David Fifield*, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson

Blocking-resistant communication through domain fronting

Abstract: We describe "domain fronting," a versatile censorship circumvention technique that hides the remote endpoint of a communication. Domain fronting works at the application layer, using HTTPS, to communicate with a forbidden host while appearing to communicate with some other host, permitted by the censor. The key idea is the use of different domain names at different layers of communication. One domain appears on the "outside" of an HTTPS request—in the DNS request and TLS Server Name Indication—while another domain appears on the "inside"—in the HTTP Host header, invisible to the censor under HTTPS encryption. A censor, unable to distinguish fronted and non-fronted traffic to a domain, must choose between allowing circumvention traffic and blocking the domain entirely, which results in expensive collateral damage. Domain fronting is easy to deploy and use and does not require special cooperation by network intermediaries. We identify a number of hard-to-block web services, such as content delivery networks, that support domain-fronted connections and are useful for censorship circumvention. Domain fronting, in various forms, is now a circumvention workhorse. We describe several months of deployment experience in the Tor, Lantern, and Psiphon circumvention systems, whose domain-fronting transports now connect thousands of users daily and transfer many terabytes per month.

Keywords: censorship circumvention

DOI 10.1515/popets-2015-0009
Received 2015-02-15; revised 2015-05-15; accepted 2015-05-15.

***Corresponding Author: David Fifield:** University of California, Berkeley, E-mail: fifield@eecs.berkeley.edu
Chang Lan: University of California, Berkeley, E-mail: clan@eecs.berkeley.edu
Rod Hynes: Psiphon Inc, E-mail: r.hynes@psiphon.ca
Percy Wegmann: Brave New Software, E-mail: ox.to.a.cart@gmail.com
Vern Paxson: University of California, Berkeley and the International Computer Science Institute, E-mail: vern@berkeley.edu

1 Introduction

Censorship is a daily reality for many Internet users. Workplaces, schools, and governments use technical and social means to prevent access to information by the network users under their control. In response, those users employ technical and social means to gain access to the forbidden information. We have seen an ongoing conflict between censor and censored, with advances on both sides, more subtle evasion countered by more powerful detection.

Circumventors, at a natural disadvantage because the censor controls the network, have a point working in their favor: the censor's distaste for "collateral damage," incidental overblocking committed in the course of censorship. Collateral damage is harmful to the censor, because the overblocked content has economic or social value, so the censor tries to avoid it. (Any censor not willing to turn off the Internet completely must derive *some* benefit from allowing access, which overblocking harms.) One way to win against censorship is to entangle circumvention traffic with other traffic whose value exceeds the censor's tolerance for overblocking.

In this paper we describe "domain fronting," a general-purpose circumvention technique based on HTTPS that hides the true destination of a communication from a censor. Fronting works with many web services that host multiple domain names behind a frontend server. These include such important infrastructure as content delivery networks (CDNs) and Google's panoply of services—a nontrivial fraction of the web. (Section 4 is a survey of suitable services.) The utility of domain fronting is not limited to HTTPS communication, nor to accessing only the domains of a specific web service. It works well as a domain-hiding component of a larger circumvention system, an HTTPS tunnel to a general-purpose proxy.

The key idea of domain fronting is the use of different domain names at different layers of communication. In an HTTPS request, the destination domain name appears in three relevant places: in the DNS query, in the TLS Server Name Indication (SNI) extension [18, §3], and in the HTTP Host header [20, §14.23]. Ordinarily, the same domain name appears in all three places.

In a domain-fronted request, however, the DNS query and SNI carry one name (the "front domain"), while the HTTP Host header, hidden from the censor by HTTPS encryption, carries another (the covert, forbidden destination).

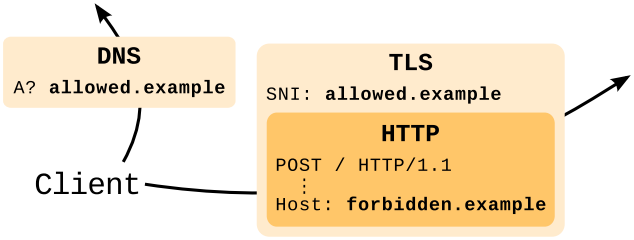


Fig. 1. Domain fronting uses different domain names at different layers. At the plaintext layers visible to the censor—the DNS request and the TLS Server Name Indication—appears the front domain **allowed.example**. At the HTTP layer, unreadable to the censor, is the actual, covert destination **forbidden.example**.

The censor cannot block on the contents of the DNS request nor the SNI without collaterally blocking the front domain. The Host header is invisible to the censor, but visible to the frontend server receiving the HTTPS request. The frontend server uses the Host header internally to route the request to its covert destination; no traffic ever reaches the putative front domain. Domain fronting has many similarities with decoy routing [29, 35, 69, 70]; it may be understood as "decoy routing at the application layer." A fuller comparison with decoy routing appears in Section 3.

This Wget command demonstrates domain fronting on Google, one of many fronting-capable services. Here, the HTTPS request has a Host header for maps.google.com, even though the DNS query and the SNI in the TLS handshake specify www.google.com. The response comes from maps.google.com.

```
$ wget -q -O - https://www.google.com/ \
    --header 'Host: maps.google.com' | \
    grep -o '<title>.*</title>'
<title>Google Maps</title>
```

A variation is "domainless" fronting, in which there is no DNS request and no SNI. It appears to the censor that the user is browsing an HTTPS site by its IP address, or using a web client that does not support SNI. Domainless fronting can be useful when there is no known front domain with sufficiently high collateral damage; it leaves the censor the choice of blocking an entire IP address (or blocking SNI-less connections entirely), rather than blocking only a single domain. According to our communication with the International Computer Science Institute's certificate notary [32], which observes on the order of 50 million TLS connections daily, 16.5% of TLS connections in June 2014 lacked SNI, which is enough to make it difficult for a censor to block SNI-less TLS outright.

Domain fronting works with CDNs because a CDN's frontend server (called an "edge server"), on receiving a request for a resource not already cached, forwards the request to the domain found in the Host header (the "origin server"). (There are other ways CDNs may work, but this "origin pull" configuration is common.) The client issues a request that appears to be destined for an unrelated front domain, which may be any of the CDN's domains that resolve to an edge server; this fronted request is what the censor sees. The edge server decrypts the request, reads the Host header and forwards the request to the specified origin, which in the circumvention scenario is a general-purpose proxy. The origin server, being a proxy, would be blocked by the censor if accessed directly—fronting hides its address from the censor.

On services that do not automatically forward requests, it is usually possible to install a trivial "reflector" web application that emulates an origin-pull CDN. In this case, fronting does not protect the address of the origin per se; rather it protects the address of the reflector application, which in turn forwards requests to the origin. Google App Engine is an example of such a service: against a censor that blocks the App Engine domain appspot.com but allows other Google domains, domain fronting enables access to a reflector running on appspot.com.

No matter the specifics of particular web services, as a general rule they do not forward requests to arbitrary domains—only to domains belonging to one of their customers. In order to deploy a domain-fronting proxy, one must become a customer of the CDN (or Google, etc.) and pay for bandwidth. It is the owner of the covert domain who pays the bandwidth bills, not the owner of the front domain, which need not have any relation to the covert domain beyond using the same web service.

The remainder of this paper is devoted to a deep exploration of domain fronting as we have deployed it in practice. Section 2 explains our threat model and assumptions. Section 3 gives general background on the circumvention problem and outlines its three grand challenges: address-based blocking, content-based blocking, and active probing. Domain-fronting systems are capable of meeting all three challenges, forcing censors to use more expensive, less reliable censorship tech-

niques that have heretofore not been seen in practice. Section 4 is a survey of CDNs and other services that are usable for fronting; we identify general principles as well as idiosyncrasies that affect implementation. The following sections are three case studies of deployment: Section 5 for Tor, Section 6 for Lantern, and Section 7 for Psiphon. Section 8 sketches domain fronting's resistance to statistical traffic analysis attacks. Section 9 has general discussion and Section 10 summarizes.

2 Threat model

Our threat model includes four actors: the censor, the censored client, the intermediate web service, and the covert destination (a proxy server). Circumvention is achieved when the client reaches the proxy, because the proxy grants access to any other destination. The client and proxy cooperate with each other. The intermediate web service need not cooperate with either, except to the extent that it does not collude with the censor.

The censor controls a (generally national) network and the links into and within it. The censor can inspect traffic flowing across all links under its control and can block or allow any packet. The censor can inject and replay traffic, and operate its own clients and servers. The client lies within the censor's network, while the intermediate web service and proxy lie outside. The censor blocks direct communication between the client and the proxy, but allows HTTPS between the client and at least one front domain or IP address on the intermediate web service.

The client, intermediate web service, and destination proxy are uncontrolled by the censor. The censor does not control a trusted certificate authority: it cannot man-in-the-middle TLS without being caught by ordinary certificate validation. The client is able to obtain the necessary circumvention software.

3 Background and related work

Broadly speaking, there are three main challenges in proxy-based circumvention: blocking by content, blocking by address, and active probing. Blocking by content is based on *what you say*, blocking by address is based on *whom you talk to*, and active probing means *the censor acts as a client*. A savvy censor will employ all these techniques, and effective circumvention requires countering them all.

A content-blocking censor inspects packets and payloads, looking, for example, for forbidden protocols or keywords. Content-based blocking is sometimes called deep packet inspection (DPI). An address-blocking censor forbids all communication with certain addresses, for example IP addresses and domain names, regardless of the contents of the communication. An active-probing censor does not limit itself to observation and manipulation of user-induced traffic only. It sends its own proxy requests (active probes), either proactively or on demand in response to observed traffic, and blacklists any proxies it finds thereby. Active probing is a precise means of identifying proxy servers, even when a protocol is hard to detect on the wire—it can regarded as a way of reducing accidental overblocking. Winter and Lindskog [66] confirmed an earlier discovery of Wilde [64] that China's Great Firewall discovers secret Tor bridges by issuing followup scans after observing a suspected Tor connection.

There are two general strategies for countering content-based blocking. The first is to look unlike anything the censor blocks; the second is to look like something the censor allows. Following the first strategy are the so-called "look-like-nothing" transports whose payloads look like a uniformly random byte stream. Examples of look-like-nothing transports are obfuscated-openssh [41] and its string of successors: obfs2 [33], obfs3 [34], ScrambleSuit [67], and obfs4 [4]. They all work by re-encrypting an underlying stream so that there remain no plaintext components, not even in the handshake and key exchange. obfs2, introduced in early 2012 [14], used a fairly weak key exchange that can be detected passively; it is now deprecated and little used. obfs3 is Tor's most-used transport [61] as of May 2015. It improves on obfs2 with a Diffie–Hellman key exchange, public keys being encoded so as to be indistinguishable from random binary strings. ScrambleSuit and obfs4 add resistance to active probing: the server accepts a TCP connection but does not send a reply until the client proves knowledge of a secret shared out of band. ScrambleSuit and obfs4 additionally obscure the traffic signature of the underlying stream by modifying packet lengths and timing.

The other strategy against DPI is the steganographic one: look like something the censor allows. fteproxy [17] uses format-transforming encryption to encode data into strings that match a given regular expression, for example a regular-expression approximation of HTTP. StegoTorus [63] transforms traffic to look like a cover protocol using a variety of special-purpose encoders. Code Talker Tunnel (formerly SkypeMorph) [47]

mimics a Skype video call. FreeWave [30] encodes a stream as an acoustic signal and sends it over VoIP to a proxy. Dust [65] uses encryption to hide static byte patterns and then shapes statistical features such as packet lengths and byte frequencies to match specified distributions.

Houmansadr et al. [28] evaluate "parrot" systems that imitate another protocol and conclude that unobservability by imitation is fundamentally flawed. To fully mimic a complex and sometimes proprietary protocol like Skype is difficult, because the system must imitate not only the normal operation of the protocol, but also its reaction to errors, its typical traffic patterns, and quirks of implementations. Geddes et al. [23] demonstrate that even non-parrot systems may be vulnerable to attacks that disrupt circumvention while having little effect on ordinary traffic. Their examination includes VoIP protocols, in which packet loss and duplication are acceptable. The censor may, for example, strategically drop certain packets in order to disrupt a covert channel, without much harming ordinary calls.

The challenge of address-based blocking is a difficult one that has inspired various creative circumvention ideas. Tor has long faced the problem of the blocking of its relays, the addresses of which appear in a public directory. In response, Tor began to reserve a portion of its relays as secret "bridges" [15] whose addresses are not publicly known. BridgeDB [9], the database of secret bridges, carefully distributes addresses so that it is easy to learn a few bridges, but hard to enumerate all of them. BridgeDB uses CAPTCHAs and other rate-limiting measures, and over short time periods, always returns the same bridges to the same requester, preventing enumeration by simple repeated queries. BridgeDB is also capable of distributing the addresses of obfuscated bridges (currently obfs3, obfs4, ScrambleSuit, and fteproxy), granting IP-blocking resistance to DPI-resistant systems that otherwise lack it.

CensorSpoofer [62] decouples upstream and downstream data channels. The client sends data to a CensorSpoofer proxy over a low-bandwidth covert channel such as email. The proxy sends data back over a UDP channel, all the time spoofing its source address so the packets appear to originate from some other "dummy" host. The censor has no IP address to block, because the proxy's true address never appears on the wire. Client and server have the challenge of agreeing on a dependable covert upstream channel that must remain unblocked, and the client must carry on a believable UDP conversation with the dummy host—a VoIP call, for example.

Flash proxy [22] resists address blocking by conscripting web users as temporary proxies. Each JavaScript-based proxy lasts only as long as a user stays on a web page, so the pool of proxies is constantly changing. If one of them is blocked, there is soon another to replace it. Flash proxy's approach to address blocking is the opposite of domain fronting's: where flash proxy uses many cheap, disposable, individually blockable proxies, domain fronting uses just a few high-value front domains on hard-to-block network infrastructure. A drawback with flash proxy's use of the browser is that the client must be able to receive a TCP connection; in particular it must not be behind network address translation (NAT), which limits flash proxy's usefulness. Part of the flash proxy protocol requires the client to send a small amount of unblockable data in a process called rendezvous. The default rendezvous mechanism has used domain fronting through Google App Engine since 2013 [58]. Flash proxy itself does nothing to defend against DPI. Connections between censored clients and browser-based proxies use WebSocket, a meta-protocol running on HTTP, but inside the WebSocket framing is the ordinary TLS-based Tor protocol.

Decoy routing [35] is a technique that puts proxies in the middle of network paths, rather than at the ends. For this reason, it is also called end-to-middle proxying. Realizations of decoy routing include Telex [70], Cirripede [29], and TapDance [69]. Decoy routing asks friendly ISPs to deploy special routers that reside on network paths between censored users and uncensored "decoy" Internet destinations. Circumvention traffic is "tagged" in a way that is detectable only by the special routers, and not by the censor. On receiving a tagged communication, the router shunts it away from its apparent, *overt destination* and toward a censored, *covert destination*. Domain fronting is similar in spirit to decoy routing: think of domain fronting as decoy routing at the application layer. In place of a router, domain fronting has a frontend server; in place of the overt destination is the front domain. Both systems tag flows in a way that is invisible to the censor: decoy routing uses, for example, a hash embedded in a client nonce, while fronting uses the HTTP Host header, encrypted inside of HTTPS. Fronting has the advantage of not requiring cooperation by network intermediaries.

Schuhard et al. [54] introduce the idea of a *routing adversary* against decoy routing, and show that the connectivity of the Internet enables censors to force network users onto paths that do not include participating routers. Simulations by Houmansadr et al. [31] show that even though such alternate paths exist, they are

many times more costly to the censor, especially when participating routers are placed strategically.

Collage [11] makes a covert channel out of web sites that accept user-generated content, like photos. Both sender and receiver rendezvous through one of these sites in order to exchange messages. The design of Collage recognizes the need for the proxy sites to be resistant to blocking, which it achieves through the wide availability of suitable sites.

CloudTransport [10] uses cloud storage services, for example Amazon S3, as a communication channel by encoding sends and receives as reads and writes to shared remote files. CloudTransport has much in common with domain fronting: it hides the true endpoint of a communication using HTTPS, and it sends traffic through a domain with high collateral damage. In CloudTransport, the hidden destination, which is a storage bucket name rather than a domain, is hidden in the path component of a URL. For example, in the S3 URL https://s3.amazonaws.com/*bucketname*/*filename*, the censor only gets to "see" the generic domain part, "s3.amazonaws.com". The path component "/*bucketname*/*filename*", which would reveal the use of CloudTransport, cannot be used for blocking because it is encrypted under HTTPS.

In a prescient 2012 blog post [8], Bryce Boe described how to gain access to Google HTTPS services through a single whitelisted Google IP address, by manual editing of a hosts file. He observed that Google App Engine could serve as a general-purpose proxy, and anticipated the countermeasure of SNI filtering, noting that sending a false SNI could defeat it.

To our knowledge, the earliest use of domain fronting for circumvention was by GoAgent [24], a tool based on Google App Engine and once widely used in China. Users of GoAgent upload a personal copy of the proxy code to App Engine, where it runs on a subdomain of appspot.com. In order to reach appspot.com, GoAgent fronts through a Google IP address, using the "domainless" model without SNI. GoAgent does not use an additional general-purpose proxy after fronting; rather, it fetches URLs directly from the App Engine servers. Because of that, GoAgent does not support protocols other than HTTP and HTTPS, and the end-to-end security of HTTPS is lost, as web pages exist in plaintext on the App Engine servers before being re-encrypted back to the client. According to a May 2013 survey [53], GoAgent was the most-used circumvention tool in China, with 35% of survey respondents having used it in the previous month. It ranked higher than paid (29%) and free VPNs (18%), and far above special-purpose tools like Tor (2.9%) and Psiphon (2.5%). GoAgent was disrupted in China starting in the beginning of June 2014, when all Google services were blocked [2, 25]. The block also affected our prototype systems when used in China with App Engine, though they continued to work in China over other web services.

4 Fronting-capable web services

In this section we survey a variety of web services and evaluate their suitability for domain fronting. Most of the services we evaluated support fronting in one form or another, but they each have their own quirks and performance characteristics. The survey is not exhaustive but it includes many of the most prominent content delivery networks. Table 1 is a summary.

Pricing across services varies widely, and depends on complicated factors such as geographical region, bandwidth tiers, price breaks, and free thresholds. Some services charge per gigabyte or per request, some for time, and some for other resources. Most services charge between $0.10 and $0.20 per GB; usually bandwidth is cheaper in North America and Europe than in the rest of the world.

Recall that even services that support domain fronting will front only for the domains of their own customers. Deployment on a new service typically requires becoming a customer, and an outlay of time and money. Of the services surveyed, we have at some time deployed on Google App Engine, Amazon CloudFront, Microsoft Azure, Fastly, and CloudFlare. The others we have only tested using manually crafted HTTP requests.

Google App Engine [26] is a web application platform. Users can upload a web app needing nothing more than a Google account. Each application gets a user-specified subdomain of appspot.com, for which almost any Google domain can serve as a front, including google.com, gmail.com, googleapis.com, and many others. App Engine can run only web applications serving short-lived requests, not a general-purpose proxy such as a Tor bridge. For that reason we use a tiny "reflector" application that merely forwards incoming requests to a long-lived proxy running elsewhere. Fronting through App Engine is attractive in the case where the censor blocks appspot.com but at least one other Google domain is reachable. App Engine costs $0.12/GB and $0.05 for each "instance hour" (the number of running instances of the app is adjusted dynamically to meet load, and you pay for each instance after the first).

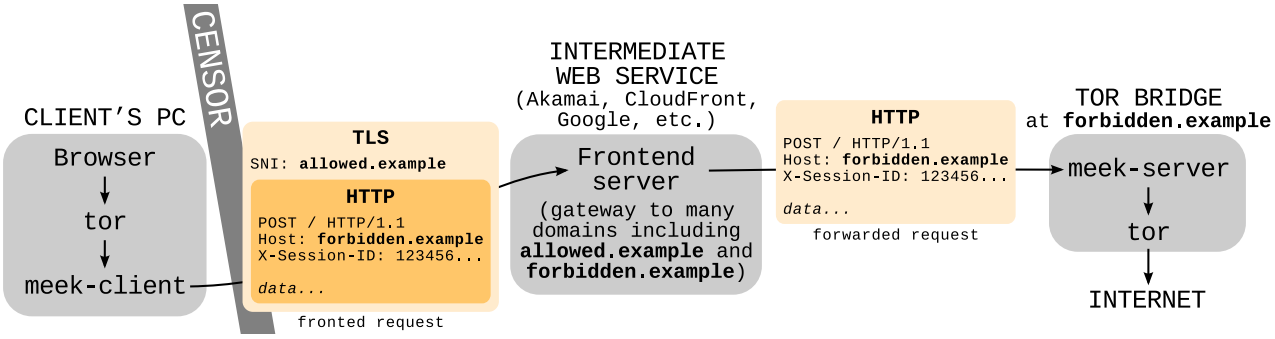


Fig. 2. Architecture of meek. The client sends an HTTP request to the Tor bridge by way of an intermediate web service such as a CDN. The client protects the bridge's domain name **forbidden.example** from the censor by fronting it with another name, here **allowed.example**. The intermediate web server decrypts the TLS layer and forwards the request to the bridge according to the Host header. The bridge sends data back to the client in the HTTP response. meek-client and meek-server are the interface between Tor and the pluggable transport; from Tor's point of view, everything between meek-client and meek-server is an opaque data transport. The host at allowed.example does not participate in the communication.

Table 1. Summary of fronting-capable services. The table does not include other kinds of services, such as shared web hosting, that may work for domain fronting. Bandwidth charges usually vary by geographic region. Many services offer price breaks starting around 10 TB/month. Prices are current as of May 2015 and are rounded to the nearest cent.

service	$/GB	$/10K reqs.	$/hour	$/month
App Engine[1]	0.12	–	0.05	–
CloudFront	0.09–0.25	0.01–0.02	–	–
Azure	0.09–0.14	–	–	–
Fastly[2]	0.12–0.19	0.01	–	–
CloudFlare[3]	–	–	–	200
Akamai[4]	–	–	–	400
Level 3[5]	0.10–0.25	–	–	–

[1] App Engine dynamically scales the number of "instances" of the application code in order to handle changing load. Every instance after the first costs $0.05/hour.

[2] Fastly has a minimum monthly charge of $50.

[3] CloudFlare charges $200/month for its "business" plan. It has other plans that cost more and less.

[4] Akamai does not publish pricing information; the prices here are from a reseller called Cache Simple, which quotes $400/month for 1000 GB transfer, and $0.50/GB for overages.

[5] Level 3 does support domain fronting per se, but paths under the secure.footprint.net domain can likely serve the same purpose. Level 3 does not publish pricing information; the prices here are from a reseller called VPS.NET, which quotes $0.10–0.25/GB.

Applications are free of charge if they stay below certain usage thresholds, for example 1 GB of bandwidth daily, making possible a distributed, upload-your-own-app model in the style of GoAgent.

Amazon CloudFront [3] is the CDN of Amazon Web Services. A CloudFront "distribution," as a CDN configuration is called, associates an automatically generated subdomain of cloudfront.net with an origin server. The front domain may be any other cloudfront.net subdomain (all of which support HTTPS through a wildcard certificate), or any other DNS alias for them. CloudFront is easy to set up: one must only set the origin domain and no reflector app is needed. Pricing per GB ranges from $0.085 for the United States and Europe, up to $0.25 for South America, with price breaks starting at 10 TB/month. There is an additional charge per 10,000 HTTPS requests, ranging from $0.0075 in the United States to $0.0160 in South America. CloudFront has a usage tier that is free of charge for a year, subject to a bandwidth limit of 50 GB/month.

Microsoft Azure [46] is a cloud computing platform that features a CDN. Like CloudFront, Azure assigns automatically generated subdomains of vo.msecnd.net. any of which can front for any other. There are other possible front domain names, like ajax.aspnetcdn.com, that are used as infrastructure by many web sites, lending them high collateral damage. Unlike CloudFront's, Azure's CDN forwards only to Azure-affiliated domains, so as with App Engine, it is necessary to run a reflector app that forwards requests to some external proxy. Bandwidth costs $0.087–0.138/GB, with price breaks starting at 10 TB/month.

Fastly [19] is a CDN. Unlike most CDNs, Fastly validates the SNI: if SNI and Host do not match, the

edge server returns an HTTP 400 ("Bad Request") error. However, if the TLS ClientHello simply omits SNI, then the Host may be any Fastly domain. Fastly therefore requires the "domainless" fronting style. Fastly's pricing model is similar to CloudFront's. They charge between $0.12 and $0.19 per GB and $0.0075 and $0.009 per 10,000 requests, depending on the region.

CloudFlare [12] is a CDN also marketed as protection against denial-of-service attacks. Like Fastly, CloudFlare checks that the SNI matches the Host header and therefore requires sending requests without SNI. CloudFlare charges a flat fee per month and does not meter bandwidth. There is a no-cost plan intended for small web sites, which is adequate for a personal domain-fronting installation. The upgraded "business" plan is $200/month.

Akamai [1] is a large CDN. Requests may be fronted through the special HTTPS domain a248.e.akamai.net, or other customer-configured DNS aliases, though it appears that certain special domains get special treatment and do not work as fronts. Akamai has the potential to provide a lot of cover: in 2010 it carried 15–20% of all web traffic [49]. Akamai does not publish pricing details, but it is reputed to be among the pricier CDNs. We found a reseller, Cache Simple, that charges $400 for 1000 GB/month, and $0.50/GB after that. The special domain a248.e.akamai.net began to be DNS-poisoned in China in late September 2014 [27] (possibly because it had been used to mirror blocked web sites), necessitating an alternative front domain in that country.

Level 3 [42] is a tier-1 network operator that has a CDN. Unlike other services in this section, Level 3 does not appear to support domain fronting. However, we mention it because it may be possible to build similar functionality using distinct URL paths under the domain secure.footprint.net (essentially using the path, rather than the Host header, as a hidden tag). Level 3 does not publish pricing data. We found a reseller, VPS.NET, that quotes $34.95 for the first 1000 GB and $0.10–0.25/GB thereafter. Level 3's special HTTPS domain secure.footprint.net is also now DNS-poisoned in China.

There are other potential deployment models apart from CDNs. For example, there are cheap web hosts that support both PHP and HTTPS (usually with a shared certificate). These features are enough to support a reflector app written in PHP, which users can upload under their own initiative. In this do-it-yourself model, blocking resistance comes not from a strong front domain, but from the diffuseness of many proxies, each carrying only a small amount of traffic. The URLs of these proxies could be kept secret, or could be carefully disseminated by a proxy-distribution service like BridgeDB [9]. Psiphon uses this approach when in "unfronted" mode.

Another alternative is deployment with the cooperation of an existing important web site, the blocking of which would result in high collateral damage. It is a nice feature of domain fronting that it does not require cooperation by the intermediate web service, but if you have cooperation, you can achieve greater efficiency. The important web site could, for example, reserve a magic URL path or domain name, and forward matching requests to a proxy running locally. The web site does two jobs: its ordinary high-value operations that make it expensive to block, and a side job of handling circumvention traffic. The censor cannot tell which is which because the difference is confused by HTTPS.

5 Deployment on Tor

We implemented domain fronting as a Tor pluggable transport [5] called meek. meek combines domain fronting with a simple HTTP-based tunneling proxy. Domain fronting enables access to the proxy; the proxy transforms a sequence of HTTP requests into a Tor data stream.

The components of the system appear in Figure 2. meek-client acts as an upstream proxy for the client's Tor process. It is essentially a web client that knows how to front HTTPS requests. When meek-client receives an outgoing chunk of data from a client Tor process, it bundles the data into a POST request and fronts the request through the web service to a Tor bridge. The Tor bridge runs a server process, meek-server, that decodes incoming HTTP requests and feeds their data payload into the Tor network.

The server-to-client stream is returned in the bodies of HTTP responses. After receiving a client request, meek-server checks for any pending data the bridge has to send back to the client, and sends it back in the HTTP response. When meek-client receives the response, it writes the body back into the client Tor.

The body of each HTTP request and response carries a small chunk of an underlying TCP stream (up to 64 KB). The chunks must be reassembled, in order, without gaps or duplicates, even in the face of transient failures of the intermediate web service. meek uses

```
POST / HTTP/1.1
Host: forbidden.example
X-Session-Id: cbIzfhx1HnR
Content-Length: 517

\x16\x03\x01\x02\x00\x01\x00\x01\xfc\x03\x03\x9b\xa9...

    HTTP/1.1 200 OK
    Content-Length: 739

    \x16\x03\x03\x00\x3e\x02\x00\x00\x3a\x03\x03\x53\x75...

POST / HTTP/1.1
Host: forbidden.example
X-Session-Id: cbIzfhx1HnR
Content-Length: 0


    HTTP/1.1 200 OK
    Content-Length: 75

    \x14\x03\x03\x00\x01\x01\x16\x03\x03\x00\x40\x06\x84...
```

Fig. 3. Requests and responses in the meek HTTP protocol. The session ID is randomly generated by the client. Request/response bodies contain successive chunks of a Tor TLS stream (`\x16\x03\x01` is the beginning of a TLSv1.0 ClientHello message). The second POST is an empty polling request. The messages shown here are encrypted inside HTTPS until after they have been fronted, so the censor cannot use the Host and X-Session-Id headers for classification.

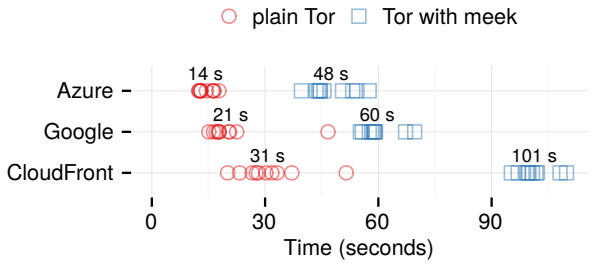


Fig. 4. Time to download an 11 MB file through Tor, with and without meek. Text labels indicate the mean of 10 measurements. Bulk-download times increase by about a factor of 3 when meek is activated. The middle and exit nodes are constant across all measurements; only the entry varies according to the service. The without-meek circuits use an entry node located at the same IP address as the corresponding with-meek circuits.

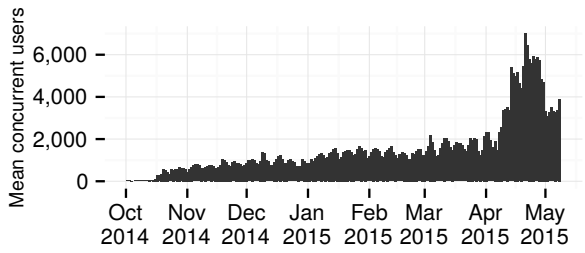
a simple approach: requests and responses are strictly serialized. The client does not send a second chunk of data (i.e., make another request) until it has received the response to its first. The reconstructed stream is simply the concatenation of bodies in the order they arrive. This technique is simple and correct, but less efficient because it needs a full round-trip between every send. See Sections 6 and 7 for alternative approaches that increase efficiency.

meek-server must be able to handle many simultaneous clients. It maintains multiple connections to a local Tor process, one for each active client. The server maps client requests to Tor connections by "session ID," a token randomly generated by the client at startup. The session ID plays the same role in the meek protocol that the (source IP, source port, dest IP, dest port) tuple plays in TCP. The client sends its session ID in a special X-Session-Id HTTP header. meek-server, when it sees a session ID for the first time, opens a new connection to the local Tor process and adds a mapping from ID to connection. Later requests with the same session ID reuse the same Tor connection. Sessions are closed after a period of inactivity. Figure 3 shows a sample of the protocol.

HTTP is fundamentally a request-based protocol. There is no way for the server to "push" data to the client without having first received a request. In order to enable the server to send back data, meek-client sends occasional empty polling requests even when it has no data to send. The polling requests simply give the server an opportunity to send a response. The polling interval starts at 100 ms and grows exponentially up to a maximum of 5 s.

The HTTP-based tunneling protocol adds overhead. Each chunk of data gets an HTTP header, then the HTTP request is wrapped in TLS. The HTTP header adds about 160 bytes [59], and TLS adds another 50 bytes or so (the exact amount depends on the ciphersuite chosen by the intermediate web service). The worst-case overhead when transporting a single encrypted Tor cell of about 540 bytes is about 40%, and less when more than one cell is sent at once. We can estimate how much overhead occurs in practice by examining CDN usage reports. In April 2015, the Amazon CloudFront backend for meek received 3,231 GB in 389 M requests [21], averaging about 8900 bytes per request. If the overhead per request is 210 bytes, then the average overhead is $210/(8900 - 210) \approx 2.4\%$. meek-client reuses the same TLS connection for many requests, so the TLS handshake's overhead is amortized. Polling requests also use bandwidth, but they are sent only when the connection is idle, so they do not affect upload or download speed.

Figure 4 measures the effect of meek's overhead on download speed. It shows the time taken to download a 11,536,384-byte file (http://speedtest.wdc01.softlayer.com/downloads/test10.zip) with and without meek, over the three web services on which we have deployed. We downloaded the file 10 times in each configuration.



	App Engine		CloudFront		Azure (est.)	
	GB	cost	GB	cost	GB	cost
early 2014	71	$8	67	$8	47	$5
Oct 2014	289	$41	479	$130	296	$31
Nov 2014	1,375	$225	1,269	$363	499	$53
Dec 2014	2,132	$327	1,579	$417	511	$64
Jan 2015	2,944	$464	2,449	$669	637	$68
Feb 2015	4,114	$651	2,369	$605	614	$65
Mar 2015	5,316	$690	3,385	$816	736	$78
Apr 2015	6,304	$886	3,231	$785	1,982	$210
total	22,545	$3,292	14,828	$3,793	5,328	$565

Fig. 5. Concurrent users of the meek pluggable transport with month-by-month transfer and cost. The Azure columns are estimates of what we would pay if we did not have a special research grant. User counts come from the Tor Metrics Portal [43, 60].

The time to download the file increases by about a factor of 3 when meek is in use. We attribute this increase to the added latency of an indirect path through the CDN, and the latency-bound nature of meek's naive serialization.

meek's primary deployment vehicle is Tor Browser [51], a derivative of Firefox that is preconfigured to use a built-in Tor client. Tor Browser features an easy interface for enabling meek and other pluggable transports. Deployment began in earnest in October 2014 with the release of Tor Browser 4.0 [50], the first release to include meek as an easy selectable option. It runs over Google App Engine, Amazon CloudFront, and Microsoft Azure. Figure 5 shows the daily average number of concurrent users. (A value of 1,000, for example, means that there were on average 1,000 users of the system at any time during the day.) Also in Figure 5 is a table of monthly costs broken down by web service. Our Azure service is currently running on a free research grant, which does not provide us with billing information. We estimate what Azure's cost would be by measuring the bandwidth used at the backing Tor bridge, and assuming bandwidth costs that match the geographic traffic mix we observe for CloudFront: roughly 62% from North America and Europe, and 38% from other regions.

5.1 Camouflage for the TLS layer

Without additional care, meek would be vulnerable to blocking by its TLS fingerprint. TLS, on which HTTPS is based, has a handshake that is largely plaintext [13, §7.4] and leaves plenty of room for variation between implementations. These differences in implementation make it possible to fingerprint TLS clients [44]. Tor itself was blocked by China in 2011 because of the distinctive ciphersuites it used at the time [57]. Figure 12a in Appendix A shows how meek-client's fingerprint would appear natively; it would be easy to block because not much other software shares the same fingerprint. Figures 12b and 12c show the fingerprints of two web browsers, which are more difficult to block because they also appear in much non-circumvention traffic.

In order to disguise its TLS fingerprint, meek-client proxies all its HTTPS requests through a real web browser. It looks like a browser, because it is a browser. We wrote extensions for Firefox and Chrome that enable them to make HTTPS requests on another program's behalf. The browser running the extension is completely separate from the Tor Browser the user interacts with. It runs in the background in a separate process, does not display a user interface, and shares no state with the user's browser. The extra cost of this arrangement is negligible in terms of latency, because communication with the headless browser occurs over a fast localhost connection, and in terms of CPU and RAM it is the same as running two browsers at once.

The client's Tor process starts both meek-client and the headless browser, then configures meek-client to proxy its requests through the browser. The headless browser is the only component that actually touches the network. It should be emphasized that the headless browser only makes domain-fronted requests to the front domain; the URLs it requests have no relation to the pages the user browses.

6 Deployment on Lantern

Lantern [40] is a free circumvention tool for casual web browsing. It does not employ onion routing and focuses more on performance and availability than on anonymity. Lantern encompasses a network of shared HTTPS proxy servers, and client software that allows censored users to find and use those proxy servers with their existing web browsers. The Lantern client also allows uncensored users to host proxy servers ("peer-

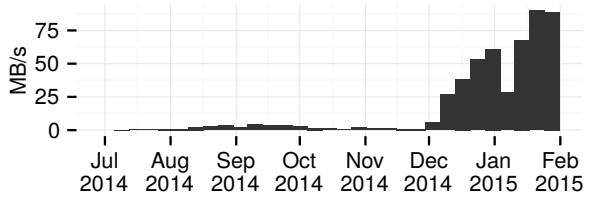


Fig. 6. MB/s served by domain-fronted Lantern proxies (both Lantern-hosted and peers).

hosted" servers) for use by others. Lantern aims to provide a secure mechanism for distributing knowledge about both Lantern-hosted and peer-hosted proxy servers using a trust network–based distribution mechanism such as Kaleidoscope [56]. In the meantime, Lantern also randomly assigns users to Lantern-hosted proxy servers.

Lantern has a centralized infrastructure for authenticating users and assigning them proxies. Its threat model assumes that the centralized infrastructure may be blocked by censors. Therefore, users must have a priori access to an unblocked proxy (a "fallback") which they use to bootstrap into the rest of the network.

Lantern originally distributed the IP addresses of fallbacks by embedding them in customized software installers that we sent to users via email autoresponder. This method prevented users from directly downloading Lantern from our website and would have made it easy for censors to discover proxies simply by signing up for Lantern (though in practice we never saw this happen).

6.1 Implementation

We rolled out domain fronting in July 2014, allowing users to download Lantern directly for the first time. The directly downloaded clients proxied all their traffic via domain fronting. After initial testing with Fastly, we changed to a different CDN, which has proven attractive because it has many unblocked front domains, it does not charge for bandwidth, and its API enables us to easily register and unregister proxies.

Figure 6 shows user bandwidth since deployment. After experiencing steady growth, in October 2014 we started randomly assigning direct HTTPS proxies to users who had direct-downloaded Lantern. This diverted some traffic from domain fronted servers to more efficient direct servers. In December 2014 and January 2015, there was a dramatic surge in domain-fronted traffic, which jumped from 1 MB/s to 100 MB/s within those two months. Activity has remained at around the 100 MB/s level since then.

Lantern's domain fronting support is provided by an application called flashlight [38], which uses library layers called enproxy [37] and fronted [39]. enproxy provides an abstract network connection interface that encodes reads and writes as a sequence of HTTP requests via a stateful enproxy proxy. enproxy allows flashlight to proxy any streaming-oriented traffic like TCP. Unlike Tor's implementation of meek, enproxy supports full-duplex transfer, which is handy for bidirectional protocols like XMPP, which Lantern uses for P2P signaling. fronted uses domain fronting to transmit enproxy's HTTP requests in a blocking-resistant manner. In practice, we configure fronted with several hundred host domains that are dialed via IP address (no DNS lookup).

Domain-fronted Lantern requests go to domain names, such as fallbacks.getiantem.org, that represent pools of servers. The CDN distributes requests to the servers in round-robin fashion. The domain-fronting protocol is stateful, so subsequent HTTP requests for the same connection are routed to the original responding proxy using its specific hostname (sticky routing), which the client obtains from a custom HTTP header. The proxy hostname serves the same request-linking purpose as the session ID does in meek.

6.2 Mitigations for increased latency

The encoding of a stream as a sequence of HTTP requests introduces additional latency beyond that of TCP. In the case of flashlight with our chosen CDN, the additional latency has several causes. We describe the causes and appropriate mitigations.

Domain fronting requires the establishment of additional TCP connections. The client, the CDN, and the proxy between themselves introduce three additional TCP connections between the client and the destination. To reduce latency, the CDN pools and reuses connections to the Lantern proxy. Unfortunately, the Lantern client cannot do the same for its connections to the CDN because the CDN seems to time out idle connections fairly aggressively. We mitigate this by aggressively pre-connecting to the CDN when we detect activity [36].

Though enproxy is mostly full duplex, reads cannot begin until the first request and its response with the sticky-routing header have been processed. This is a basic limitation.

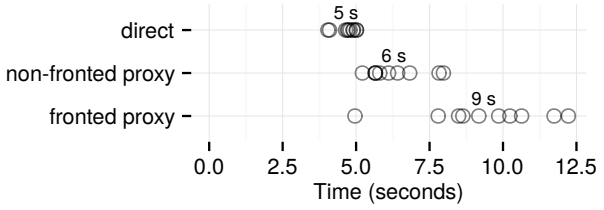


Fig. 7. Time to download an 11 MB file direct, through a one-hop proxy, and through Lantern with domain fronting. The enproxy transport and extra CDN hop increase download times by about a factor of 2. Text labels indicate the mean of 10 measurements.

enproxy does not pipeline HTTP requests. Even though reads and writes are full duplex, a write cannot proceed until the flush of previous writes has been acknowledged with an HTTP response—a full round-trip is necessary between each flush. In the future, HTTP/2's request pipelining will potentially improve on latency.

enproxy has no way of knowing when the client is done writing. If the data were streaming directly from the client through to the proxy, this would not be a problem, but CDNs buffer uploads: small uploads aren't actually forwarded to the proxy until the HTTP request is finished. enproxy assumes that a writer is finished if it detects inactivity for more than 35 ms, at which point it flushes the write by finishing the HTTP request. This introduces at least 35 ms of additional latency, and potentially more if the guess is wrong and the write is not actually finished, since we now have to wait for a full round trip before the next write can proceed. This latency is particularly noticeable when proxying TLS traffic, as the TLS handshake consists of several small messages in both directions.

This last source of latency can be eliminated if enproxy can know for sure when a writer is finished. This could be achieved by letting enproxy handle the HTTP protocol specifically. Doing so would allow enproxy to know when the user agent is finished sending an HTTP request and when the destination is finished responding. However, doing the same for HTTPS would require a local man-in-the-middle attack on the TLS connection in order to expose the flow of requests. Furthermore, this approach would work only for HTTP clients. Other traffic, like XMPP, would require additional support for those protocols.

Figure 7 compares download speeds with and without a domain fronting proxy. It is based on 10 downloads of the same 11 MB file used in the Tor bandwidth test in the previous section, however located on a different server close to the Lantern proxy servers: http://speedtest.ams01.softlayer.com/downloads/test10.zip. The fronted proxy causes download times to approximately double. Because of Lantern's round-robin rotation of front domains, the performance of the fronted proxy may vary over time according to the route to the CDN.

6.3 Direct domain fronting

The Lantern network includes a geolocation server. This server is directly registered on the CDN and the Lantern client domain-fronts to it without using any proxies, reducing latency and saving proxy resources. This sort of direct domain fronting technique could in theory be implemented for any web site simply by registering it under a custom domain such as facebook.direct.getiantem.org. It could even be accomplished for HTTPS, but would require the client software to man-in-the-middle local HTTPS connections between browser and proxy, exposing the plaintext not only to the Lantern client but also to the CDN. In practice, web sites that use the CDN already expose their plaintext to the CDN, so this may be an acceptable solution.

7 Deployment on Psiphon

The Psiphon circumvention system [52] is a centrally managed, geographically diverse network of thousands of proxy servers. It has a performance-oriented, one-hop architecture. Much of its infrastructure is hosted with cloud providers. As of January 2015, Psiphon has over two million daily unique users. Psiphon client software runs on popular platforms, including Windows and Android. The system is designed to tunnel a broad range of

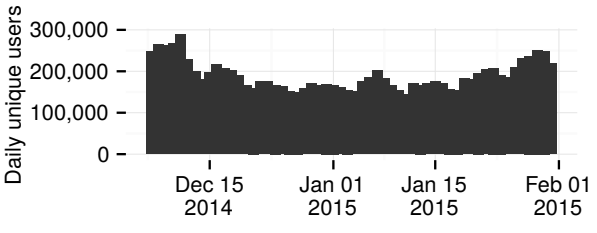


Fig. 8. Daily unique users of meek with Psiphon. Clients send a coarse-grained "last connected" timestamp when connecting. A unique user is counted whenever a user connects with a timestamp before the day under consideration.

host traffic: web browsing, video streaming, and mobile app data transfer. Client software is designed for ease of use; users are not asked to perform any configuration.

Psiphon has faced threats including blocking by DPI—both blacklisting and whitelisting—and blocking by address. For example, in 2013, Psiphon circumvented HTTP-whitelisting DPI by sending an "HTTP prefix" (the first few bytes of an HTTP request) before the start of its regular upstream flow [6].

Psiphon strives to distribute its server addresses in such a way that most clients discover enough servers to have several options in the case of a server being blocked, while making it difficult to enumerate all servers. In February 2014, Psiphon was specifically targeted for address-based blocking, and this blocking was aggressive enough to have a major impact on our user base, though not all users were blocked. As part of our response we integrated and deployed meek-based domain fronting, largely based on Tor's implementation, with some modifications. It was fully deployed in June 2014. Figure 8 shows the number of unique daily users of fronted meek with Psiphon.

In addition, Psiphon also employs meek in what we call "unfronted" mode. Unfronted meek omits the TLS layer and the protocol on the wire is HTTP. As fully compliant HTTP, unfronted meek supersedes the "HTTP prefix" defense against HTTP whitelisting. Unfronted meek is not routed through CDNs, and as such is only a defense against DPI whitelisting and not against proxy address enumeration. We envision a potential future fronted HTTP protocol with both properties, which requires cooperating with CDNs to route our HTTP requests based on, for example, some obfuscated HTTP header element.

7.1 Implementation

Psiphon's core protocol is SSH. SSH provides an encryption layer for communication between Psiphon clients and servers; the primary purpose of this encryption is to frustrate DPI. On top of SSH, we add an obfuscated-openssh [41] layer that transforms the SSH handshake into a random stream, and add random padding to the handshake. The payload within the meek transport appears to be random data and lacks a trivial packet size signature in its initial requests and responses. Psiphon clients authenticate servers using SSH public keys obtained out of band, a process that is bootstrapped with server keys embedded in the client binaries.

Psiphon uses a modified version of the meek protocol described in Section 5. The session ID header contains extra information: a protocol version number and the destination Psiphon server address. As this cookie will be visible to the censor in unfronted mode, its value is encrypted in a NaCl crypto_box [7] using the public key of the destination meek-server; then obfuscated; then formatted as an innocuous-seeming cookie with a randomly selected key. meek-server uses the protocol version number to determine if the connecting meek-client supports Psiphon-specific protocol enhancements. The destination address is the SSH server to which meek-server should forward traffic.

In Psiphon, meek-client transmits its chosen session ID on its first HTTP request, after which meek-server assigns a distinct ID to be used on subsequent requests. This change allows meek-server to distinguish new and existing sessions when a client sends a request after a long delay (such as after an Android device awakes from sleeping), when meek-server may have already expired and discarded its session.

We ported meek-client, originally written in Go, to Java for Android. On Android, we make HTTP and HTTPS requests using the Apache HttpClient component, in order to have a TLS fingerprint like those of other Android apps making web service requests.

The Psiphon meek-server inspects CDN-injected headers, like X-Forwarded-For, to determine the client's IP address. The address is mapped to a geographic region that is used in recording usage statistics.

7.2 Server selection

When a user starts a Psiphon client, the client initiates connections to up to ten different servers simultaneously, keeping the first to be fully established. Candidate servers are chosen at random from cached lists of known servers and a mix of different protocols, both fronted and non-fronted, are used. The purpose of the simultaneous connections is to minimize user wait time in case certain protocols are blocked, certain servers are blocked by address, or certain servers are at capacity and rejecting new connections. This process also tends to pick the closest data center, and the one with lowest cost, as it tends to pick lower-latency direct connections over domain-fronted connections.

We made two modifications to server selection in order to accommodate fronting. First, we changed the notion of an "established connection" from TCP connection completion to full SSH handshake completion.

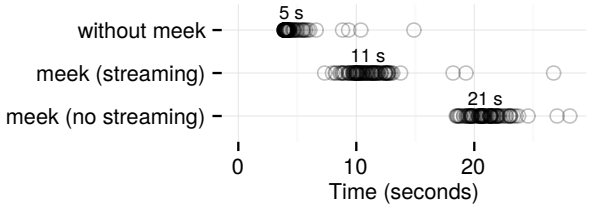


Fig. 9. Time to download an 11 MB file through Psiphon over three transports: one-hop obfuscated-openssh proxy without meek; meek with streaming downloads; and meek without streaming downloads. Text labels indicate the mean of the 50 fastest measurements out of 100. Bulk-download times increase by about a factor of 3 when meek is activated. With fixed-size HTTP bodies, meek costs about a factor of 4 in download time. With the optimization of unlimited-size HTTP bodies, the download time decreases to about a factor of 2.

This ensures that both hops are measured in the fronted case. Second, we adjusted our protocol selection schedule to ensure that, while we generally favor the fastest connection, we do not expose the system to an attack that would force us to use a degraded protocol. For example, a censor could use a DPI attack that allows all connections to establish, but then terminate or severely throttle non-whitelisted protocols after some short time period. If the client detects such degraded conditions, it begins to favor fronted and unfronted protocols over the faster obfuscated SSH direct connections.

7.3 Performance

We identified a need to improve the video streaming and download performance of meek-tunneled traffic. In addressing this, we considered the cost per HTTP request of some candidate CDNs, a lack of support for HTTP pipelining in our components, and a concern about the DPI signature of upstream-only or downstream-only HTTP connections. As a compromise between these considerations, we made a tweak to the meek protocol: instead of sending at most 64 KB in each HTTP response, responses stream as much as possible, as long as there is data to send and for up to 200 ms.

This tweak yielded a significant performance improvement, with download speeds increasing by up to 4–5×, and 1080p video playback becoming smooth. Under heavy downstream conditions, we observe response bodies up to 1 MB, 300 KB on average, although the exact traffic signature is highly dependent on the tunneled application. We tuned the timeout parameter through subjective usability testing focused on latency while web browsing and simultaneously downloading large files.

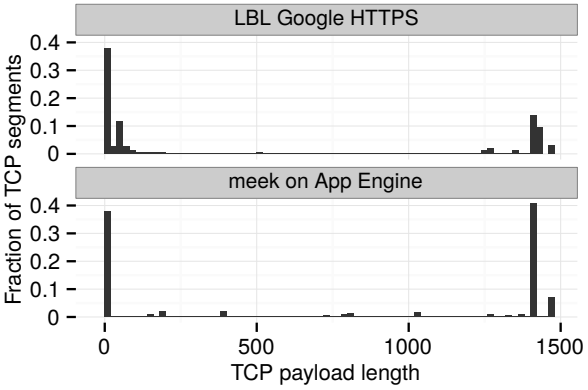
Figure 9 compares the time taken to download a file both with and without meek, and with and without the streaming download optimization. The target is the same speedtest.wdc01 URL used in the Tor performance tests in Section 5. The performance effect of meek is about a factor-4 increase in download time; streaming downloads cut the increase in half.

8 Traffic analysis

In developing domain fronting circumvention systems, we hope to deprive the censor of easy distinguishers and force the use of more expensive, less reliable classification tests—generally, to increase the cost of censorship. We believe that domain fronting, implemented with care, meets the primary challenges of proxy-based circumvention. It defeats IP- and DNS-based blocking because the IP- and DNS-layer information seen by the censor are not those of the proxy; content-based blocking because content is encrypted under HTTPS; and active probing because though a censor may be able to discover that a web service is used for circumvention, it cannot block the service without incurring significant collateral damage.

Our experiences with deploying circumvention systems has led us to conclude that other potential means of censorship—e.g., identifying circumventing content by analyzing packet length distributions—do not currently have relevance when considering the practices of today's censors. We speculate that censors find such tests unattractive because they require storing significant state and are susceptible to misclassification. More broadly, we are not aware of any nation-level censorship event that made use of such traffic features.

Nevertheless, we expect censors to adapt to a changing environment and to begin deploying more sophisticated (but also more expensive and less reliable) tests. The issue of traffic analysis is a general one [68], and mostly separable from domain fronting itself. That is, domain fronting does not preclude various traffic shaping techniques and algorithms, which can be developed independently and plugged in when the censors of the world make them necessary. This section contains a sketch of domain fronting's resistance to certain traffic analysis features, though a case study of meek with Tor and a trace of non-circumvention traffic. While we identify some features that may give a censor leverage



LBL Google HTTPS		meek on App Engine	
0 bytes	37.6%	1418 bytes	40.5%
1430 bytes	9.1%	0 bytes	37.7%
1418 bytes	8.5%	1460 bytes	7.2%
41 bytes	6.1%	396 bytes	2.0%
1416 bytes	3.1%	196 bytes	1.8%
1460 bytes	2.9%	1024 bytes	1.5%

Fig. 10. Comparison of TCP payload length distributions in ordinary HTTPS connections to Google services from the LBL traffic trace, and meek running on App Engine, fronted through www.google.com.

in distinguishing circumvention traffic, we believe that the systems we have deployed are sufficiently resistant to the censors of today, and do not block the way to future enhancements to traffic analysis resistance.

As domain fronting is based on HTTPS, we evaluate distinguishability from "ordinary" HTTPS traffic. We compare two traffic traces. The first is HTTPS traffic from Lawrence Berkeley National Laboratory (LBL), a large (≈ 4K users) research lab, comprising data to and from TCP port 443 on any Google server. Its size is 313 MB (packet headers only, not payloads) and it lasts 10 minutes. The IP addresses in this first trace were masked, replaced by a counter. The second trace is of meek in Tor Browser, browsing the home pages of the top 500 Alexa web sites over Google and App Engine. It is 687 MB in size and covers 4.5 hours.

8.1 Packet length distribution

A censor could attempt to block an encrypted tunnel by its distribution of packet lengths, if it is distinctive enough. Figure 10 compares the packet length distributions of the sample traces. Keeping in mind that the LBL trace represents many users, operating systems, and web browsers, and the meek trace only one of each,

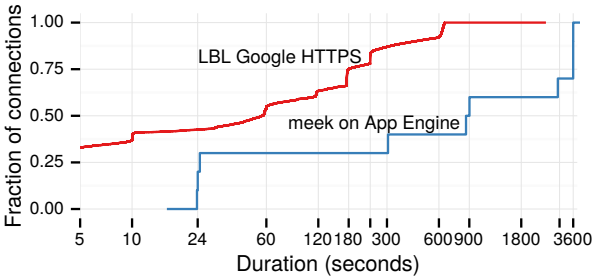


Fig. 11. CDF of connection duration. The x-axis is logarithmic.

the two are not grossly different. In both cases, about 38% of packets are empty (mostly ACKs), with many packets near the usual TCP Maximum Segment Size of 1460 bytes. Conspicuous in the meek trace are a small peaks at a few specific lengths, and a lack of short payloads of around 50 bytes. Both of characteristics are probably reflections of the fixed cell size of the underlying Tor stream.

8.2 Connection lifetime

The total duration of TCP connections is another potential distinguisher. Figure 11 shows the cumulative probability of connection durations in the two traces. The LBL trace has interesting concentrations on certain round numbers: 10/60/120/180/240 seconds. We hypothesize that they are caused by keepalive timeouts in web browsers and servers and periodic polling by web apps. The small rise at 600 seconds is an artifact caused by the 10-minute duration of the trace. We do not know how much longer than 10 minutes those connections lasted, but they are only 8% of observed connections.

The meek trace shows a propensity for longer connections. In 4.5 hours, there were only 10 connections, three of them lasting for an hour. The long connections are caused by the client browser extension's aggressive use of long-lived HTTP keepalive connections, and by its being constantly busy, giving every opportunity for connection reuse. 60% of meek's connections lasted five minutes or longer, while only 13% of ordinary traffic's did. meek had essentially no connections lasting less than 24 seconds, but such short connections were over 42% of the LBL trace. 30% (3 out of 10) of meek's connections lasted almost exactly one hour, evidently reflecting a built-in keepalive limit in either the client browser extension or in App Engine.

In light of these measurements, the censor may decide simply to terminate long-lived HTTPS connections.

According to our traffic trace, doing so will not disrupt more than 8% of ordinary connections (although such long connections may be valuable large transfers with higher collateral damage). The censor can lower the timing threshold, at the cost of more false positives. In order to be effective, then censor must cut off the client completely; otherwise the client may start a new connection with the same session ID and begin where it left off.

We do not know of any obvious traffic characteristics that reliably distinguish domain fronting from other HTTPS traffic. Long-lived connections and packet lengths are potential targets for a more concerted attack. We are fundamentally trying to solve a problem of steganography, to make circumvention traffic fit some model of "normal" traffic. However, this can be regarded as an advantage. What is a challenge for the evaluator is also a challenge for the censor, simply because it is difficult to characterize just what normal traffic *is*, especially behind a CDN that may host variety of services such as software updates, video streaming, and ordinary web pages. Circumvention traffic need not be perfectly indistinguishable, only indistinguishable enough that that blocking it causes more and costlier false positives than the censor can accept.

9 Discussion

Domain fronting derives its strength from the collateral damage that results from blocking the front domain. It should not—nor should any other circumvention technique—be thought of as unblockable; rather, one should think of what it costs the censor to block it. What is unblockable by one censor may be blocked by another that has different resources and incentives. Blocking resistance depends on the strength of the front domain and on the censor's cost calculus, which has both economic and social components.

We can at least roughly quantify the cost of blocking any domain fronting system in general. It is the minimum cost of: blocking a domain; deploying traffic analysis to distinguish circumvention from other traffic; or conducting some attack outside our threat model, for example physical surveillance of Internet users. A censor could also, for example, block HTTPS entirely, but that is likely to be even more damaging than targeted blocking of a domain. The cost of blocking a domain—and the benefit of blocking circumvention—will vary by censor. For example, China can afford to block twitter.com and facebook.com partly because it has domestic replacements for those services, but not all censors have the same resources. In June 2014, the Great Firewall of China took the unprecedented step of blocking all Google services [2, 25], including all potential fronts for App Engine. It is not clear whether the blocking targeted domain fronting systems like GoAgent; our own systems were only prototypes at that point. Since then, domain fronting to App Engine has been effectively stamped out in China, though it continues to work over other web services.

A censor could directly confront the operators of an intermediate web service and ask them to disable domain fronting (or simply get rid of customers like us who facilitate circumvention). The censor could threaten to block the service entirely, costing it business. Whether such an attack succeeds again depends on specific costs and motivations. A powerful censor may be able to carry out its threat, but others will harm themselves more by blocking a valuable service than the circumvention traffic is worth.

Reliance on paid web services creates the potential for a "financial denial of service" attack against domain fronting systems, in which the censor uses the service excessively in an attempt to drive up the operators' costs. In March 2015, the anticensorship group GreatFire, which had used various cloud services for censorship circumvention in China, was the target of a distributed denial of service attack against their hosting on Amazon Web Services [55]. The attack lasted for days and incurred tens of thousands of dollars in bandwidth charges. The attack against Amazon was followed shortly by one against GitHub, the largest in the site's history [48]. The second attack specifically targeted GreatFire's accounts there. The available evidence indicates that both attacks were coordinated from within China, using an offensive network system dubbed the "Great Cannon" [45]. Such an attack could be mitigated by active defenses that shut down a service when it is being used excessively, though this only protects against ruinous costs and will not defeat a long-term attack. It is noteworthy that a world-class censor's first reaction was a disruptive, unsubtle denial of service attack—though we cannot say for sure that the censor did not have something better up its sleeve. GreatFire speculated that the attacks were precipitated by the publication of an article in the *Wall Street Journal* [16] that described in detail domain fronting and other "collateral freedom" techniques. The interview associated with the article also caused CloudFlare to begin matching SNI and Host header, in an apparent attempt to thwart domain fronting.

Fronting shares a potential weakness with decoy routing, which is that the network paths to the overt and covert destinations diverge. The difference in paths may create side channels—different latencies for instance—that distinguish domain-fronted traffic from the traffic that really arrives at its apparent destination. For example, a CDN can be expected to have responses to some fraction of requests already in cache, and respond to those requests with low latency, while domain-fronted requests always go all the way to the destination with higher latency. Schuhard et al. [54, §5] applied latency measurement to decoy routing. The authors of TapDance [69, §5.1] observe that such an attack is difficult to carry out in practice, because it requires knowledge of the performance characteristics of many diverse resources behind the proxy, some of which are not accessible to the censor (login-protected web pages, for example). Domain fronting favors the circumventor even more, because of the variety of resources behind a CDN.

The intermediate web service has a privileged network position from which it may monitor domain-fronted traffic. Even though the censor does not know which client IP addresses are engaging in circumvention, the CDN knows. The risk is especially acute when client browses a web site of the same entity that controls the intermediate web server, for example browsing YouTube while fronting through www.google.com. When this happens, the web service gets to see both entry and exit traffic, and is in a better position to attempt to correlate flows by timing and volume, even when the underlying channel is an encrypted protocol like Tor. This phenomenon seems hard to counter, because the front domain needs to be a popular one in order to have high collateral damage, but popular domains are also the ones that users tend to want to visit. It is in theory possible to dynamically switch between multiple fronts, so as to avoid the situation where the destination and front are under the same control, at the cost of leaking information about where the user is *not* going at a given moment.

A censor that can man-in-the-middle HTTPS connections can detect domain fronting merely by removing encryption and inspecting the Host header. Unless the censor controls a certificate authority, this attack falls to ordinary HTTPS certificate validation. Against a censor that controls a trusted certificate authority, certificate pinning is an effective defense. If the underlying transport is an authenticated and encrypted one like Tor, then the destination and contents of a user's connection will remain secret, even if the user is outed as a circumventor.

10 Summary

We have presented domain fronting, an application-layer censorship circumvention technique that uses different domain names at different layers of communication in order to hide the true destination of a message. Domain fronting resists the main challenges offered by the censors of today: content blocking, address blocking, and active probing. We have implemented domain fronting in three popular circumvention systems: Tor, Lantern, and Psiphon, and reported on the experience of deployment. We begin an investigation into the more difficult, less reliable means of traffic analysis that we believe will be necessary to block domain fronting.

Code and acknowledgments

The meek pluggable transport has a home page at https://trac.torproject.org/projects/tor/wiki/doc/meek and source code at https://gitweb.torproject.org/pluggable-transports/meek.git. The source code of Lantern's flashlight proxy is at https://github.com/getlantern/flashlight; other components are in sibling repositories. Psiphon's source code is at https://bitbucket.org/psiphon/psiphon-circumvention-system.

We would like to thank Yawning Angel, George Kadianakis, Georg Koppen, Lunar, and the members of the tor-dev, tor-qa, and traffic-obf mailing lists who responded to our design ideas, reviewed source code, and tested our prototypes. Arlo Breault wrote the flashproxy-reg-appspot program mentioned in Section 3, an early application of domain fronting. Leif Ryge and Jacob Appelbaum tipped us off that domain fronting was possible. Sadia Afroz, Michael Tschantz, and Doug Tygar were sources of inspiring conversation. Johanna Amann provided us with an estimate of the fraction of SNI-bearing TLS handshakes.

This work was supported in part by the National Science Foundation under grant 1223717. The opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Akamai. http://www.akamai.com/.

[2] P. Alpha. Google disrupted prior to Tiananmen anniversary; mirror sites enable uncensored access to information, June 2014. https://en.greatfire.org/blog/2014/jun/google-disrupted-prior-tiananmen-anniversary-mirror-sites-enable-uncensored-access.

[3] Amazon CloudFront. https://aws.amazon.com/cloudfront/.

[4] Y. Angel and P. Winter. obfs4 (the obfourscator), May 2014. https://gitweb.torproject.org/pluggable-transports/obfs4.git/tree/doc/obfs4-spec.txt.

[5] J. Appelbaum and N. Mathewson. Pluggable transport specification, Oct. 2010. https://gitweb.torproject.org/torspec.git/tree/pt-spec.txt.

[6] ASL19 and Psiphon. Information controls: Iran's presidential elections. Technical report, 2013. https://asl19.org/cctr/iran-2013election-report/.

[7] D. J. Bernstein, T. Lange, and P. Schwabe. Public-key authenticated encryption: crypto_box, Aug. 2010. http://nacl.cr.yp.to/box.html.

[8] B. Boe. Bypassing Gogo's inflight Internet authentication, Mar. 2012. http://bryceboe.com/2012/03/12/bypassing-gogos-inflight-internet-authentication/.

[9] BridgeDB. https://bridges.torproject.org/.

[10] C. Brubaker, A. Houmansadr, and V. Shmatikov. CloudTransport: Using cloud storage for censorship-resistant networking. In *Proceedings of the 14th Privacy Enhancing Technologies Symposium (PETS)*, July 2014. http://www.cs.utexas.edu/~amir/papers/CloudTransport.pdf.

[11] S. Burnett, N. Feamster, and S. Vempala. Chipping away at censorship firewalls with user-generated content. In *USENIX Security Symposium*, Washington, DC, USA, Aug. 2010. USENIX. https://www.usenix.org/event/sec10/tech/full_papers/Burnett.pdf.

[12] CloudFlare. https://www.cloudflare.com/.

[13] T. Dierks and E. Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2, Aug. 2008. https://tools.ietf.org/html/rfc5246.

[14] R. Dingledine. Obfsproxy: the next step in the censorship arms race, Feb. 2012. https://blog.torproject.org/blog/obfsproxy-next-step-censorship-arms-race.

[15] R. Dingledine and N. Mathewson. Design of a blocking-resistant anonymity system. Technical Report 2006-11-001, Tor Project, Nov. 2006. https://research.torproject.org/techreports/blocking-2006-11.pdf.

[16] E. Dou and A. Barr. U.S. cloud providers face backlash from China's censors. Wall Street Journal, Mar. 2015. http://www.wsj.com/articles/u-s-cloud-providers-face-backlash-from-chinas-censors-1426541126.

[17] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 20th ACM conference on Computer and Communications Security (CCS)*, Nov. 2013. https://kpdyer.com/publications/ccs2013-fte.pdf.

[18] D. Eastlake. RFC 6066: Transport Layer Security (TLS) extensions: Extension definitions, Jan. 2011. https://tools.ietf.org/html/rfc6066.

[19] Fastly. http://www.fastly.com/.

[20] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol — HTTP/1.1, June 1999. https://tools.ietf.org/html/rfc2616.

[21] D. Fifield. Summary of meek's costs, April 2015, May 2015. https://lists.torproject.org/pipermail/tor-dev/2015-May/008767.html.

[22] D. Fifield, N. Hardison, J. Ellithorpe, E. Stark, R. Dingledine, P. Porras, and D. Boneh. Evading censorship with browser-based proxies. In *Proceedings of the 12th Privacy Enhancing Technologies Symposium (PETS)*. Springer, July 2012. https://crypto.stanford.edu/flashproxy/flashproxy.pdf.

[23] J. Geddes, M. Schuchard, and N. Hopper. Cover your ACKs: Pitfalls of covert channel censorship circumvention. In *Proceedings of the 20th ACM conference on Computer and Communications Security (CCS)*, Nov. 2013. http://www-users.cs.umn.edu/~hopper/ccs13-cya.pdf.

[24] GoAgent. https://github.com/goagent/goagent.

[25] Google. Google Transparency Report: China, all products, May 31, 2014–present, July 2014. https://www.google.com/transparencyreport/traffic/disruptions/124/.

[26] Google App Engine. https://cloud.google.com/appengine/.

[27] GreatFire.org. https://a248.e.akamai.net is 100% blocked in China. https://en.greatfire.org/https/a248.e.akamai.net.

[28] A. Houmansadr, C. Brubaker, and V. Shmatikov. The parrot is dead: Observing unobservable network communications. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, May 2013. http://www.cs.utexas.edu/~amir/papers/parrot.pdf.

[29] A. Houmansadr, G. T. K. Nguyen, M. Caesar, and N. Borisov. Cirripede: Circumvention infrastructure using router redirection with plausible deniability. In *Proceedings of the 18th ACM conference on Computer and Communications Security (CCS)*, Oct. 2011. http://hatswitch.org/~nikita/papers/cirripede-ccs11.pdf.

[30] A. Houmansadr, T. Riedl, N. Borisov, and A. Singer. I want my voice to be heard: IP over voice-over-IP for unobservable censorship circumvention. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*. Internet Society, Feb. 2013. http://www.cs.utexas.edu/~amir/papers/FreeWave.pdf.

[31] A. Houmansadr, E. L. Wong, and V. Shmatikov. No direction home: The true cost of routing around decoys. In *Proceedings of the 21st Network and Distributed Security Symposium (NDSS)*. Internet Society, Feb. 2014. http://www.cs.utexas.edu/~amir/papers/DecoyCosts.pdf.

[32] The ICSI certificate notary. http://notary.icsi.berkeley.edu/.

[33] G. Kadianakis and N. Mathewson. obfs2 (the twobfuscator), Jan. 2011. https://gitweb.torproject.org/pluggable-transports/obfsproxy.git/tree/doc/obfs2/obfs2-protocol-spec.txt.

[34] G. Kadianakis and N. Mathewson. obfs3 (the threebfuscator), Jan. 2013. https://gitweb.torproject.org/pluggable-transports/obfsproxy.git/tree/doc/obfs3/obfs3-protocol-spec.txt.

[35] J. Karlin, D. Ellard, A. W. Jackson, C. E. Jones, G. Lauer, D. P. Mankins, and W. T. Strayer. Decoy routing: Toward unblockable internet communication. In *Proceedings of the USENIX Workshop on Free and Open Communications on*

the Internet (FOCI), Aug. 2011. https://www.usenix.org/events/foci11/tech/final_files/Karlin.pdf.

[36] Lantern. connpool. https://github.com/getlantern/connpool.

[37] Lantern. enproxy. https://github.com/getlantern/enproxy.

[38] Lantern. flashlight. https://github.com/getlantern/flashlight-build.

[39] Lantern. fronted. https://github.com/getlantern/fronted.

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

[41] B. Leidl. obfuscated-openssh, Apr. 2010. https://github.com/brl/obfuscated-openssh.

[42] Level 3. http://www.level3.com.

[43] K. Loesing. Counting daily bridge users. Technical Report 2012-10-001, Tor Project, Oct. 2012. https://research.torproject.org/techreports/counting-daily-bridge-users-2012-10-24.pdf.

[44] M. Majkowski. SSL fingerprinting for p0f, June 2012. https://idea.popcount.org/2012-06-17-ssl-fingerprinting-for-p0f/.

[45] B. Marczak, N. Weaver, J. Dalek, R. Ensafi, D. Fifield, S. McKune, A. Rey, J. Scott-Railton, R. Deibert, and V. Paxson. China's Great Cannon. https://citizenlab.org/2015/04/chinas-great-cannon/.

[46] Microsoft Azure. https://azure.microsoft.com/.

[47] H. M. Moghaddam, B. Li, M. Derakhshani, and I. Goldberg. SkypeMorph: Protocol obfuscation for Tor bridges. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS)*, Oct. 2012. https://cs.uwaterloo.ca/~iang/pubs/skypemorph-ccs.pdf.

[48] J. Newland. Large scale DDoS attack on github.com. https://github.com/blog/1981-large-scale-ddos-attack-on-github-com.

[49] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai network: A platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010. http://www.akamai.com/dl/technical_publications/network_overview_osr.pdf.

[50] M. Perry. Tor Browser 4.0 is released, Oct. 2014. https://blog.torproject.org/blog/tor-browser-40-released.

[51] M. Perry, E. Clark, and S. Murdoch. The design and implementation of the Tor Browser. Technical report, Tor Project, Mar. 2013. https://www.torproject.org/projects/torbrowser/design/.

[52] Psiphon Team. A technical description of Psiphon, Mar. 2014. https://psiphon.ca/en/blog/psiphon-a-technical-description.

[53] D. Robinson, H. Yu, and A. An. Collateral freedom: A snapshot of Chinese users circumventing censorship. Technical report, Open Internet Tools Project, May 2013. https://openitp.org/pdfs/CollateralFreedom.pdf.

[54] M. Schuchard, J. Geddes, C. Thompson, and N. Hopper. Routing around decoys. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS)*, Oct. 2012. http://www-users.cs.umn.edu/~hopper/decoy-ccs12.pdf.

[55] C. Smith. We are under attack, Mar. 2015. https://en.greatfire.org/blog/2015/mar/we-are-under-attack.

[56] Y. Sovran, J. Li, and L. Submaranian. Unblocking the Internet: Social networks foil censors. Technical Report TR2008-918, Computer Science Department, New York University, Sept. 2009. http://kscope.news.cs.nyu.edu/pub/TR-2008-918.pdf.

[57] Tor Project. #4744: GFW probes based on Tor's SSL cipher list, Dec. 2011. https://bugs.torproject.org/4744.

[58] Tor Project. #8860: Registration over App Engine, May 2013. https://bugs.torproject.org/8860.

[59] Tor Project. #12778: Put meek HTTP headers on a diet, Aug. 2014. https://bugs.torproject.org/12778.

[60] Tor Project. Bridge users using transport meek, May 2015. https://metrics.torproject.org/userstats-bridge-transport.html?graph=userstats-bridge-transport&end=2015-05-15&transport=meek.

[61] Tor Project. Bridge users using transport obfs3, May 2015. https://metrics.torproject.org/userstats-bridge-transport.html?graph=userstats-bridge-transport&end=2015-05-15&transport=obfs3.

[62] Q. Wang, X. Gong, G. T. K. Nguyen, A. Houmansadr, and N. Borisov. CensorSpoofer: Asymmetric communication using IP spoofing for censorship-resistant web browsing. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS)*, Oct. 2012. https://netfiles.uiuc.edu/qwang26/www/publications/censorspoofer.pdf.

[63] 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 19th ACM conference on Computer and Communications Security (CCS)*, Oct. 2012. http://www.owlfolio.org/media/2010/05/stegotorus.pdf.

[64] T. Wilde. Great Firewall Tor probing circa 09 DEC 2011. Technical report, Team Cymru, Jan. 2012. https://gist.github.com/da3c7a9af01d74cd7de7.

[65] B. Wiley. Dust: A blocking-resistant internet transport protocol. Technical report, School of Information, University of Texas at Austin, 2011. http://blanu.net/Dust.pdf https://github.com/blanu/Dust/blob/master/hs/README.

[66] P. Winter and S. Lindskog. How the Great Firewall of China is blocking Tor. In *Proceedings of the USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, Aug. 2012. https://www.usenix.org/system/files/conference/foci12/foci12-final2.pdf.

[67] P. Winter, T. Pulls, and J. Fuss. ScrambleSuit: A polymorphic network protocol to circumvent censorship. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES)*. ACM, Nov. 2013. http://www.cs.kau.se/philwint/pdf/wpes2013.pdf.

[68] C. Wright, S. Coull, and F. Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *Proceedings of the 16th Network and Distributed Security Symposium (NDSS)*. IEEE, Feb. 2009. https://www.internetsociety.org/sites/default/files/wright.pdf.

[69] E. Wustrow, C. M. Swanson, and J. A. Halderman. TapDance: End-to-middle anticensorship without flow blocking. In *Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, Aug. 2014. USENIX Association. https://jhalderm.com/pub/papers/tapdance-sec14.pdf.

[70] E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman. Telex: Anticensorship in the network infrastructure. In *Proceedings of the 20th USENIX Security Symposium*, Aug. 2011. https://www.usenix.org/events/sec/tech/full_papers/Wustrow.pdf.

A Sample TLS fingerprints

(a) Go 1.4.2's crypto/tls library

```
Ciphersuites (13):
  TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
  TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
  TLS_ECDHE_RSA_WITH_RC4_128_SHA
  TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
  TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
  TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
  TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
  TLS_RSA_WITH_RC4_128_SHA
  TLS_RSA_WITH_AES_128_CBC_SHA
  TLS_RSA_WITH_AES_256_CBC_SHA
  TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
  TLS_RSA_WITH_3DES_EDE_CBC_SHA
Extensions (6):
  server_name
  status_request
  elliptic_curves
  ec_point_formats
  signature_algorithms
  renegotiation_info
```

(b) Firefox 31

```
Ciphersuites (23):
  TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
  TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
  TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
  TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
  TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
  TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
  TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
  TLS_ECDHE_RSA_WITH_RC4_128_SHA
  TLS_DHE_RSA_WITH_AES_128_CBC_SHA
  TLS_DHE_DSS_WITH_AES_128_CBC_SHA
  TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
  TLS_DHE_RSA_WITH_AES_256_CBC_SHA
  TLS_DHE_DSS_WITH_AES_256_CBC_SHA
  TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
  TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
  TLS_RSA_WITH_AES_128_CBC_SHA
  TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
  TLS_RSA_WITH_AES_256_CBC_SHA
  TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
  TLS_RSA_WITH_3DES_EDE_CBC_SHA
  TLS_RSA_WITH_RC4_128_SHA
  TLS_RSA_WITH_RC4_128_MD5
Extensions (8):
  server_name
  renegotiation_info
  elliptic_curves
  ec_point_formats
  SessionTicket TLS
  next_protocol_negotiation
  status_request
  signature_algorithms
```

(c) Chrome 40

```
Ciphersuites (18):
  TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
  TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
  TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
  TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
  TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
  TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
  TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
  TLS_ECDHE_RSA_WITH_RC4_128_SHA
  TLS_DHE_RSA_WITH_AES_128_CBC_SHA
  TLS_DHE_DSS_WITH_AES_128_CBC_SHA
  TLS_DHE_RSA_WITH_AES_256_CBC_SHA
  TLS_RSA_WITH_AES_128_GCM_SHA256
  TLS_RSA_WITH_AES_128_CBC_SHA
  TLS_RSA_WITH_AES_256_CBC_SHA
  TLS_RSA_WITH_3DES_EDE_CBC_SHA
  TLS_RSA_WITH_RC4_128_SHA
  TLS_RSA_WITH_RC4_128_MD5
Extensions (10):
  server_name
  renegotiation_info
  elliptic_curves
  ec_point_formats
  SessionTicket TLS
  next_protocol_negotiation
  Application Layer Protocol Negotiation
  Channel ID
  status_request
  signature_algorithms
```

Fig. 12. Selected differences in ClientHello messages in three different TLS implementations. Even though the contents of application data records are hidden by encryption, the plaintext headers of TLS reveal information about the implementation. This figure illustrates the need to disguise the TLS fingerprint so that it is not easily identified as pertaining to a circumvention tool.