Please see [47] for details of that test. We consider it for block sizes of 8, 16, and 32 bytes. This gives us a total of nine tests, the entropy distribution test with the five distinct block sizes, the byte uniformity test, and the SPRT tests with three different block sizes. In all cases we apply the payload length checks mentioned already.

[3]Using `scipy.stats.ks_2samp` [18].

[4]Technically, accepting the null hypothesis in this way should only be performed when a test has sufficient power. As we will see using the test in this way performs well, and so we dispense with a power analysis.

To evaluate these tests we proceeded as follows. We use stratified random sampling (in our case we sample the same number of traces of each of the type of traffic) to select a 30% fraction of the traces from *HandShakeDataset* to construct a validation set and use the remaining 70% as a test set. Recall that *HandShakeDataset* consists of 5,000 handshake messages for each of obsproxy3, obfsproxy4, SSL/TLS, HTTP, and SSH flows, meaning that we have both obfuscated and un-obfuscated traces represented. We discuss false positives on the university data sets below.

We use the validation set to pick the best performing of the nine tests, and then analyze the test's performance on the test set. We repeat this procedure 10 times with fresh random splits of the test sets. In all 10 iterations the best-performing test on the validation set is the entropy distribution test with block size $k = 8$. Table 5 shows the results broken down by protocol types and including for comparison the performance of some of the other tests on the test set. The results are strong: the entropy distribution test with $k = 8$ had only one false positive and zero false negatives in the worst case. We note that the payload length checks are critical here, we showed that SSL/TLS flows had a false-positive rate of roughly 23% on average when one omits the check. With the length checks in place, however, the p-values computed during the entropy distribution test for SSL, HTTP and SSH handshake messages are always near zero (a definitive rejection of the null hypothesis), whereas the median p-values for the flows declared to be obfsproxy is 0.99. Thus we consider the tests decisive.

False positive analysis using campus traces. The above analysis of false positives was in a closed-world setting with just the three protocol types (SSH/HTTP/TLS). We take the best test from above and evaluate false positives against the network traces seen from the university data sets. Specifically we apply the entropy distribution test (with $k = 8$) on the first packet of every TCP flow in *OfficeDataset*, *CloudDataset*, and *WifiDataset* (assuming the packet is non-empty). We found 3,998 (0.33%), 19,247 (0.25%), and 12,786 (0.24%) false positives, respectively.

According to Bro, the 36,031 false-positive flows were distributed as follows: 18,939 were SSL/TLS, 9,873 were HTTP, and the remaining 7,219 flows were reported as "unknown". We applied nDPI [28], an open source DPI that can detect hundreds of application protocols, to the unknown flows. It was able to classify 1,673 (23.1%) of these as follows: 1,275 were Real Time Messaging Protocol (RTMP), 92 were SMTP, 14 were SSH, and the remaining were generated by other various applications.

The false-positive SSL/TLS flows account for 1.22% of all SSL/TLS flows examined, while the false-positive HTTP flows account for 0.10% of all HTTP flows examined. We find the SSL/TLS false positives are associated with 1,907 unique destination IPs, and 90% of these false positives are associated with only 685 unique destinations. Therefore, a large fraction of these false positives could be caused by specific server-side settings. Upon further examination of the destinations in the latter mislabeled HTTP flows, we find just 357 unique servers. The URLs of these servers contain only printable ASCII characters but have the same entropy profile as a random string. The false-positive SSH flows all have encrypted handshakes, which suggests they may use obfuscated SSH [23]. The handshake message of RTMP consists of a 1-byte protocol version and a 1536-byte random string, causing our embellished entropy test to label all RTMP flows as positive.

5.2 Detecting FTE

As currently implemented in the Tor Browser Bundle, the first message in an FTE flow is an HTTP GET request. The URIs in these requests also have large Shannon entropies because

	FNR		FPR		
	Obfs3	**Obfs4**	**SSL**	**HTTP**	**SSH**
Entropy dist. test	0	0	0.02	0	0
Uniform byte test	5.1	5.0	0	0	0
SPRT (8-byte)	60.2	58.3	30.6	0	0
SPRT (16-byte)	24.4	24.8	0.06	0	0
SPRT (32-byte)	6.6	6.9	5.3	0	0

Table 5: A comparison of the average false-positive/false-negative rates (in percentage) of different tests for obfsproxy handshake message detection, across ten rounds of randomized validation/test splits. While the entropy distribution test (only shown for $k = 8$) was chosen in all ten rounds, we show the test-set performance of the other style of tests for comparison. All these tests are performed with the payload length check.

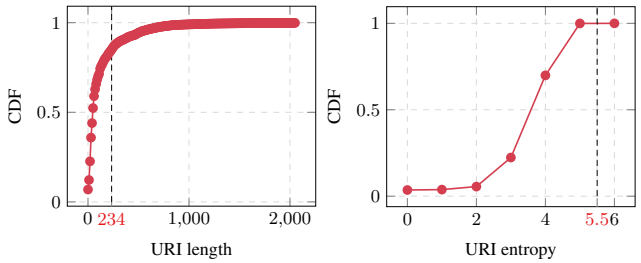

Figure 6: CDF of lengths (**left**) and entropies (**right**) of all URIs extracted for three campus network datasets.

their characters are direct encodings of ciphertext bits into bytes. Estimating the entropies of all URIs produced by FTE in 30% of traces (randomly sampled) in *HandShakeDataset*, we find a relatively narrow range of 5.5 to 5.8 bits. In *HandShakeDataset*, for non-FTE HTTP GET messages, the URIs always have entropies of less than 5.1 bits. This leads to a simple *URI-entropy test*: if the URI in the first GET request of a HTTP flow has entropy of at least 5.5 bits, then we label the flow FTE.

Unsurprisingly, this test produces zero false negatives and zero false positives against *HandShakeDataset*. We also evaluated the URI-entropy test against *OfficeDataset*, *CloudDataset*, and *WifiDataset*. For an HTTP flow that contains GET requests, we extract the URI from the first of these, and perform the URI-entropy test on it. We found a total of 420,795 URIs are mislabeled as FTE, or about 4% of the roughly 10 M URIs examined.

To sharpen our test, we observe that the URIs in the HTTP requests generated by FTE have a constant length of 239 bytes. So we embellish our URI-entropy test so that positively labeled flows must also have a URI length of exactly 239 bytes. This significantly reduces the false-positive rate: only 55, 167, and 42 non-FTE flows are mislabeled as FTE in the *OfficeDataset*, *CloudDataset*, and *WifiDataset* datasets, respectively. That is a total of just 264 false-positives out of around 10 M samples, giving the embellished test a very small false-positive rate.

Note that the above tests can in fact be implemented without message reconstruction, since the tests only involve data in the first packet at fixed locations.

We note that [8] suggests using FTE URI length alone for detection. But we find that a length-only test causes about a 15% false-positive rate over the same 10 M flows. The CDFs of URI lengths and entropies are given in Figure 6.

5.3 Discussion

Our results show that entropy-based tests, embellished with simple length heuristics, can accurately detect obfsproxy3/4 and FTE with relatively low false-positive rates on real network traffic. White et al. [47] show that slightly simplified versions of our entropy-based tests can be implemented on commodity hardware, making them applicable at enterprise scale. The main simplification is that their tests make use of a fixed-size string, whereas ours use up to one full TCP payload. We have not yet considered the efficacy of our tests using truncated payloads.

We note that our entropy-based tests do not perform well at detecting meek-generated traffic, because meek tunnels over HTTPS. To detect meek one will need more sophisticated attacks, as shown in the next section. Finally, we point out that the entropy-based tests in this section are far less resource-intensive than the semantics-based attacks of §4. Those attacks in most cases require flow reconstruction, and these do not.

6. ML-BASED ATTACKS

Using machine learning (ML) for traffic classification is not a novel idea [1, 5, 49], but we are unaware of any prior work that has attempted its use in detecting obfuscated traffic. In this section, we develop attacks using ML with carefully selected features.

Our classification model will be binary, with output labels "positive" (obfuscated) and "negative" (not obfuscated). During training, we will label all training traces that were generated by a targeted obfuscator with "positive", and all other traces as "negative". Thus we will have a classifier for each obfuscator that aims to distinguish a given obfuscator from all other traffic. When attempting to detect the use of any Tor pluggable transports, one could run all of our tests in parallel (one targeting each obfuscation method) and if any of them returns "positive" then the meta-decision would be that Tor with some pluggable transport is in use.

Before describing the actual classifiers, let us discuss the features that they will use, training considerations, and the details of how we evaluated their efficacy.

Features. Our ML-based attack will make use of three types of features. The first are *entropy-based features*. For a given flow, we calculate the Shannon entropy of every packet payload, and use the maximum/minimum/average entropies in packets in each direction (upstream or downstream) as features. The selection of this feature is inspired by the results of the entropy-based tests in §5.

The second feature type is *timing-based features*, which are based on an observation regarding traffic patterns in meek. According to the documentation, meek clients will send packets to a meek server periodically, to check if the server has data to send. This results in timing patterns in the TCP ACK traffic of meek connections that differ from typical TLS connections [42]. To capture this as a feature, we calculate the intervals between two consecutive TCP ACK packets (in the same direction) in a flow, and group the intervals into 30 bins. Based on our observations of meek traffic, the intervals are usually very small: 55% of them are less than 10 ms, and 99.9% are less than one second. So we limit the maximum considered ACK interval to one second. We use $(x, y]$ to denote a bin of width $y - x$, accepting all numbers r such that $x < r \leq y$. (All numbers are milliseconds.) Across our 30 bins, the widths are as follows: between 0 and 10 the bin width is 1 (i.e., $(0, 1]$, $(1, 2]$, etc.), between 10 and 100 the bin width is 10, between 100 and 1000 the bin width is 100, and the final bin is $(1000, \infty]$. We use variable-width bins to balance timing granularity and computational efficiency, and because the interval distribution is skewed towards lower values. Instead of using absolute numbers, our per-bin feature is the percentage of inter-ACK intervals that fall into a given bin.

Finally, we consider *packet-header features*. The first of these is the percentage of TCP ACK packets sent in each direction. This is motivated by the behavior of meek just discussed; we expect to see a larger-than-normal number of TCP ACK packets in meek flows. The second follows previous works (e.g. [7, 30]) by exploiting packet length as a distinguishing characteristic. For a given flow, we use the five most common packet-payload lengths (in each direction) as a feature.

Windowing strategies. We will have two main goals for our classifiers. The first is to accurately detect all of the obfuscators under consideration. The second is to detect obfuscated traffic as soon as possible within a flow, which helps to minimize the amount of state required. Based on our results for the entropy-based tests, we can hope to classify some obfuscated flows based solely on the first message. For meek, however, this seems unlikely to work because meek leverages genuine TLS connections. So, to accommodate both goals, we allow the classifier to train over a window of flow traffic. Inspired by [6, 20, 27], we consider two windowing strategies:

(1) The classifier inspects the first X packets of a flow (including SYN and ACK packets), and extracts features from these packets to do classification.

(2) The classifier extracts features from the packets within the first X seconds of a flow to do classification.

Examining the distributions of durations and sizes (number of packets in a flow) of all obfuscator traces collected, and considering that we want the classifier to be able to make a decision as soon as possible, we limit the range of X in our tests as follows. For the packet-count strategy, $X \in \{30, 35, 40, \ldots, 300\}$, and for the time-based strategy $X \in \{2, 3, 4, \ldots, 10\}$. Choosing the endpoints for the packet-count strategy, we can hope to classify between 90 and 99 percent of obfuscator flows before they terminate; simulating online classification, as opposed to after-the-fact. The same reasoning applies to the boundaries for the time-based strategy.

Details of classifier evaluation. Our initial measure of classifier performance uses the *TorA*, *TorB*, and *TorC* datasets. Recall that these include both synthetic Tor traces under each of the pluggable transports, as well as synthetic traces for SSL/TLS and HTTP.

For any given dataset, we perform nested cross validation with ten outer folds and ten inner folds. We measure performance for that dataset by taking an average over ten outer folds. In an outer fold, we perform a stratified random sampling of the target dataset to select 70% of the traces to create the test set, and use the remaining 30% for training and validation of classifiers for the inner folds.

Each inner fold is as follows: First, we perform a second stratified random sampling, taking a one-third for training and leaving the remainder for validation.[5] Next, we fix a classification strategy by choosing: classification algorithm (K-nearest neighbor, Naive Bayes, or CART), windowing strategy (number of packets or time for each allowed X), and feature set (any single feature, any pair of features, or all features). There are 1,344 different classification strategies, in total. For a given strategy, we train five classifers, one for each of the obfuscators. Each of these is tested on the validation set, and we record the average performance of the five classifiers. This gives an average measure for a particular classification strategy.

We pick the parameter combination that results in the classifiers that perform best on average across all inner folds. Then, using the "winning" classification strategy, we finally train a new classifier using *all* traces from the training/validation portion. This final classifier is what is tested on the test set.

As mentioned, we repeat the training and testing for 10 randomized 70-30 outer folds, and we will report the averages over these folds in a moment. In addition to reporting true-positive and false-positive percentages, we will report area under the precision-recall curve (PR-AUC) (c.f., [10]). A higher PR-AUC indicates a higher true-positive rate and lower false-positive rate. Specifically, a classifier with PR-AUC equal to one is perfect: it gives only true positives with no false positives. We calculate PR-AUC using the scikit-learn tool [9].

Looking ahead, we will sometimes also report on the average result of testing classifiers not chosen by the training/validation regime in order to understand the benefit of using, e.g., some particular feature.

6.1 Results

We will first analyze the performance of ML-based attacks using the synthetic datasets, and then present the false-positive rates seen on the campus traces. To summarize, the best classifier achieves a high average PR-AUC (0.987), a high average true-positive rate (0.986), and a low average false-positive rate (0.003), across all five obfuscators as measured on the same synthetic data set it is trained on. The classifiers all perform significantly worse when tested on a synthetic dataset for which they were not trained. Finally, the highest false-positive rate of any classifier as measured on the campus datasets (none of which were available during training) is 0.65%.

Classifier parameters. Using *TorA*, we found that the best-performing classifiers were essentially always CART decision trees using the packet-count windowing strategy. The best classifiers used between 280 and 300 packets, which we consider too large for practicality. However, classifiers using up to 30 packets already perform within 0.3% PR-AUC of the best performing, and so we from now on restrict our attention to them.

Feature performance. We next discuss how the different types of features effect classifier performance. Recall that we used entropy-based, packet-timing, and packet-header features. In Figure 7 we compare classifier true positive and false positive rates when testing specific combinations of features for the *TorA* dataset. We can see that using only entropy-based and/or packet-header features can already achieve high true positives and low false positives, with pretty low variance across folds (indicated by the error bars). Timing-based features showed a higher false-positive rate, and we conclude that a combination of entropy-based and packet-heading features performs best. Our training procedure indeed always selected the combination of the entropy-based and packet-header features.

Unexpectedly, entropy-based features work well for detecting meek. Examining the number of packets with non-zero-byte payload in the first 30 packets of the meek traces and the SSL/TLS traces in the Tor datasets, we find 70% of SSL/TLS flows have more than 18 packets with non-zero-byte payload, whereas at least 96% of the meek traces have less than 18 packets with non-zero-byte payload. So the number of samples for calculating entropy statistics in a meek trace and a SSL/TLS trace are often different, which biases the minimum, average, and maximum entropy scores

[5] There are perhaps more standard choices for the split sizes, but we do not expect different choices to significantly impact results. The large test set size will tend to produce conservative estimations of our classifiers' performances.

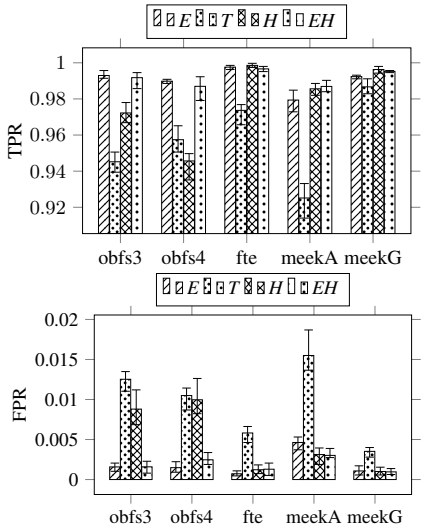

Figure 7: A comparison of true-positive (top graph) and false-positive (bottom graph) rates by features used. "E" indicates entropy-based feature, "T" the timing-based features, "H" the packet header feature set, and "EH" indicates a combination of entropy-based and packet-header features. Note that for clarity the graphs have truncated y-axes.

used as features. This bias is caught by our classifier to differentiate between meek and SSL/TLS.

Portability. We now turn to testing the "portability" of this ML approach. We used the two additional data sets *TorB* and *TorC*, which are collected in environments distinct from that of *TorA*. Whereas *TorA* is collected on an Ubuntu VM connected to a campus network, *TorB* is an Ubuntu VM connected to a home wireless network and *TorC* is a Windows VM connected to a home wireless network. We build three distinct classifiers using each of the three datasets using our procedure as above, but now augmenting the testing phase to also test against a stratified random sample of 70% of each of the other two datasets. The resulting matrix of average true and false positive rates (across all target obfuscators) is given in Table 8. The diagonal corresponds to training and testing on the same environment, whereas scores off the diagonal correspond to training and testing on different environments.

As can be seen, the ML classifiers do very well when trained and tested in the same environment. However, using the classifiers to attempt to classify a network flows generated by a distinct operating system and/or network significantly hinders performance. When using the same operating system, but different networks (the *TorA*/*TorB* and *TorB*/*TorA* entries) one sees less drastic reduction in performance. Changing operating systems however has large impact, with true positive rates being as low as 52% and false-positive rates reaching 12%. This provides some evidence that censors will indeed need to train classifiers on example traces representing all of the target types for which they need to be deployed.

False positives in campus traces. We finally turn to analyzing false-positive rates of a classifier trained with the methodology and data above when faced with real, not synthetic, network traces. We first combine the *TorA*, *TorB* and *TorC* datasets and use the combination to train and validate a classifier. We perform stratified randomized splits of 30% for training and 70% for validation, train a classifier for each obfuscator, and compute the PR-AUC on

TRAIN\TEST	*TorA*	*TorB*	*TorC*
TorA	0.99 (0.002)	0.88 (0.01)	0.52 (0.02)
TorB	0.93 (0.009)	0.99 (0.002)	0.78 (0.03)
TorC	0.57 (0.12)	0.64 (0.12)	0.99 (0.002)

Table 8: The effect of training and testing in the same or different environments. Reported is the average true-positive rate (average false-positive rate in the parentheses) across classifiers for all obfuscators using the dataset labeling the row for training and the dataset labeling the column for testing.

PT\Dataset	*OfficeDataset*	*CloudDataset*	*WifiDataset*
obfs3	5,281 (0.43%)	14,714 (0.20%)	34,726 (0.65%)
obfs4	730 (0.06%)	16,257 (0.22%)	24,221 (0.46%)
FTE	6,437 (0.53%)	23,432 (0.31%)	19,857 (0.37%)
meekA	2,065 (0.17%)	787 (0.01%)	1,024 (0.02%)
meekG	837 (0.07%)	3 (0%)	2 (0%)
Total	0.98%	0.70%	1.40%

Table 9: False positives of classifiers on the campus network datasets. The value in the parentheses is the false-positive rate of the selected classifier on a given campus network dataset. Recall that the number of flows tested for *OfficeDataset*, *CloudDataset*, and *WifiDataset* are 1.22 M, 7.48 M, and 5.32 M respectively.

the validation set. We repeat this procedure 20 times with fresh random splits, and select the classifiers with the highest PR-AUC. The PR-AUC of all the classifiers are greater than 0.90 (ranging from 0.91 - 0.99). The true-positive rates (false-positive rates) of the best/selected classifiers are 97.2% (0.13%), 97.0% (0.19%), 97.3% (0.08%), 98.3% (0.08%), and 97.8% (0.06%) for obfsproxy3, obfsproxy4, meek-amazon, meek-google, and FTE respectively.

Next, for each TCP flow in the campus datasets, we extract the features using the first 30 packets, label it as negative, and then test it with each obfuscator classifier. A summary of the results are in Table 9. The total false-positive rate of the classifiers across all three datasets is 0.98%. As broken down by dataset, we see as low as 0.70% (*CloudDataset*) and as high as 1.40% (*WifiDataset*).[6]

The classifier for obfsproxy3 produces the most false positives, followed by the classifier for FTE. The classifier for meek-google produces a relatively small number of false positives, which is a total of 842 false positives out of 14 M flows. That the wireless network exhibits the largest number of false positives may be due to their noisier nature [33,34]. For instance, when there are multiple TCP retransmissions in a flow, most of the packets we examined in the first 30 packets could be identical, hindering classification.

The false positives are associate with 12,551 distinct hosts, a small fraction of all monitored hosts, in the monitored networks. Meanwhile, they are only associated with 6,239 distinct destination IPs outside the campus networks. Less than 30% of these hosts or destination IPs are associated with more than 90% of the false positives. We find that a single IP outside the campus networks can contribute to as high as 4.6% of the false positives (as high as 1.2% for a single source IP inside the networks). This suggests that specific server-side or client-side settings could be the reasons for false positives.

We also determined the protocols of the false-positive flows. As shown in Table 10, most of the false positives are HTTP, SSL/TLS or *unknown* flows, according to Bro. The false positives

[6]Note that the total rates are not equal to the sum of the individual obfuscator false-positive rates, as some traces are falsely labeled by multiple classifiers.

Protocol\PT	obfs3	obfs4	FTE	meekA	meekG
HTTP	12,295 (0.12%)	10,383 (0.10%)	43,673 (0.42%)	1,414 (0.01%)	0
SSL/TLS	29,705 (1.91%)	22,768 (1.47%)	3,773 (0.24%)	1,684 (0.11%)	391 (0.03%)
SSH	0	0	33 (1.94%)	0	0
SMTP	114 (0.55%)	13 (0.06%)	73 (0.35%)	0	0
Unknown	2,739 (0.12%)	6,345 (0.27%)	2,148 (0.09%)	609 (0.03%)	451 (0.02%)
Total	44,853	39,509	49,700	3,707	842

Table 10: Breakdown of the numbers of flows from our campus traces incorrectly labeled by our ML classifiers as the indicated obfuscator. The values in the parentheses are the percentage of flows labeled by Bro as the indicated protocol that represent false positives (e.g., 0.12% of HTTP flows are mislabeled as obfsproxy3 by our classifier). "Unknown" means Bro fails to identify the protocol of the flow. The total number of false positives across all protocols is shown in the final row.

of the meek-amazon classifier have a diversity in their protocols. We examine some of the mislabeled flows, and realize they may use a patched version of protocols such as SSH with encrypted handshakes [23].

We use nDPI to examine the flows labeled as unknown by Bro. nDPI fails to identify 13.5% of the unknown flows, and reports 45 protocols found in the remainder. Of these, 21 are built atop HTTP or SSL/TLS. These protocols account for 78.2% of the unknown flows. Whether an ML approach can be enhanced to reduce false positives even further remains an open question.

6.2 Discussion

Our results show that trained classifiers using traffic-analysis-type techniques are effective at detecting obfuscators. This holds true even for the meek family of obfuscators, which are widely considered the hardest to detect by DPI. The true-positive rates are high, and the false-positive rates are relatively small. We discuss the impact of the latter in more detail in the next section.

Though the training process of the ML-based approaches are complex, the decision trees emitted by training are, themselves, actually quite simple: evaluating the trees requires between 6 and 13 integer comparisons. These comparisons use per-flow state including just a small number of integer counters, for up to the first 30 packets. We therefore believe the trees themselves will be relatively easy to deploy, while the trickier issue will be the lack of portability we observed. This specifically implies that building good decision trees requires careful training in, ideally, the local network environment to which they will be deployed.

7. ESTIMATING THE IMPACT OF FALSE POSITIVES

As we have emphasized, a critical component of successful obfuscator detection is achieving low false-positive rates and our work is the first that assesses them. Here we discuss the impact of false positives and whether the rates achieved by our detection approaches are sufficient for a censor's purpose.

Let us fix a particular representative scenario from our data, other scenarios can be analyzed similarly. Say a nation-state censor deploys the combined ML-based classifiers for each of the five obfuscators in the office environment, and assume a true-positive rate of 99% (as measured with *TorA*) and the false-positive rate of about 1% (as measured using the *OfficeDataset* dataset). For the foreseeable future, the base rate of obfuscator traffic will continue to be a tiny fraction of the overall traffic. Suppose, pessimistically for our analyses, that one out of every one billion flows is actually obfuscated traffic. Because of the low base rate, only about 1 in 10,000,000 flows marked as obfuscated will, in fact, be obfuscated. (Not 1 in 100 of marked flows, as one falling victim to the base rate fallacy might assume.) This is likely to result degradation of Internet performance, as perceived by "legitimate" users.

Aggressive censors may be willing to cause such degradation. Of the blocked connections in this scenario, about 34.9% will be encrypted connections (TLS, SSH), and some censors have already demonstrated a willingness to block these. Iran, for example, has been known to block all TLS connections, or at least degrade their performance [4]. (Degrading bandwidth of such flows will tend to force Tor to fail, but not non-Tor TLS.) Other countries have mandated that ISPs prevent or limit use of encrypted protocols [14, 32]. Even so, the majority of flows erroneously labeled by our techniques as obfuscated are plain, unencrypted HTTP. It seems reasonable that blocking these will be less palatable for censors with sensitivity to collateral damage.

Some sophisticated censors have moved to a two-stage detection pipeline to improve accuracy. China's so-called Great Firewall first uses DPI to flag flows as potentially resulting from a censorship circumvention tool. Measurements suggest that they flag any TLS connection that has the same list of allowed ciphers as a Tor client as a possible Tor connection [50, 51]. The destination IP address of all such flows is then submitted to a distributed system that asynchronously performs an active probe of the remote IP address using a Tor client. If the remote IP completes a Tor handshake, then the IP is is added to a blacklist. See [50, 51] for a detailed discussion of this behavior.

When using our obfuscator detection approaches in such a two-stage pipeline, the false-positive rate of the deployed approach will dictate the load on the more expensive, but more accurate, second-stage. (Perfectly accurate, active attack on Tor that was just described.) We refer to this second-stage as the DPI's slow path, and now turn to analyzing the slow-path load that would result from using our tests using the collected traces for simulation.

Table 11 shows a summary of the load seen by the DPI in terms of active flows per second as broken down by the various traces. For our purposes, a flow is active from the time stamp of its first packet to the time stamp of its last packet. We also report statistics regarding the load on the slow-path. A flow is labeled as going to the slow path whenever the combined ML-based classifier labels the traffic as obfuscated (i.e., any of the individual classifiers gives a positive label to the flow). As is shown, the maximum number of new slow-path flows per second is modest –in the worst case 54 are active in any given second– and the average is less than one. This suggests that, with minimal investment in the slow-path infrastructure, a censor could easily keep up with the false-positive rate of our obfuscation detectors. Of course, nation-state censors deal networks even larger than those considered in our work. Hence, caution should be exercised when extrapolating results from our setting to others.

8. CONCLUSION

We set out to answer the question of whether censors can use DPI to detect use of in-use network protocol obfuscators. Unfortunately,

	OfficeDataset	*CloudDataset*	*WifiDataset*
Avg. DPI load	14.6	138.5	182.2
Max. DPI load	362	1,042	1,580
Avg. slow-path load	0.08	0.60	0.77
Max. slow-path load	29	24	54
Slow-path active time	4.7%	38.2%	39.1%

Table 11: Summary statistics for DPI load (number of flows per second) and the slow-path load—the number of flows per second flagged as any obfuscator by our best-performing ML classifiers.

our analyses suggest that the answer is 'yes'. We present the first comprehensive analysis of detectability of in-use network protocol obfuscators, as they are deployed in Tor. Our analyses reveal fast entropy-based tests for randomizer protocols and FTE (which is mostly randomized), and slightly less efficient, but still practical, machine learning-based attacks that reliably detect meek, a state-of-the-art tunneling obfuscator. We also show that some semantics-based detection tests suggested in the literature are less effective than a censor might like, due to the inherent long tail of non-standard network traffic. This suggests that future development of semantics-based tests should necessarily perform false positive analyses. Towards helping future researchers with such tasks, we will make our analysis platform open source for other researchers.

It is important to note that the detection techniques we explore can be, in turn, easily circumvented in almost all cases with simple updates to the obfuscator. This suggests that with the current state-of-the-knowledge on building practical obfuscators, anti-censorship tools will only have the advantage when censors remain ignorant of (or choose to ignore knowledge of) the details of their design. Building more robust, future-proof obfuscators that cannot be blocked by future, efficient DPI algorithms with knowledge of the obfuscator design remains an open question.

9. ACKNOWLEDGMENTS

The authors would like to especially thank the system administrators at the University of Wisconsin who helped ensure the experiments went smoothly. This work was supported in part by the National Science Foundation under grants CNS-1546033, CNS-1330308, CNS-1065134, CNS-0845610, and CNS-1319061, and by a generous gift from Dr. Eric Schmidt (New Digital Age grant).

10. REFERENCES

[1] M. AlSabah, K. Bauer, and I. Goldberg. Enhancing tor's performance using real-time traffic classification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 73–84. ACM, 2012.

[2] Apache. Hive operators and user-defined functions. `https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF`, 2015.

[3] J. Appelbaum and N. Mathewson. Pluggable transports for circumvention. `https://www.torproject.org/docs/pluggable-transports.html.en`, 2012.

[4] S. Aryan, H. Aryan, and J. A. Halderman. Internet censorship in iran: A first look. *Free and Open Communications on the Internet, Washington, DC, USA*, 2013.

[5] J. Barker, P. Hannay, and P. Szewczyk. Using traffic analysis to identify the second generation onion router. In *Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on*, pages 72–78. IEEE, 2011.

[6] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian. Traffic classification on the fly. *ACM SIGCOMM Computer Communication Review*, 36(2):23–26, 2006.

[7] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, pages 605–616. ACM, 2012.

[8] A. Communeau, P. Quillent, and A. Compain. Detecting FTE proxy. 2014.

[9] D. Cournapeau. Scikit-learn: Machine learning in Python. `http://scikit-learn.org/`, 2007.

[10] J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

[11] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

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

[13] K. P. Dyer, S. E. Coull, and T. Shrimpton. Marionette: A programmable network-traffic obfuscation system. In *Proceedings of USENIX Security 2015*, August 2015.

[14] Eugene. Age of surveillance: the fish is rotting from its head. `http://non-linear-response.blogspot.com/2011/01/age-of-surveillance-fish-is-rotting.html`, 2011.

[15] Hill01. "Weird" utf-8 characters in POST body causing content-length mismatch. `https://github.com/strongloop/express/issues/1816`, 2013.

[16] A. Houmansadr, C. Brubaker, and V. Shmatikov. The parrot is dead: Observing unobservable network communications. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 65–79. IEEE, 2013.

[17] ISO. 171/sc 2: Iso 32000–1: 2008 document management-portable document format-part 1: Pdf 1.7.

[18] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python. `http://www.scipy.org/`, 2001.

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

[20] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: multilevel traffic classification in the dark. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 229–240. ACM, 2005.

[21] K. H. Kremer. The trouble with the XREF table. `http://khkonsulting.com/2013/01/the-trouble-with-the-xref-table/`, 2013.

[22] B. Krishnamurthy, J. C. Mogul, and D. M. Kristol. Key differences between HTTP/1.0 and HTTP/1.1. *Computer Networks*, 31(11):1737–1751, 1999.

[23] B. Leidl. Obfuscated-openssh. `https://github.com/brl/obfuscated-openssh`, 2009.

[24] D. Luchaup, K. P. Dyer, S. Jha, T. Ristenpart, and T. Shrimpton. LibFTE: A toolkit for constructing practical, format-abiding encryption schemes. In *Proceedings of USENIX Security 2014*, August 2014.

[25] Microsoft. Invalid content-length header may cause requests to fail through ISA server. https://support.microsoft.com/en-us/kb/300707, 2007.

[26] H. Mohajeri Moghaddam, B. Li, M. Derakhshani, and I. Goldberg. Skypemorph: Protocol obfuscation for tor bridges. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 97–108. ACM, 2012.

[27] T. T. Nguyen and G. Armitage. Training on multiple sub-flows to optimise the use of machine learning classifiers in real-world ip networks. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 369–376. IEEE, 2006.

[28] Ntop.org. nDPI. http://www.ntop.org/products/deep-packet-inspection/ndpi/, 2015.

[29] G. Nychis, V. Sekar, D. G. Andersen, H. Kim, and H. Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 151–156. ACM, 2008.

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

[31] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[32] Phobos. Kazakhstan upgrades censorship to deep packet inspection. https://blog.torproject.org/blog/kazakhstan-upgrades-censorship-deep-packet-inspection, 2012.

[33] S. Rayanchu, A. Mishra, D. Agrawal, S. Saha, and S. Banerjee. Diagnosing wireless packet losses in 802.11: Separating collision from weak signal. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, 2008.

[34] C. Reis, R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Measurement-based models of delivery and interference in static wireless networks. volume 36, pages 51–62. ACM, 2006.

[35] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, DTIC Document, 2001.

[36] C. Sanders, J. Valletta, B. Yuan, and D. Johnson. Employing entropy in the detection and monitoring of network covert channels. 2012.

[37] Seleniumhq.org. Selenium - web browser automation. http://www.seleniumhq.org/, 2015.

[38] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[39] Tor project. Obfsproxy2. https://gitweb.torproject.org/pluggable-transports/obfsproxy.git/tree/doc/obfs2/obfs2-protocol-spec.txt, 2015.

[40] Tor project. Obfsproxy3. https://gitweb.torproject.org/pluggable-transports/obfsproxy.git/tree/doc/obfs3/obfs3-protocol-spec.txt, 2015.

[41] Tor project. Stem. https://stem.torproject.org/, 2015.

[42] Tor project. Tor meek. https://trac.torproject.org/projects/tor/wiki/doc/meek, 2015.

[43] Tor project. Tor metrics. https://metrics.torproject.org/, 2015.

[44] University of Washington. uProxy. https://www.uproxy.org/, 2015.

[45] Q. Wang, X. Gong, G. T. Nguyen, A. Houmansadr, and N. Borisov. Censorspoofer: asymmetric communication using ip spoofing for censorship-resistant web browsing. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 121–132. ACM, 2012.

[46] Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, S. Cheung, F. Wang, and D. Boneh. Stegotorus: a camouflage proxy for the tor anonymity system. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 109–120. ACM, 2012.

[47] A. M. White, S. Krishnan, M. Bailey, F. Monrose, and P. A. Porras. Clear and present data: Opaque traffic and its security implications for the future. In *NDSS*, 2013.

[48] B. Wiley. Dust: A blocking-resistant internet transport protocol. *Technical rep ort. http://blanu. net/Dust. pdf*, 2011.

[49] N. Williams, S. Zander, and G. Armitage. A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification. *ACM SIGCOMM Computer Communication Review*, 36(5):5–16, 2006.

[50] P. Winter and J. R. Crandall. The great firewall of China: How it blocks tor and why it is hard to pinpoint. 2012.

[51] P. Winter and S. Lindskog. How the great firewall of China is blocking tor. *Free and Open Communications on the Internet*, 2012.

[52] P. Winter, T. Pulls, and J. 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.

[53] X. Xu, Z. M. Mao, and J. A. Halderman. Internet censorship in China: Where does the filtering occur? In *Passive and Active Measurement*, pages 133–142. Springer, 2011.

[54] Yawning. Obfsproxy4. https://github.com/Yawning/obfs4/blob/master/doc/obfs4-spec.txt, 2015.

[55] J. Yuan, Z. Li, and R. Yuan. Information entropy based clustering method for unsupervised internet traffic classification. In *Communications, 2008. ICC'08. IEEE International Conference on*, pages 1588–1592. IEEE, 2008.