Blocking-resistant communication through domain fronting

Abstract—**We describe "domain fronting," an application-layer technique for HTTPS that hides the remote endpoint of a communication for the purpose of censorship circumvention. Domain fronting enables communication that is apparently with an allowed domain, but actually with a forbidden domain. The key idea is the use of different domain names at different layers of communication. One domain is used on the "outside" of an HTTPS request—in the DNS request and TLS Server Name Indication—while another is used on the "inside"—in the HTTP Host header, invisible to the censor under HTTPS encryption. We identify a number of hard-to-block web services, like content delivery networks and Google, that support fronting because they ignore the outside of an HTTPS request and dispatch it internally according to the Host header. If a censor is unable to distinguish fronted circumvention traffic and non-fronted ordinary traffic, then blocking fronted traffic means blocking an entire web service, with resulting expensive collateral damage.**

We have implemented domain fronting in a system called meek, a pluggable transport for Tor. meek combines fronting with an HTTP-based tunneling proxy. The meek server's external interface is that of an ordinary web server. Domain fronting enables a client to reach the server, which is presumed blocked by the censor. The server receives HTTP requests, decodes them, and feeds their payloads into a Tor relay, returning downstream data in the HTTP response. A censor watching the communication sees only a sequence of HTTPS requests to an allowed domain. We describe the results of an initial deployment. meek, or something based on it, has been adopted by other circumvention systems including [blinded for anonymous submission].

Hiding the endpoint and obscuring byte patterns are important parts of blocking-resistant communication, but there are other, more subtle considerations. We describe what we have done to disguise other traffic "tells" in meek, such as using a real web browser as a tool for making fronted HTTPS requests, in order to disguise meek's TLS fingerprint. We argue that these measures, combined with domain fronting, increase the censor's effort beyond simple one-shot blocking techniques such as IP address blocking, and into the realm of more expensive, less reliable statistical tests.

I. 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 use technical and social means to gain access to the forbidden information. What has emerged is an ongoing conflict between censor and censored, with advances on both sides, more subtle evasion being countered by more powerful detection.

What we as circumventors have in our favor is the censor's distaste for "collateral damage," accidental overblocking committed in the course of trying to censor something else. One way to win against censorship is to entangle circumvention traffic with ordinary Internet traffic in a way that both must be blocked together, or not at all. If the ordinary traffic is valuable enough, above the censor's tolerance for overblocking, then the circumvention traffic will get through.

In this paper we describe "domain fronting," a general-purpose 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 an HTTPS frontend server. These include such high–collateral damage infrastructure as big content delivery networks (CDNs) and Google's panoply of services—a nontrivial fraction of the web. See Section VI for a list of suitable services. Domain fronting can be used to tunnel traffic to a general-purpose proxy, so that circumvention is not limited only to HTTPS, nor to the domains of any specific web service.

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 places relevant to the discussion: in the DNS query, in the TLS Server Name Indication (SNI) extension [1, Section 3], and in the HTTP Host header [2, Section 14.23]. Usually, the same domain name appears in all three places. But in a domain-fronted request, the DNS query and SNI carry one name (the "front domain"), while the HTTP Host header, which is hidden from the censor by HTTPS encryption, carries another (the actual destination).

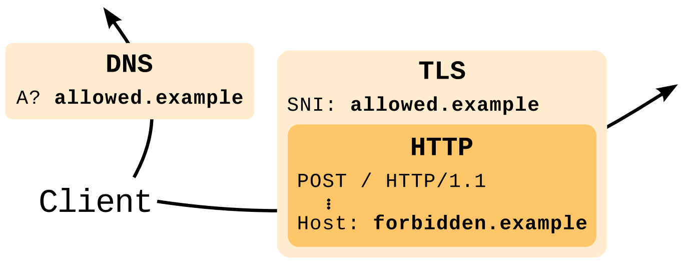


Fig. 1: Domain fronting uses different domain names at different layers of communication. At the plaintext layers visible to the censor—the DNS request and the TLS Server Name Indication—appears the front domain **allowed.example**. At the HTTP layer, unreadable to the censor but meaningful to the intermediate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

The censor may block neither the DNS request nor the SNI without collaterally blocking the front domain. The Host header is invisible to the censor, but visible to the server receiving the HTTPS request, which uses the header to internally route the request to its destination. No traffic ever reaches the front domain, which may be oblivious to the circumvention.

For those familiar with decoy routing [3], [4], [5], [6], domain fronting can be understood as "decoy routing at the application layer." A fuller comparison with decoy routing appears in Section II.

This Wget command shows domain fronting in action on Google's infrastructure. Here, an HTTPS request for www.google.com has a Host header for maps.google.com. The HTML in the response is that of Google Maps.

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

Domain fronting works with CDNs because a CDN's front-end 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 for CDNs to work, but this "origin pull" configuration is common.) Fronting works with Google because Google App Engine, a web application platform, can host a simple "reflector" application that emulates an origin-pull CDN, simply forwarding every request to another server. In both cases, the intermediate web service does not forward requests to just anywhere—it has to be to the domain of a customer. In order to run a fronting backend, one generally has to create an account with the web service and pay for bandwidth.

A variation on fronting is "domainless" fronting, in which there is no DNS request and no SNI. It looks to the censor like the user is browsing an HTTPS site by its IP address, or using a web client that doesn't support SNI. Domainless fronting can be useful when there is no known front domain with sufficiently high collateral damage. The censor is faced with the choice of blocking an entire IP address, not only a domain; blocking requests without SNI; or trying to find some other, presumably more expensive, test to distinguish circumvention traffic.

We have developed domain fronting into a full-fledged circumvention system called meek, implemented as pluggable transport [7] for Tor. meek combines fronting with an HTTP-based tunneling proxy: upstream data are transformed into a sequence of HTTP requests, which are fronted in order to reach a proxy server. The server decodes the requests and feeds their payloads into the Tor network. Downstream data received by the server from Tor are returned to the client within HTTP responses. meek's architecture appears in Figure 2. Section IV describes how it works in detail.

meek has been built into an experimental release of Tor Browser, and has a small number of users. Our deployment so far uses Google App Engine and Amazon CloudFront as intermediate web service backends. Section VI names other fronting-capable services and Section IX describes other possible deployments. In addition to the Tor pluggable transport, systems based on meek are now used by other circumvention systems including [blinded for anonymous submission].

Domain fronting and HTTPS take care of two big circumvention challenges, those of hiding the destination and of obscuring the contents of a message. There remain other, more subtle, attributes of communication that could serve as a basis for blocking. For example, the plaintext handshake is a fingerprintable part of TLS. Section V describes how we use a web browser extension as an instrument for making HTTPS requests, to make meek's TLS and HTTPS characteristics match those of a browser as closely as possible. Our goal is to force the censor to employ more sophisticated, expensive, and error-prone statistical tests to classify traffic as allowed or prohibited, rather than domain blacklists or deep packet inspection—to force the practice of censorship to exist in a world not of simple packet filters, but of probabilities and false positives. A preliminary analysis of meek's resistance to traffic analysis appears in Section VII. The analysis is based on comparing traffic traces of meek to samples of HTTPS traffic at a network router.

To our knowledge, the earliest application of domain fronting to circumvention was by GoAgent [8], a tool based on App Engine and once widely used in China. Fronting has also been used with success since 2013 for the rendezvous component of another transport called flash proxy [9]. GoAgent and flash proxy are the systems that most directly inspire ours.

II. Background and related work

Broadly speaking, there are two main challenges in circumvention: blocking by content and blocking by address. Blocking by content is based on *what you say*, and blocking by address is based on *whom you talk to*. 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, whatever the contents of the communication may be. A savvy censor will use both means of censorship, and effective circumvention requires countering both.

To the above challenges we add a related one, active probing. An active-probing censor does not merely passively observe traffic: it injects its own traffic, acting as if it were an ordinary client and connecting to suspected proxy servers to see how they respond. Then a connection is successful, the server can be blacklisted. Active probing is a precise means of identifying whether a server belongs to a circumvention system, even when a protocol is hard to detect on the wire. Winter and Lindskog [10] confirmed an earlier discovery of Wilde [11] that China's Great Firewall identifies Tor bridges by issuing scans that follow up on observed client connections, to discover whether a server speaks the Tor protocol. The discovery of active probing was the motivation for probing resistance in ScrambleSuit [12] and obfs4 [13], described in this section.

There are two general strategies for countering content-based blocking. The first is to make traffic look unlike anything forbidden by the censor; that is, fail to match a blacklist. The second is to resemble a protocol that is explicitly allowed by the censor; that is, match a whitelist. Following the first strategy are the so-called "look-like-nothing" transports whose payloads are indistinguishable from a uniformly random byte stream. Good examples of look-like-nothing transports are obfs2 [14] and its successor obfs3 [15], Tor's go-to pluggable transports since early 2012 [16]. Both obfs2 and obfs3 work by

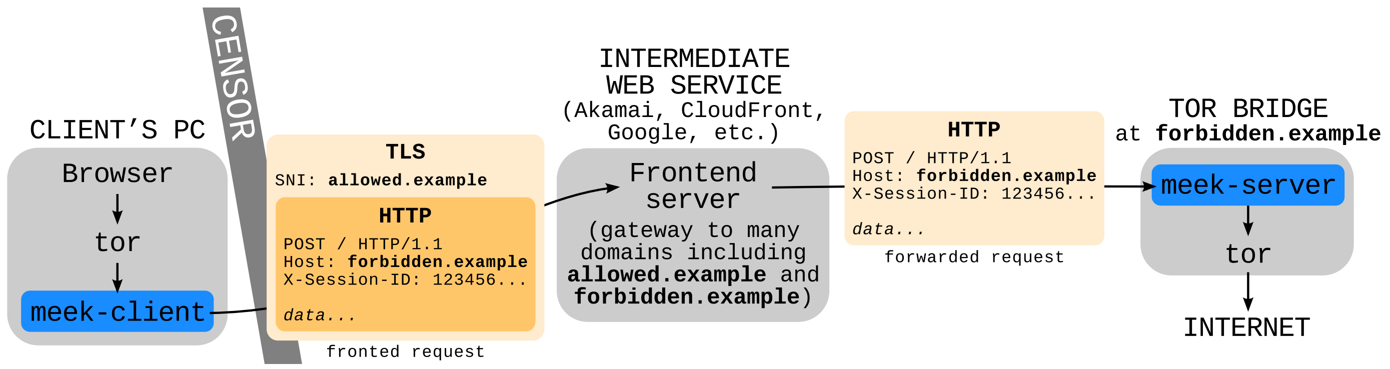


Fig. 2: Architecture of meek. The client sends an HTTP request to the Tor bridge by way of an intermediate web service such as a CDN. The client protects the bridge's domain name **forbidden.example** from the censor by fronting it with another name, here **allowed.example**. The intermediate web server 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 transport plugins, the interface between Tor and meek. The host actually at allowed.example does not participate in the communication.

superenciphering an underlying stream, using a key exchange that has no plaintext components. ScrambleSuit [12] is like obfs3 in its payload obfuscation, but additionally obscures its traffic signature (packet lengths and timing), and is designed to resist active probing. A ScrambleSuit server will accept a TCP connection but not send a reply until the client proves knowledge of a shared secret. obfs4 [13] (which despite the name has more in common with ScrambleSuit than with obfs2 and obfs3) builds on ScrambleSuit with more efficient cryptography and an authenticated handshake.

The other strategy against DPI is the steganographic one: look like something the censor does not block. 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 [18] transforms traffic to look like a cover protocol using a variety of special-purpose encoders. Code Talker Tunnel (formerly called SkypeMorph) [19] mimics a Skype video call. FreeWave [20] encodes a stream as an acoustic signal and sends it over VoIP to a proxy. Dust [21] uses encryption to hide static byte patterns and then shapes statistical features such as packet lengths and byte frequencies so that they match given distributions.

Houmansadr et al. [22] 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 VoIP calls.

The other grand challenge of proxy-based circumvention is address-based blocking. 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 new relays as secret "bridges" [24] whose addresses are not publicly known. BridgeDB, the database of secret bridges, carefully distributes bridge addresses so that anyone can learn a few bridges, but it is hard to learn all of them. BridgeDB uses captchas and other rate-limiting measures, and over short time periods, always returns the same set of bridges to the same requester, preventing enumeration by simple repeated queries. BridgeDB is also capable of distributing the addresses of obfuscated bridges (obfs3 or ScrambleSuit); the combination of careful bridge address distribution and obfuscated protocols gives Tor's system a realistic claim to addressing both main challenges of circumvention. Address distribution appears to be the weaker side of the defense, as evidenced by real-world censors' apparent preference for blocking bridge addresses over real-time DPI-based blocking [10].

CensorSpoofer [25] 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 [9] 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 in a sense the opposite of meek's: where flash proxy uses many cheap, disposable, individually blockable proxies, meek uses just a few high-value front domains on hard-to-block network infrastructure. A quirk of the browser proxy model 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 App Engine since 2013 [26]. 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 Tor protocol.

A technique known as OSS [27] (for "online scanning service") bounces messages through web services that are capable of making HTTP requests. For example, a censored client may ask an HTTPS-based online translation service to translate the web page at http://forbidden.example/*data...*, where "*data...*" is the URL-embedded message the client wants to send. The translation service requests the URL, unwittingly sending the message on behalf of the client. OSS shares certain similarities with domain fronting: an unblockable web service takes the place of the front domain, and the destination of the communication is embedded somewhere in an HTTPS query rather than in the Host header.

Decoy routing [3] 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 [4], Cirripede [5], and TapDance [6]. Decoy routing asks friendly ISPs to deploy special routers that lie 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 within HTTPS. Fronting has the advantage that it doesn't require knowing cooperation by network intermediaries.

Schuhard et al. [28] 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. [29] show that even though such alternate paths exist, they are many times more costly to the censor, especially when participating routers are placed strategically.

CloudTransport [30] uses cloud storage, 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 with HTTPS, and 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 inside HTTPS.

GoAgent [8] is a direct inspiration for meek in its use of domain fronting with App Engine. Traffic is fronted by the Google frontend server, using the "domainless" fronting model without SNI, and delivered to its destination by a specialized app running on App Engine. GoAgent doesn't use an additional general-purpose proxy after fronting; rather, HTTP and HTTPS URLs are fetched directly from the App Engine servers. Because of that, GoAgent is limited to carrying HTTP and HTTPS traffic only, and the contents of all communications are revealed to Google. Users of GoAgent generally upload their own personal copy of the proxy code to App Engine, which is free of charge to use as long as they do not exceed a bandwidth quota. According to a May 2013 survey [31], GoAgent was the circumvention tool most used in China, with 35% of survey respondents having used it in the previous month. This figure is higher than that of paid (29%) and free VPNs (18%), and far above that of other special-purpose tools like Tor (2.9%) and Psiphon (2.5%). Use of GoAgent in China was disrupted starting in the beginning of June 2014 when all Google services (including GoAgent's front domains) were blocked [32]. meek over App Engine was similarly affected.

Cloud-Entry [33] is somewhat similar to GoAgent. It also uses App Engine, but uses a socket API rather than a URL fetch API, in order to support protocols other than HTTP and HTTPS. Cloud-Entry has the same problem with NAT that flash proxy does, because the proxy tries to connect back to the client with a socket. App Engine kills web requests (and their associated socket connections) when they take longer than 60 seconds, so users must reconnect at least that often.

Psiphon [34] is a network of many one-hop proxies, with encryption and traffic obfuscation between client and server. Lantern [35] uses a network of proxies running on volunteer computers; proxy addresses are shared only with trusted friends in order to make them difficult to enumerate.

III. Threat model

The model includes four actors: the censor, the censored client, the intermediate web service, and the destination Tor bridge. Circumvention is achieved when the client evades the censor to reach the bridge, because the bridge can then proxy to anywhere. The client and bridge cooperate with each other. The intermediate web service does not need to cooperate with either, except to the extent that it doesn't collude with the censor.

The censor controls a network and the links into and within it. The censor is able to inspect traffic flowing across all links under its control and may block or allow any packet. The censor may 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 bridge lie outside. The censor blocks direct communication between the client and the bridge, 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 TLS certificate authority; specifically it cannot man-in-the-middle an HTTPS session without being caught by ordinary certificate verification. The client has a way to get a copy of the necessary client programs.

Section IX shows what happens when some of the threat model's assumptions are violated; specifically what happens when the intermediate web service's frontend server is inside the censor's network and when the censor controls a certificate authority.

IV. Architecture and protocol of meek

The components of the system are shown in Figure 2. The parts that are new in meek are the programs meek-client and meek-server, which are meek's interface with Tor.

meek-client 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 it up into a POST request and fronts the request through the web service, which in turn forwards it to meek-server on the bridge. If the response returned by meek-server contains any HTTP body, meek-client decodes the body and sends it back to the Tor client. meek-client takes pains to make its TLS fingerprint look like that of a browser; see Section V.

meek-server is a special-purpose web server. When meek-server receives a request, it extracts the body and feeds it into the Tor network. If there is anything pending to be sent back to the client from Tor, meek-server includes it in the body of the HTTP response. meek-server also handles session management: keeping track of which requests correspond to which active Tor connection.

When meek is used with Google App Engine, one other component is needed: a small "reflector" web app that simply copies any request it receives and forwards it to a remote instance of meek-server. App Engine cannot directly run a Tor bridge, so we bounce the requests to somewhere that can. The reflector app emulates the behavior of an origin-pull CDN, which also simply copies and forwards requests when the response is not already cached.

The main technical challenge of the protocol is splitting a TCP stream up into a sequence of HTTP requests and responses. The body of each HTTP request and HTTP response carries a small chunk of upstream or downstream communication. There must therefore be some way of making sure that chunks are correctly reassembled at the other end, in order and without gaps or duplicates. meek takes a naive approach: requests and responses are strictly serialized. The client doesn't send a second chunk of data (i.e, make another HTTPS request) until it has received the response to its first. The reconstructed stream is simply the concatenation of bodies in the order they arrive. Handling stream reconstruction this way is simple and guarantees correctness, but has performance drawbacks because there is a full round-trip between every send. Section IX proposes a more efficient encoding that would enable multiple requests in parallel.

meek-server must handle many clients at once, and therefore maintains many open connections to a local Tor process, one for each active client. When a new request arrives, the server must determine which open session it belongs to (or whether it is a brand-new client needing a new session). In TCP, different streams are distinguished by their (source IP, source port, dest IP, dest port) tuple, but that doesn't work for meek, for two reasons: the source IP of the requests is that of the intermediate web service, not of the actual client; and a logical stream is chopped into multiple independent HTTP requests, so the source port is always changing. Instead, each client generates a random 256-bit session ID for each new Tor stream, which it sends with its requests in a special X-Session-Id HTTP header. meek-server, when it sees a session ID for the first time, opens up a new connection to the local Tor process and associates the connection and ID in a cache. Subsequent requests with the same session ID will reuse the same Tor connection. Sessions are closed after a period of inactivity. Figure 3 shows a sample of the protocol.

```
POST / HTTP/1.1
Host: forbidden.example
X-Session-Id: cbIzfhx1Hn+RHURmIPhjgY+W3B6zA8Ua6dd92DLsc0E=
Content-Length: 517

\x16\x03\x01\x02\x00\x01\x00\x01\xfc\x03\x03\x9b\xa9\x9f...

    HTTP/1.1 200 OK
    Content-Length: 739

    \x16\x03\x03\x00\x3e\x02\x00\x00\x3a\x03\x03\x53\x75\xa2...

POST / HTTP/1.1
Host: forbidden.example
X-Session-Id: cbIzfhx1Hn+RHURmIPhjgY+W3B6zA8Ua6dd92DLsc0E=
Content-Length: 0


    HTTP/1.1 200 OK
    Content-Length: 75

    \x14\x03\x03\x00\x01\x01\x16\x03\x03\x00\x40\x06\x84\x25...
```

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\x02` is the beginning of a TLS ClientHello message). The second POST request is an empty polling request. The messages shown here is encrypted inside HTTPS until after it has been fronted, so the censor cannot use the Host and X-Session-Id headers for classification.

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 uses one of two techniques. The first is polling: every so often the client sends a request with an empty payload if it has nothing else to send, simply to give the server an opportunity to send something back. The polling interval starts short (100 ms) and grows exponentially up to a maximum of 5 s. The other technique uses streaming HTTP responses over persistent HTTP connections. When the server sends a response, it keeps the connection open, continually appending data to the response body while it has anything to send. When the server receives a new request from the same client, it closes any response stream it has open to that client and starts a new one. This second technique is especially beneficial for bulk downloads, but doesn't work on all services: notably Google App Engine doesn't support streaming of this kind.

As an encrypted tunneling protocol, meek necessarily adds overhead. Each chunk of data gets an HTTP header, and then the HTTP request is wrapped in TLS. We estimate the overhead when meek is used to transport Tor. Tor uses fixed-size cells of 514 bytes [36, Section 0.2]. Each cell is wrapped in a TLS Application Data record, which adds 29 bytes when using a GCM ciphersuite (the default in current versions of Tor). This 543-byte record is received by meek-client, which adds an HTTP header whose length is about 400 bytes (varying slightly depending on the length of the Host field). The HTTP request goes into its own TLS Application Data record when it becomes HTTPS, adding about 50 bytes depending on how much padding is used by the ciphersuite in use.

514	Tor cell
+29	Tor TLS record
≈+400	meek HTTP header
≈ +50	meek TLS record
≈ 1000	total

The total overhead varies according to the length of the Host field in the header and the specifics of the intermediate web service's TLS configuration, but is somewhere in the neighborhood of $(543+450)/543-1 \approx 83\%$. This is the worst case, when Tor sends a single cell. Tor can bunch multiple cells into one transmission; if two cells are sent at once, the overhead drops to ≈41%; if three, it is ≈28%. Bulk uploads and downloads tend to have less overhead, bunching many cells at once. Streaming HTTP responses, when they are possible, reduce overhead further, because only one HTTP header is needed for a large number of transmissions. Empty polling requests also use bandwidth, but they are sent only when the connection is idle, so they don't affect upload or download speed. As is discussed in Section VII, meek-client reuses the same TLS connection for many requests, so the fixed overhead of the initial TLS handshake is amortized.

There is additional latency overhead caused by meek's extra network hop: rather than accessing the Tor bridge directly, the client first goes to the intermediate web service. How much latency the extra hop adds depends, of course, on the network location of the client, web service, and Tor bridge. Strict serialization of requests and responses becomes a problem with large round-trip times, as throughput becomes limited by latency. Section IX describes ways to improve performance with high latencies.

V. Camouflage for the TLS layer

Domain fronting defeats address-based blocking and active probing. HTTPS foils simple DPI, hiding any byte patterns in the underlying stream. The censor may also apply DPI to the tunnel itself in order to try to distinguish fronted from non-fronted traffic. This section is about TLS fingerprinting, which is the censor's remaining easy distinguisher, and how we use a web browser with an unremarkable TLS fingerprint to disguise the HTTPS tunnel. Section VII is about more subtle and hard-to-apply forms of traffic analysis.

The TLS handshake is largely plaintext [37, Section 7.4], and leaves enough room for variation, for example in the client's lists of ciphersuites and extensions, that it is possible to fingerprint different client TLS implementations [38] based on their handshake. (We are only concerned with fingerprinting the client, because the server's TLS comes from the intermediate web service; i.e., it is what you would expect to see when connecting to the front domain.) Tor itself was blocked by China in 2011 because of the distinctive ciphersuite list it used at that time [39]. Figure 4 shows examples of how TLS client implementations may detectably differ. meek-client is written in the Go programming language, which has a custom TLS library [40]; Figure 4a shows how it would appear on the network with no TLS camouflage. Figures 4b and 4c, shows two browsers which have TLS fingerprints different from Go's and different from each other's. A censor could block an uncamouflaged meek using Go's TLS fingerprint, if it considered the collateral damage resulting from blocking other Go programs to be slight enough.

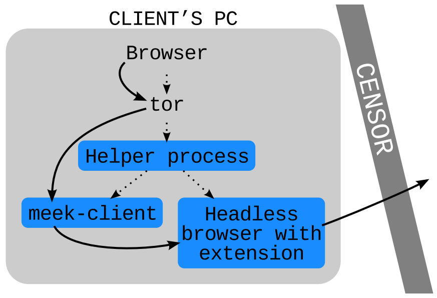


Fig. 5: Client architecture including a browser extension for TLS camouflage. Dashed lines show the process hierarchy. Solid lines show the flow of outgoing communication. The headless browser runs in the background and is invisible to the user. This figure is a zoomed-in view of the "Client's PC" component of Figure 2.

In order to disguise its TLS fingerprint, meek-client proxies all its HTTPS requests through an actual web browser. We implemented extensions for two web browsers, Firefox and Chrome, that enable them to make HTTPS requests on behalf of meek-client. The TLS fingerprint looks like that of a browser, because it is that of a browser. The browser instance running the extension is completely separate from the browser the user interacts with. It runs invisibly in the background in a separate process, doesn't display a user interface, and shares no state with the user's browser.

Figure 5 shows the interaction of components on the client's computer. The local Tor process runs a helper process that starts both meek-client and the headless browser, then configures meek-client to proxy its requests through the browser extension. The user's browser proxies through Tor; Tor proxies through meek-client; and meek-client proxies through the headless browser. The headless browser is the only part that ever actually touches the network. It should be emphasized that the URLs requested by the headless browser extension have no relation to the URLs the user browses; the extension only ever makes requests to the front domain.

The use of a browser extension brings benefits beyond TLS camouflage. It means meek's HTTP tunnel inherits other characteristics of the browser: for instance how often it resolves DNS, its connection reuse, and its HTTP keepalive behavior. The Firefox extension has special advantages when meek is used with Tor Browser because Tor Browser is a modified version of Firefox: we can use the same browser executable for both the browser the user interacts with (Tor Browser) and the one providing TLS camouflage (headless Firefox), saving

(a) Go 1.2’s crypto/tls library

```
Ciphersuites:
  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:
  server_name
  status_request
  elliptic_curves
  ec_point_formats
  signature_algorithms
```

(b) Firefox 24

```
Ciphersuites:
  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_3DES_EDE_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_DSS_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_DSS_WITH_CAMELLIA_256_CBC_SHA
  TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
  TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
  TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
  TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
  TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
  TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
  TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA
  TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA
  TLS_ECDH_ECDSA_WITH_RC4_128_SHA
  TLS_ECDH_RSA_WITH_RC4_128_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_SEED_CBC_SHA
  SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA
  TLS_RSA_WITH_3DES_EDE_CBC_SHA
  TLS_RSA_WITH_RC4_128_SHA
  TLS_RSA_WITH_RC4_128_MD5
Extensions:
  server_name
  renegotiation_info
  elliptic_curves
  ec_point_formats
  SessionTicket TLS
  next_protocol_negotiation
```

(c) Chrome 33

```
Ciphersuites:
  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_CHACHA20_POLY1305_SHA256
  TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
  TLS_RSA_WITH_AES_128_GCM_SHA256
  TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
  TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
  TLS_DHE_RSA_WITH_AES_256_CBC_SHA
  TLS_RSA_WITH_AES_256_CBC_SHA
  TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
  TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
  TLS_ECDHE_RSA_WITH_RC4_128_SHA
  TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  TLS_DHE_RSA_WITH_AES_128_CBC_SHA
  TLS_DHE_DSS_WITH_AES_128_CBC_SHA
  TLS_RSA_WITH_RC4_128_SHA
  TLS_RSA_WITH_RC4_128_MD5
  TLS_RSA_WITH_AES_128_CBC_SHA
  TLS_RSA_WITH_3DES_EDE_CBC_SHA
Extensions:
  server_name
  renegotiation_info
  elliptic_curves
  ec_point_formats
  SessionTicket TLS
  next_protocol_negotiation
  application_layer_protocol_negotiation
  Channel ID
  status_request
  signature_algorithms
  Signed Certificate Timestamp
```

Fig. 4: Selected differences in ClientHello messages in three different TLS implementations.

space in distribution packages and sparing the user the chore of having to separately configure a browser to run the extension. In Tor Browser, the headless Firefox is started and stopped transparently when meek is activated and deactivated.

No TLS camouflage is needed between the intermediate web service and meek-server, because that portion of the communication flow is not observed by the censor.

Appropriate TLS camouflage may differ depending on circumstances. On a traditional desktop PC, using a browser is probably the right choice. When [blinded circumvention tool] ported meek-client to Android, they didn’t use a browser as camouflage, but rather made all HTTPS requests with the standard Android HTTP API. This decision makes sense because other Android applications also use the standard API and share the same TLS fingerprint.

VI. A SURVEY OF FRONTING-CAPABLE WEB SERVICES

This section contains a brief survey of fronting-capable web services we have found, each with its pros and cons. The list is not exhaustive and is meant to illustrate the broad support for fronting that exists. Of the services listed, we have so far deployed meek only on Google App Engine and Amazon CloudFront. For the others, we have only checked that fronting works. Recall that even web services that support fronting will front only for their own customers, so new each deployment typically requires an investment of time and money.

Google App Engine [41] is a web application platform. Each application gets a user-specified subdomain of appspot.com, which can be fronted through almost any Google domain, including google.com, gmail.com, googleapis.com, and many others. App Engine can run only web applications, not a full Tor bridge; for that reason we use a tiny “reflector” application that merely copies incoming requests to a Tor bridge hosted elsewhere. Fronting through App Engine is attractive in the case where appspot.com is blocked—not unlikely, because of the ease of building a proxy there—but at least one other Google service is reachable. Applications that use less than one gigabyte of bandwidth daily are free of charge, making possible an upload-your-own-app model à la GoAgent.

Amazon CloudFront [42], not to be confused with CloudFlare, is the CDN of Amazon Web Services. A CloudFront “distribution” associates an automatically generated subdomain of cloudfront.net with an origin server, which in our case is an instance of meek-server. The front domain may be any other of CloudFront’s automatically generated subdomains, all of which support HTTPS through a wildcard certificate. CloudFront has a usage tier that is free of charge for a year, subject to a bandwidth limit of 50 gigabytes per month.

CloudFlare [43], not to be confused with CloudFront, is a CDN also marketed as protection against denial-of-service attacks. It appears that any domain name on CloudFlare can front for any other, even custom user-selected domains; however it is not obvious which one is best to use as a front. The front domain needs to be a popular one (causing high collateral damage if blocked), or the censor may simply block it entirely. It is doubly bad when a domain is blocked in this way, because not only is circumvention foiled, but some unsuspecting third party has just had their web site blocked. A solution may be to round-robin through a pool of popular domain names; or to use “domainless” fronting that doesn’t send SNI.

Akamai [44] is a large CDN. Requests may be fronted through the special HTTPS domain a248.e.akamai.net, which serves a certificate also good for *.akamaihd.net. The collateral damage of blocking Akamai is enormous: in 2010 it carried 15–20% of all web traffic [45]. Akamai costs more than other CDNs; it would likely require institutional support to keep a circumvention system running on it in the long term.

Microsoft Azure [46] is a cloud computing platform that features a CDN. Similarly to CloudFront, users get automatically generated subdomains of vo.msecnd.net, all of which can front for each other. There are other front domain names, like ajax.aspnetcdn.com, that may offer high collateral damage.

Fastly [47] is a CDN. It is the only one of the services we tested in that that appears to enforce SNI: if the SNI and Host do not match, the Fastly edge server returns an HTTP 400 ("Bad Request") error. However, if the TLS ClientHello omits SNI, then the Host may be any Fastly domain. It is the only service we have found that requires the "domainless" fronting style.

VII. TRAFFIC ANALYSIS

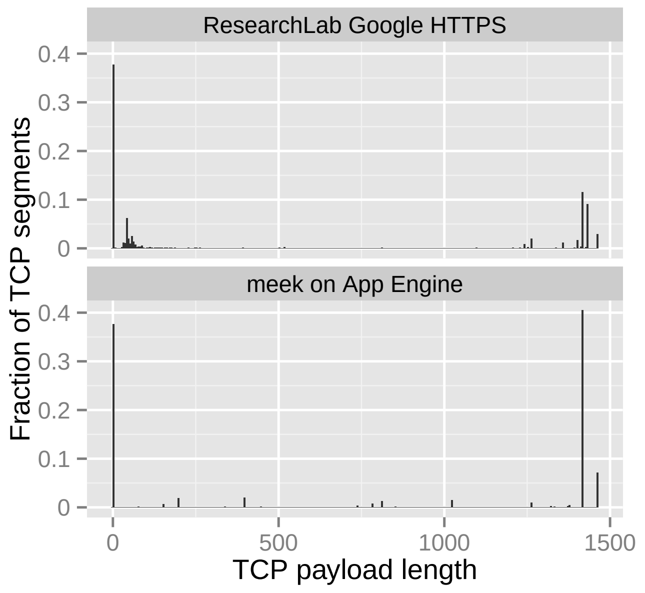
One of our goals in developing meek is to force the censor to use relatively more expensive statistical classifiers to effect blocking—generally, to increase the cost of censorship. We believe meek is resistant to the "easy" forms of blocking, namely IP address and DNS blocking, simple pattern-based DPI, and active probing. We hope to force the censor to resort to measuring more subtle features of the communication. In this section we provide a preliminary analysis of meek's resistance to such attacks. Specifically, we consider three traffic features: packet length distribution, number of simultaneous connections, and connection lifetime.

The issue of statistical traffic analysis is a general one [48], and mostly orthogonal to meek's main idea, domain fronting. It may be that the problem of traffic analysis is best solved by a modular component not necessarily specifically designed for meek.

As meek is based on HTTPS, we compare traffic traces of meek against traces of "ordinary" HTTPS traffic. We did a packet capture of automatically browsing the home pages of the top 500 Alexa web sites using meek in Tor Browser, with the browser camouflage described in Section V, and Google App Engine as the intermediate web service. The meek trace is 687 megabytes in all and covers about 4.5 hours of network activity. Our representative of ordinary HTTPS traffic is a sanitized 10-minute trace from a research lab [blinded as ResearchLab for anonymous submission], comprising data to and from TCP port 443 on any Google server. The ResearchLab trace consists 313 megabytes of packet headers, not payloads, destined for TCP port 443 on any Google server. The original IP addresses were masked by being replaced by a counter before we received the trace.

A. Packet lengths

Protocols vary in their packet lengths; a censor may block a circumvention protocol if its length distribution is distinct enough. Figure 6 compares the packet length distributions of the sample traffic traces. In both cases, about 38% of packets have an empty payload, mostly ACKs. There is a small peak in both traces at the usual TCP Maximum Segment Size of 1460 bytes. The concentration of values near 1418 bytes may be caused by Google servers' sizing of TLS records so they fit within TCP segments. Conspicuous in the meek trace are a small number of discrete frequent packet sizes, and the lack of a cluster of short payloads of around 50 bytes. Both of these characteristics are probably reflections of Tor's fixed cell size and the nearly fixed-size HTTP headers added by meek-client and meek-server.



ResearchLab 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. 6: Comparison of TCP payload length distributions between ordinary HTTPS connections to Google services from the ResearchLab traffic trace, and meek running on App Engine, fronted through www.google.com.

B. Connection lifetime

We consider the duration of TCP connections as a potential distinguisher. Figure 7 shows the cumulative probability of connection durations in our two traces. Considering the ResearchLab trace first, we see interesting probability concentrations on round numbers: 10 s, 60 s, 120 s, 180 s, and 240 s. We hypothesize that this phenomenon is caused by keepalive timeouts in web browsers and servers, and polling behavior of web apps. The small rise at 600 seconds is an artifact caused by the fact that the trace is only 10 minutes long. Connections lasting longer than 10 minutes are not accurately reflected, however we can say that only about 8% of connections lasted longer than 10 minutes.

The meek trace shows a propensity for longer connections.

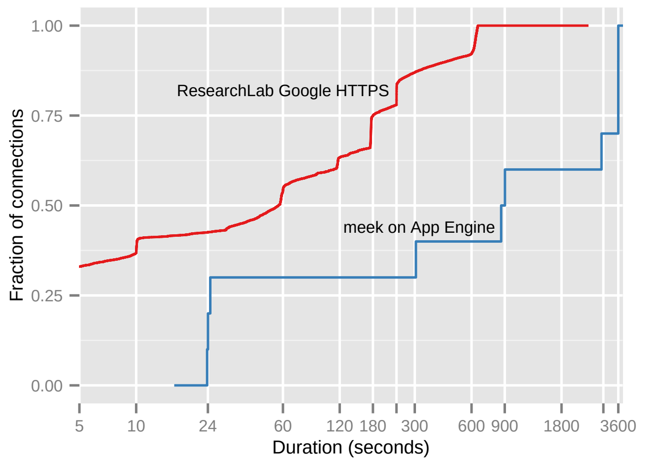


Fig. 7: CDF of connection lifetime. The horizontal axis is logarithmic. meek's connections tend to last longer than those of ordinary HTTPS traffic.

Over 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 use of long-lived HTTP keepalive connections, combined with meek-client's polling requests, which keep the connection from becoming idle. The traffic trace represents the browsing of hundreds of web sites and thousands of HTTPS transactions, but they all occurred over just a few TCP/TLS connections. 60% of meek connections lasted five minutes or longer, while only 13% of those of ordinary traffic did. meek has essentially no connections lasting less than 24 seconds, but such short connections account for over 42% of the ResearchLab trace. 30% (3 out of 10) of meek's connections lasted almost exactly one hour, evidently reflecting a built-in time limit in either the client browser extension or in App Engine.

In light of these findings, the censor may decide to block any long-lived HTTPS connections between a client and a web service. According to our traffic trace, doing so will not disrupt more than 8% of ordinary connections; whether that is an acceptable cost depends on the time threshold and the censor's tolerance for collateral damage. The censor can lower the timing threshold, at the cost of more false positives. Duration-based blocking is favorable for the circumventor, because if the censor terminates connections after, say, 10 minutes, in any case that's 10 minutes of circumvention achieved, after which meek can start a new connection and allow Tor to rebuild its circuits. Future enhancements in meek may even allow reconnecting using the same session ID, so there will be no effect on the client other than a brief delay while a new HTTPS connection is established.

C. Concurrent connections

The number of concurrent connections is another potential distinguisher. We have seen how meek tends to use one long-lived connection through which it tunnels all its requests. Figure 8 compares the number of concurrent HTTPS connections to Google per client in both traces. The number of connections

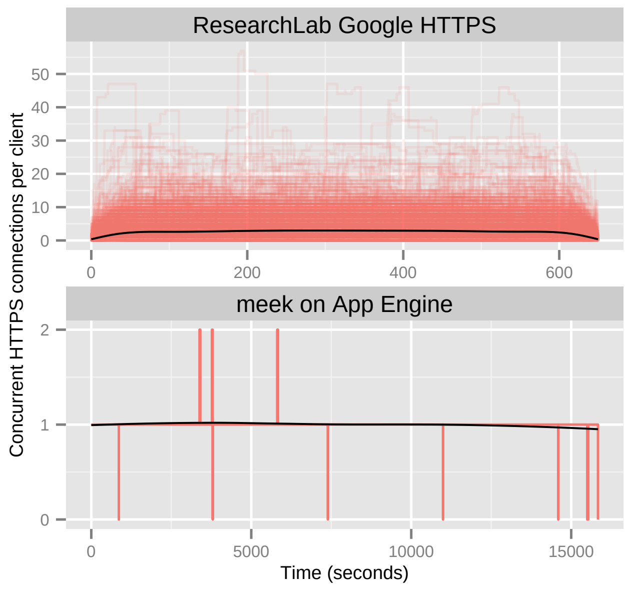


Fig. 8: Number of simultaneous HTTPS connections to any Google server. Each pale line indicates the number of HTTPS connections of a single client IP address. The ResearchLab trace shows 2,745 unique clients and the meek trace only one. The dark line is a smoothed average number of connections per client. The meek count stays close to 1 throughout, only momentarily jumping to 0 or 2. The average number of connections per user in the ResearchLab trace is 2.61, and in the meek trace is 1.002.

in the ResearchLab trace is variable, with some hosts briefly spiking up above 50 connections, but the average case is very heavily skewed to smaller numbers. The average number of connections in the ResearchLab trace is 2.61, with 28% of clients never having more than 2 concurrent connections. In the meek trace the average is almost exactly 1, only briefly deviating when one connection ends and another begins. We conclude that having a small number of concurrent connections is unremarkable.

♢

So far we do not find any obvious traffic characteristics that can be useful to reliably distinguish meek from other HTTPS traffic. The long-lived connections and few common packet sizes are potential targets for a more concerted attack. The fundamental problem we face is essentially that of steganography, the need to make our traffic fit some model of "normal" traffic. The problem is inherently difficult, because it is difficult to say just what normal traffic *is*. It depends on the network environment; it also depends on the endpoint. Google's traffic is different than Akamai's.

For the purposes of censorship circumvention, one doesn't need to match exactly a "typical" traffic profile; only not to deviate from it too much. Circumvention traffic must only be difficult enough to distinguish from ordinary traffic that blocking it requires an amount of false positives unacceptable

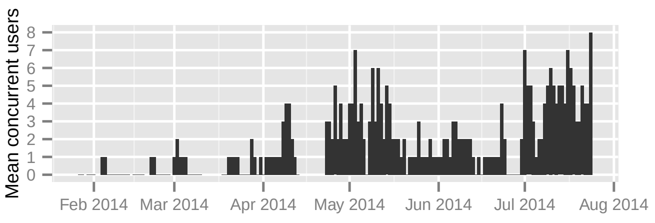


Fig. 9: Concurrent users of the meek pluggable transport for Tor.

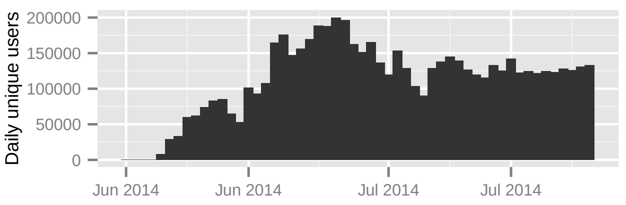


Fig. 10: Daily unique users of meek with [blinded circumvention tool] (note vertical axis). The decrease at the end of June was caused by an unblocking event that made meek unnecessary for some users.

to the censor. Even 1% of Google traffic accidentally blocked may exceed the collateral damage tolerance of some censors. Further, it is hard to characterize what normal use is of, say, a CDN: they provide access to ordinary web sites, but also to video streams and software updates, and not all clients are web browsers. The difficulty we face is symmetric; if finding the right model is difficult task for us, it is also difficult for the censor. We have seen that censors prefer to use easy classifiers when they exist. By removing the easy ways, we increase the censor's cost.

VIII. Deployment

Development on meek as a Tor pluggable transport [7] began in January 2014. Most users who use Tor use it with Tor Browser [49], a customized version of Firefox that is preconfigured to use a built-in Tor client. Tor Browser has an easy interface for selecting pluggable transports. We made a series of experimental releases of Tor Browser featuring meek, releasing the first one on February 18. [Some details of deployment are omitted for anonymous submission.]

Figure 9 shows the average number of concurrent users of meek with Tor since close to the beginning of development. A measurement of 5, for example, means that there were on average five simultaneous users of the system at any time during the day. The measurements come from the Tor Metrics Portal [50], which aggregates statistics reported by Tor relays. The number of users is estimated by counting the number of Tor directory requests made by clients [51].

[Blinded circumvention tool] began their independent deployment of meek in June 2014. Figure 10 shows the number of unique daily users of meek, estimated through clients' self-reporting of their most recent connection date. The developers tell us that the drop that occurs in the middle of June was not caused by a blocking event, but rather an unblocking event that made meek unnecessary for some users. Although Figures 9 and 10 are not directly comparable because they count slightly different things, it is clear that meek-with-[blinded circumvention tool], with its thousands of daily users, is much more widely used than meek-with-Tor, which has not yet seen a formal release.

Our experimental Tor Browser releases have all been based on Google with App Engine. Support for CloudFront was not realized until late July. We are running the backends on paid accounts of these services for use by the public. The cost for bandwidth on App Engine is $0.12 per gigabyte, with one gigabyte free each day. The total App Engine bill so far has been $2.07, with most days' usage being below the billable threshold. There has not yet been enough use of CloudFront to cost anything.

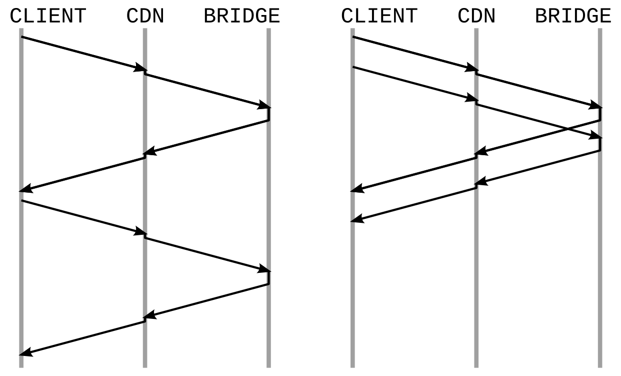
A longer-term goal is to find a way to fund continued operation if meek becomes popular. Although some of the systems listed in Section VI are inexpensive enough that some users could upload their own fronting backend, thereby diffusing costs, we believe that an organization operating a backend for the use of the public has compelling usability advantages. With client software configured to use the public backend by default, there is nothing for the user to upload and no out-of-band communication is needed before getting started—when everything works it offers circumvention without configuration.

IX. Discussion

Domain fronting derives part of its unblockability from the high collateral damage of blocking the fronted-through domain. Fronting through www.google.com to App Engine, for example, is making a bet that the censor will lack the temerity to block Google. The Great Firewall of China called our bluff in June of 2014, when it blocked access to all Google services [32]. In 2012, the government of Pakistan blocked all of YouTube in order to restrict access to one video, and in 2011, the government of Egypt briefly "turned off the net" completely. These examples show that blocking resistance is not an absolute, black-and-white security property; it can only be discussed in an economic context, relative to the costs borne by the censor and by users of a system. To be sure, blocking Google is an extreme measure, probably out of the reach of most censors. Blocking all of the fronting-capable services listed in Section VI would be even harder.

A censor could attack domain fronting by going directly to the operators of the intermediate web service and asking them either to break fronting (i.e., require SNI to be present and SNI and Host header to agree), or to get rid of their customers who are engaged in enabling circumvention (i.e., us). The censor could threaten to block the web service entirely otherwise. Whether the attack will succeed depends on the censor's and the web service's specific costs and motivations, so we cannot make any general statements. It is ultimately a question of who needs whom more: whether it hurts the web service or the censor more when the service is blocked.

Domain fronting shares a potential weakness with decoy routing: the overt and covert destinations lie at the ends of different network paths. The difference in paths may create



(a) Sequential (b) Parallel

Fig. 11: The nature of potential performance gain from sending fronted requests in parallel.

side channels—latency measurements for instance—that enable the censor to distinguish fronted traffic from traffic that is truly destined to the 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. Fronted traffic, on the other hand, always continues all the way to the origin server after reaching the CDN (as if nothing were ever cached), resulting in a higher latency. Latency measurement was applied to decoy routing by Schuhard et al. [28, Section 5], who compared empirical round-trip-time distributions using a Kolmogorov–Smirnov test.

The intermediate web service enjoys a privileged network position, from which it is able to monitor all circumvention traffic. Even if the censor doesn't know who the users are, the CDN knows. The web service has basically the same network visibility as the client's ISP would have, if the client were not circumventing. For this reason, domain fronting should be combined with another layer, for example Tor, that prevents the web service from observing the contents and destination of messages. There is additional risk when the client is browsing a web site controlled by the same service as the one they are fronting through, for example browsing YouTube while fronting through www.google.com. Even with Tor, when this happens the web service gets to see both entry and exit traffic, and is in a position to perform traffic correlation attacks. The problem seems hard to counter, because the front domain needs to be popular in order to have high collateral damage, but popular domains are also the ones that people tend to want to go to. A possible mitigation is padding that lasts until the second hop of a three-hop Tor circuit, which would reduce the correlation between entry and exit traffic.

We have assumed that the intermediate web service is outside the censor's domain of control, so that communication between the web service and Tor bridge doesn't have to be protected. What happens when the web service—a CDN edge server, for instance—is inside the network controlled by the censor? If the edge server forwards the request to the bridge as an ordinary HTTP request on the public Internet, then blocking it is as easy as blocking the IP address of the bridge. On the other hand, if the web service encapsulates the request and sends it encrypted, or over its own fast private network, to another server outside of the censor's control, then fronting will still work.

The "domainless" fronting style, which omits SNI, is useful when there is no good front domain known for a web service. At least one CDN, Fastly, requires the domainless style. All current web browsers support SNI, so it is possible that lack of SNI could be used as a distinguisher. According to our communication with a representative of [a blinded organization observing many TLS connections], 83.5% of connections used SNI in June 2014. There are apparently enough connections without SNI that a censor will at least have to consider seriously the costs and benefits before deciding to block SNI-less TLS outright.

The current implementation of meek strictly serializes its HTTPS requests and responses, in order to keep chunks of the underlying TCP stream in the proper order. The unfortunate side effect of this simple approach is added latency that gets worse with longer round-trip times. In future work we plan to add an additional reliability layer, perhaps like the system of sequence numbers and acknowledgments used in OSS [27], that would allow requests to be pipelined and cope with out-of-order delivery. Figure 11a shows how the current implementation works, and Figure 11b shows the nature of the performance improvement that could be gained with a reliability layer and parallel requests.

meek's use of HTTPS is subject to the usual public key infrastructure and trust model. The browser running the HTTPS camouflage extension (Section V) has a list of trusted certificate authorities. A censor that controls a certificate authority trusted by the browser is able to perform a man-in-the-middle attack between the client and the intermediate web service, and detect fronting by decrypting TLS and checking for specific Host headers. Such a censor would be able to detect circumvention, but still not read the client's plaintext because of the additional encryption and authentication of the tunneled Tor layer. It seems risky for the censor to risk being detected in a man-in-the-middle attack merely to confirm that a user is circumventing, but it is not impossible for a sufficiently powerful censor. The problem doesn't exist when the front domain is one that happens have its public key "pinned" in the headless browser, for instance when using the Chrome extension and fronting through www.google.com.

The key idea of meek is domain fronting, but its HTTP tunneling ability is useful on its own. Another idea for deployment is to use thousands of simple HTTP reflectors on cheap web hosting (in place of domain fronting on a big CDN). The request-reflecting role of a CDN can be played by a PHP script, for example, or a special web server configuration. Their URLs unprotected by domain fronting, each reflector would be easily blockable individually, so there would need to be sufficiently many that a censor could not block them all. This deployment model is essentially the same as with ordinary Tor bridges: have lots of volunteer-operated bridges and make it hard to enumerate them all. The advantage that an HTTP transport would have is that it is easier for a volunteer to upload a PHP file to a web host than to set up and maintain a Tor bridge.

X. Summary

We have presented domain fronting, an HTTPS technique that uses different domain names at different communication layers in order to hide the remote endpoint of a communication. We implemented meek, a pluggable transport for Tor. We show that it resists common methods of blocking, and begin an investigation into its resistance against more subtle and expensive means of traffic analysis. We discuss the results of an initial deployment.

[Links to source code omitted for anonymous submission.]

References

[1] D. Eastlake, "RFC 6066: Transport Layer Security (TLS) extensions: Extension definitions," Internet Engineering Task Force, Jan. 2011, https://tools.ietf.org/html/rfc6066.

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

[3] 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, http://www.usenix.org/events/foci11/tech/final_files/Karlin.pdf.

[4] 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, http://www.usenix.org/events/sec/tech/full_papers/Wustrow.pdf.

[5] 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.

[6] "TapDance: End-to-middle anticensorship without flow blocking," in *Proceedings of the 23rd USENIX Security Symposium*. San Diego, CA: USENIX Association, Aug. 2014, https://jhalderm.com/pub/papers/tapdance-sec14.pdf.

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

[8] "GoAgent," https://github.com/goagent/goagent.

[9] 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, Jul. 2012, https://crypto.stanford.edu/flashproxy/flashproxy.pdf.

[10] 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.

[11] T. Wilde, "Great Firewall Tor probing circa 09 DEC 2011," Team Cymru, Tech. Rep., Jan. 2012, https://gist.github.com/da3c7a9af01d74cd7de7.

[12] 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.

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

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

[15] ——, "obfs3 (the threebfuscator)," Jan. 2013, https://gitweb.torproject.org/pluggable-transports/obfsproxy.git/blob/HEAD:/doc/obfs3/obfs3-protocol-spec.txt.

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

[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] 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.

[19] 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.

[20] 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.

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

[22] 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.

[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] R. Dingledine and N. Mathewson, "Design of a blocking-resistant anonymity system," Tor Project, Tech. Rep. 2006-1, Nov. 2006, https://svn.torproject.org/svn/projects/design-paper/blocking.pdf.

[25] 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.

[26] Tor Project, "#8860: Registration over App Engine," May 2013, https://trac.torproject.org/projects/tor/ticket/8860.

[27] D. Fifield, G. Nakibly, and D. Boneh, "OSS: Using online scanning services for censorship circumvention," in *Proceedings of the 13th Privacy Enhancing Technologies Symposium (PETS)*, Jul. 2013, https://www.bamsoftware.com/papers/oss.pdf.

[28] 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.

[29] 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.

[30] 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)*, Jul. 2014, http://www.cs.utexas.edu/~amir/papers/CloudTransport.pdf.

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

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

[33] Z. Yuan, X. Chen, Y. Xue, and Y. Dong, "Cloud-entry: Elusive Tor entry

points in cloud," May 2014, http://www.ieee-security.org/TC/SP2014/posters/YUANZ.pdf https://github.com/zlyuan/GTor.

[34] "Psiphon 3 circumvention system README," https://bitbucket.org/psiphon/psiphon-circumvention-system.

[35] "Lantern," https://getlantern.org/.

[36] R. Dingledine and N. Mathewson, "Tor protocol specification," Jun. 2014, https://gitweb.torproject.org/torspec.git/blob/HEAD:/tor-spec.txt.

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

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

[39] Tor Project, "#4744: GFW probes based on Tor's SSL cipher list," Dec. 2011, https://trac.torproject.org/projects/tor/ticket/4744.

[40] "Package tls," https://golang.org/pkg/crypto/tls/.

[41] "Google App Engine," https://developers.google.com/appengine/.

[42] "Amazon CloudFront," https://aws.amazon.com/cloudfront/.

[43] "CloudFlare," https://www.cloudflare.com/.

[44] "Akamai," http://www.akamai.com/.

[45] E. Nygren, R. K. Sitaraman, and J. Sun, "The Akamai network: A platform for high-performance Internet applications," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 2–19, 2010, http://www.akamai.com/dl/technical_publications/network_overview_osr.pdf.

[46] "Microsoft Azure," http://azure.microsoft.com/.

[47] "Fastly," http://www.fastly.com/.

[48] 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.

[49] M. Perry, E. Clark, and S. Murdoch, "The design and implementation of the Tor Browser," Tech. Rep., Mar. 2013. [Online]. Available: https://www.torproject.org/projects/torbrowser/design/

[50] Tor Project, "Bridge users using transport meek," May 2014, https://metrics.torproject.org/users.html?graph=userstats-bridge-transport&transport=meek#userstats-bridge-transport.

[51] K. Loesing, "Counting daily bridge users," Tech. Rep. 2012-10-001, Oct. 2012, https://research.torproject.org/techreports/counting-daily-bridge-users-2012-10-24.pdf.