StegoTorus: A Camouflage Proxy for the Tor Anonymity System

Zachary Weinberg,[1,2] Jeffrey Wang,[3] Vinod Yegneswaran,[2] Linda Briesemeister,[2]
Steven Cheung,[2] Frank Wang,[3] and Dan Boneh[3]

[1]Carnegie Mellon University [2]SRI International [3]Stanford University

ABSTRACT

Internet censorship by governments is an increasingly common practice worldwide. Internet users and censors are locked in an arms race: as users find ways to evade censorship schemes, the censors develop countermeasures for the evasion tactics. One of the most popular and effective circumvention tools, Tor, must regularly adjust its network traffic signature to remain usable.

We present StegoTorus, a tool that comprehensively disguises Tor from protocol analysis. To foil analysis of packet contents, Tor's traffic is steganographed to resemble an innocuous cover protocol, such as HTTP. To foil analysis at the transport level, the Tor circuit is distributed over many shorter-lived connections with per-packet characteristics that mimic cover-protocol traffic. Our evaluation demonstrates that StegoTorus improves the resilience of Tor to fingerprinting attacks and delivers usable performance.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: Security and protection; K.4.1 [**Public Policy Issues**]: Transborder data flow

General Terms: Algorithms, Design, Security

Keywords: Anticensorship, Circumvention Tools, Cryptosystems, Steganography

1. INTRODUCTION

Freedom of speech and decentralization are bedrock principles of the modern Internet. John Gilmore famously said that "the Net interprets censorship as damage, and routes around it" [31]. It is more difficult for a central authority to control what is published on the Internet than on older, broadcast-based media; in 2011, the Internet's utility to the "Arab Spring" revolutions prompted a spokesman for the US Department of State to label it "the Che Guevara of the 21st century" [63]. Nonetheless, national governments can easily inspect, manipulate, and block nearly all network traffic that crosses their borders. Over a third of all nations impose "filters" on their citizens' view of the Internet [16]. As the Internet continues to grow in scope and importance, we can expect that governments will only increase their efforts to control it [13].

Tools for evading online censorship are nearly as old as the censorship itself [33, 56]. At present, one of the most effective circumvention tools is "Tor" [24]. Tor provides anonymity for its users by interposing three relays between each user and the sites that the user visits. Each relay can decrypt just enough of each packet to learn the next hop. No observer at any single point in the network, not even a malicious relay, can know both the source and the destination of Tor traffic.

Although Tor was not designed as an anticensorship tool, it works well in that role. Repressive governments respond by blocking Tor itself. In 2010 and 2011, Iran attempted to block Tor traffic by scanning TLS handshakes for Diffie-Hellman parameters and/or certificate features that were characteristic of Tor [20, 48]. China employed a similar technique but enhanced it with active probing of the suspected Tor relay, mimicking the initial sequence of Tor protocol messages in detail [70]. The Tor developers defeated these blocks with small adjustments to their software.

For a few days in early 2012, Iran blocked *all* outbound HTTPS connections to many websites [60], including Tor's primary site.[1] To evade this more drastic blockade, Tor deployed a program called *obfsproxy* (for "obfuscating proxy") [21]. Obfsproxy applies an additional stream cipher to Tor's traffic. This frustrates any filter looking for a specific plaintext pattern (such as a TLS handshake), but does not significantly alter packet sizes and timing. As we will discuss in Section 5.1, this means that Tor is still easily fingerprintable.

Contributions: In this paper, we present an elaboration on the obfsproxy concept, StegoTorus. StegoTorus currently consists of:

- A generic architecture for concealing Tor traffic within an innocuous "cover protocol" (Section 2).
- A novel encrypted transport protocol geared specifically for the needs of steganography (Section 3).
- Two proof-of-concept steganography modules (Section 4).

We will demonstrate the ease of detecting un-camouflaged Tor traffic and StegoTorus' effectiveness at concealing it, even with the current proof-of-concept steganography (Sections 5.1 and 5.2). We will also demonstrate that StegoTorus imposes a reasonable amount of overhead for what it does (Section 6).

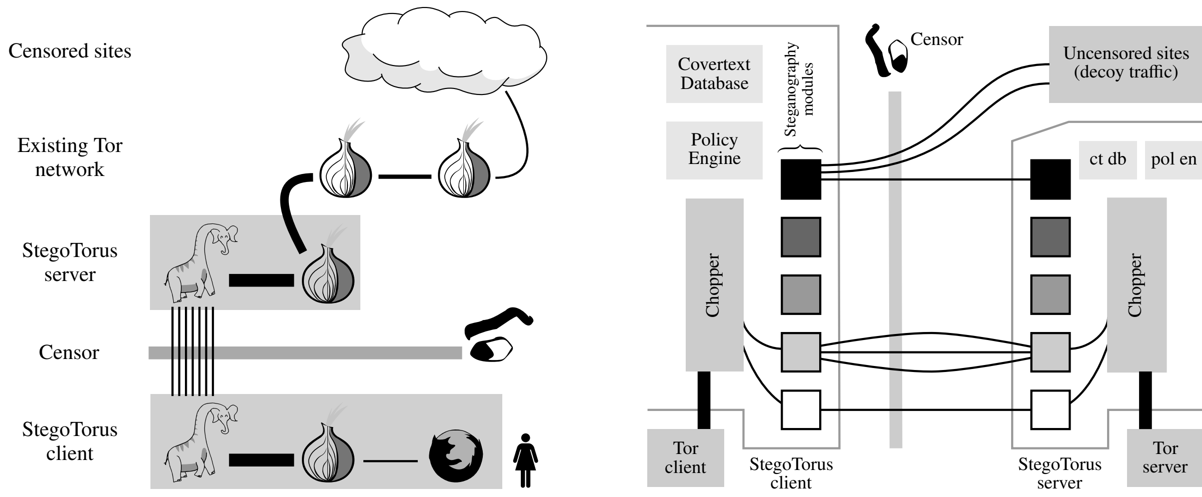
We anticipate that censors will adapt quickly to this advance on the circumvention side of the arms race; more sophisticated and varied steganography modules are under active development. Ultimately, an attacker will need to defeat *all* of the steganography modules used by StegoTorus to block Tor traffic.

2. ARCHITECTURE

StegoTorus acts as a "pluggable transport" [47] for Tor, replacing its usual direct connection to a relay server. Pluggable transport is an extension of SOCKS [43], so StegoTorus could also camouflage traffic produced by other applications that can use a SOCKS proxy.

ACM, 2012. This is the author's version of the work. It is posted here by permission of ACM for your personal use. The definitive version will appear in *CCS'12*, October 16–18, 2012, Raleigh, North Carolina, USA.

[1] `https://www.torproject.org/`



(a) Data flow. The user's browser uses Tor as a SOCKS proxy; Tor uses StegoTorus as *its* SOCKS proxy. StegoTorus disguises the Tor link as innocuous cover-protocol traffic, perhaps split over many TCP connections, that pass through the perimeter filters and reach the StegoTorus server. The server decodes the steganography and passes Tor traffic to the relay network.

(b) Internal architecture. The *chopper* distributes re-encrypted Tor traffic to *steganography modules*, which conceal it within cover-protocol messages. Steganography modules can also generate decoy traffic directed at uninvolved hosts. Counterpart modules in the recipient decode the messages and reassemble the Tor link. Both sides have access to independent *covertext databases*. Overall control rests with a configurable *policy engine*.

Figure 1: High-level overview of StegoTorus.

Figure 1a shows how the Tor+StegoTorus system transports data between the user's browser and censored websites; Figure 1b shows the internal structure of the StegoTorus client and server. StegoTorus applies two additional layers of obfuscation to Tor traffic:

Chopping converts the ordered sequence of fixed-length "cells" that Tor produces, into variable-length "blocks" that do not have to be delivered in order. Each block is re-encrypted using a novel cryptosystem geared for the needs of steganography: every byte of its output is computationally indistinguishable from randomness.

Chopping can be used by itself for minimum overhead; since its output has no predictable content and randomized packet sizes, this is enough to defeat all known pattern filters and the attacks described in Section 5. However, a pattern filter that *only* passes protocols with known, recognizable headers would block it.

Chopping produces "blocks" that do not have to be delivered in order. However, they must be delivered reliably. At present, all our cover protocols run over TCP. StegoTorus cannot run directly over UDP, as Dust does [71], but it could run over DCCP, or a UDP-based cover protocol that provides reliable delivery.

Steganography disguises each block as a message in an innocuous *cover protocol*, such as an unencrypted HTTP request or response. Since blocks can be delivered out of order, StegoTorus can distribute a Tor link over many cover connections, improving both efficiency and difficulty of detection. A StegoTorus server can listen on many IP addresses, so that its clients appear to be talking to many unrelated servers. StegoTorus clients can generate decoy traffic to uninvolved hosts, making detection even more difficult.

2.1 Design Goals

StegoTorus preserves Tor's basic design goals:

Unlinkability: The censor should not be able to determine which Internet users communicate with which remote hosts via Tor.

Performance: Unlinkable access to the Internet should not be so much slower than "unmasked" access that users will reject the trade-off.

Robustness: The system should preserve its other design goals in the face of active attacks.

StegoTorus also seeks to provide:

Undetectability: The censor should not be able to determine which Internet users are using StegoTorus.

Unblockability: The censor should not be able to block StegoTorus without also blocking a great deal of unrelated traffic.

The terms "unlinkability" and "undetectability" are defined precisely by Pfitzmann and Hansen [59].

2.2 Threat Model

We model a censorious adversary more or less as Infranet [27] and Telex [72] do. A censor has a *network perimeter*, which cuts the global connectivity graph into two disconnected components. One of these components is "inside" (or "censored") and the other "outside." The censor controls all the network infrastructure inside the perimeter, but *not* the software on end users' computers. (Attempts to mandate censorware on end users' computers, such as China's 2009 "Green Dam" initiative, have so far been unsuccessful [73].)

The censor wishes to prevent the censored nodes from retrieving material that meets some definition of undesirability; we assume that no such material is *hosted* inside the perimeter.

2.2.1 Perimeter Filtering

The censor programs the routers for all perimeter-crossing links to observe all cleartext traffic that they forward. This includes any cleartext portion of a mostly-encrypted protocol, such as IP and TCP headers and TLS record framing. Using three general techniques, the routers detect undesirable material and prevent it from crossing the perimeter.

Address filters prevent all communication with the IP addresses of servers that are thought to host undesirable material. China maintains a blacklist of Tor entry nodes as part of its "Great Firewall."

Pattern filters look for deterministic patterns in cleartext that may indicate undesirable material. As mentioned in Section 1, Iran was able to block all use of Tor for a few weeks with a pattern

filter looking for a particular Diffie-Hellman public modulus in TLS handshakes.

Statistical filters look for stochastic patterns, and can take low-level packet characteristics (size, arrival time, etc.) into account as well as cleartext headers and payloads. While statistical filters for Tor are easy to construct (we describe one in Section 5.1), we are not aware of any use of them in the field, to date.

2.2.2 *Limits on the Censor*

Perimeter filtering must operate in real time on a tremendous traffic volume. To give some idea of the necessary scale, the CAIDA project's "Anonymized Internet Traces 2011" data set [14] consists of the first 64 bytes of every packet that traversed a backbone router in Chicago for one hour on a Wednesday afternoon; there are 1.96 billion packets in the set, for a total of 116 gigabytes of data. This corresponds to an average traffic rate of 540,000 packets per second; a filtering router that needs an extra two microseconds to process each packet will *halve* overall throughput.

The precise capabilities of commercial "deep packet inspection" hardware are not widely advertised. We assume, in general, that a nation-state adversary has access to equipment that can perform a two-stage analysis. The first stage sees every packet, runs with a hard realtime deadline, and must judge the vast majority of packets to be uninteresting. The second stage can only examine a tiny fraction of the TCP flows crossing the router, and may be limited to responding after-the-fact. This is consistent with the observed behavior of China's active probes for Tor bridges [70].

We assume all Tor relays are outside the perimeter, and the censor does not operate malicious Tor relays, nor does it observe traffic among outside nodes. If any of these assumptions are false, the censor may be able to break Tor's unlinkability guarantee [50, 53]. StegoTorus obfuscates the traffic between the Tor client and the first Tor relay. Since the client is inside the perimeter, and the relay outside, StegoTorus controls what the perimeter routers observe.

We also assume that the censor does not "turn off the Internet" (that is, disconnect from the global network). This *was* done by several countries during the Arab Spring, but only for a short time, in response to imminent existential threat, and with negative consequences for those who tried it. We expect that other governments will not repeat this mistake. Similarly, we assume that address filters prevent communication with only a relatively small number of IP addresses. Unlike systems such as Telex and Cirripede [38], however, StegoTorus can potentially work even if the only protocol allowed to cross the perimeter is unencrypted HTTP.

It is difficult for two parties to communicate securely if they have never communicated securely in the past. Tor users must first obtain the Tor client and learn the address of at least one relay, via some extra-Tor method. Similarly, StegoTorus users must obtain the StegoTorus client (as well as the Tor client) and learn the address and public key of at least one StegoTorus server. We assume that all necessary software and an initial server contact can be smuggled over the perimeter, via "sneakernet" if necessary.

Since we anticipate that StegoTorus servers will be aggressively blocked with address filters, we are developing a "rendezvous" mechanism for distributing server address updates to StegoTorus users [45]. In this paper we assume that the user knows contact information for server(s) that have not yet been blocked.

3. CHOPPING AND REASSEMBLY

As we described briefly in Section 2, chopping converts the traffic on a Tor link into a more malleable format: a sequence of variable-size *blocks*, independently padded and deliverable out of order. Every byte of each block is computationally indistinguishable from randomness, as defined in [62]; this is a baseline requirement for the hiddentext in theoretically secure steganographic schemes. [37, 66] The module that performs this job (and its inverse) is, naturally, called the *chopper*.

The block format is shown in Figure 2a. The bulk of each block is encrypted using AES in GCM mode [25], which provides both confidentiality and message integrity [7]. The block header consists of a 32-bit sequence number; two length fields, d and p, indicating respectively how much data and padding the block carries; an opcode field, F, discussed below; and a 56-bit check field, which must be zero. The minimum block length is 32 bytes (128-bit header, 128-bit MAC) and the maximum is $2^{17} + 32$ bytes. Block length is controlled by the steganography modules; the chopper will fabricate blocks exactly as long as requested, using data if possible, padding if there is not enough. Padding consists of binary zeroes. Blocks containing only padding ($d = 0$) are generated when there is no data available but the cover protocol requires transmission.

The sequence number permits the receiver to sort incoming blocks into their original order. It serves the same function as a TCP sequence number, but it always starts at zero, counts blocks rather than bytes, and may not wrap around (see Section 3.3). It also serves to ensure that the same header is never transmitted twice. This is important because the header must also be encrypted to render it indistinguishable from randomness, and needs integrity protection to preclude chosen-ciphertext attacks [2, 9], but we can't include it in the data authenticated by GCM because we have to decrypt d and p in order to know where the authentication tag begins.

Instead, we protect the header with a custom short-message authenticated encryption mode that relies only on the basic AES pseudorandom permutation. The check field brings the header up to exactly the AES block size, and we encrypt it as a standalone block with a different key from that used for the payload. Before the receiver acts on a decrypted header, it verifies that every bit of the check field is zero, and that the sequence number is within a 256-block-wide receive window. An active attacker who modifies the ciphertext of the header has less than one chance in 2^{80} of passing this verification. We recycle the *ciphertext* of the header as the GCM nonce for the payload.

3.1 Function Codes

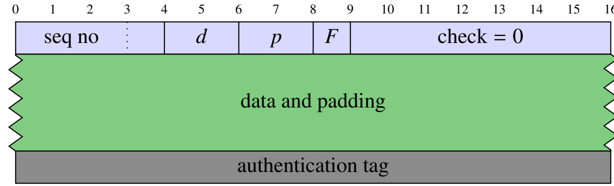
The F field of the block header controls how the receiver will process the block. All presently-defined codes are listed in Table 1. Some codes are only valid in handshakes; see below.

No.	Name	Semantics
0	DATA	Application data to be relayed.
1	FIN	Last block of application data to be relayed.
2	RST	Protocol error; close the link immediately.
3	RC	Reconnect: associate this new connection with an existing link.
4	NC	New link, client side. See section 3.2 for details.
5	NS	New link, server side.
6	RKI	Initiate rekeying; see Section 3.3 for details.
7	RKR	Respond to rekeying.
8–127		Reserved for future definition.
128–255		Reserved for steganography modules.

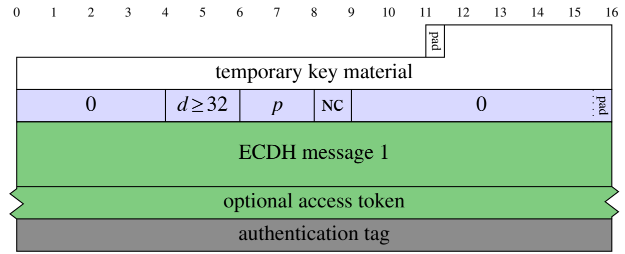
Table 1: Codes for the F field

3.2 Handshake Messages

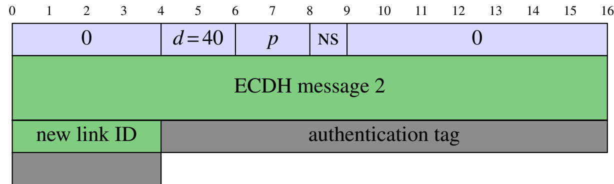
The first few bytes of data sent on each new connection are a *handshake message* (henceforth just "handshake"), which informs



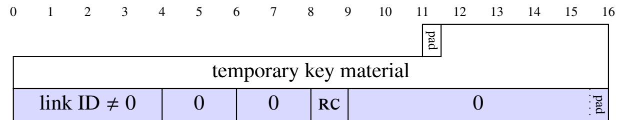
(a) Blocks consist of a header, up to $2^{15} - 1$ bytes each of data and padding, and an authentication tag. d: data length; p: padding length; F: function code. $d + p$ is not required to be a multiple of 16.



(b) A new-link request consists of a Möller key encapsulation, followed by a block encrypted with the temporary key material. Four padding bits are copied into the "check" field to make them non-malleable.



(c) A new-link response is a normal block, encrypted with the temporary key material.



(d) A reconnection request is just a key encapsulation and a modified header. The link ID replaces the sequence number, d and p must be zero, and there is no authentication tag.

Figure 2: Message formats before steganography. Background shading indicates encryption mode. ■: header encryption; ■: GCM encryption; ■: GCM authentication tag; □: Möller key encapsulation. There are 16 bytes per row of each diagram.

the server whether this connection belongs to an existing link or to a new one. If the connection belongs to a new link, the server replies with a handshake of its own, and both peers derive new session keys for the link from the data in the handshakes. There are three handshake formats, shown in figures 2b, 2d, and 2c. Handshakes have similar overall structure to blocks, but vary in details.

Most asymmetric cryptosystems' ciphertexts are easily distinguished from randomness. We use Möller's elliptic-curve key encapsulation mechanism [52], which is designed to produce random ciphertexts. It can only be used to establish a weak shared secret, which we refer to as "temporary key material." It has the unfortunate property of producing 164-*bit* messages, which must be padded to a whole number of bytes. The padding bits could be flipped by an adversary without any visible effect. To prevent this information leak, the *check* field of the header that immediately follows a key encapsulation contains a copy of the padding. Also, the server must maintain a replay cache of all key encapsulations it has seen recently, and discard any handshake with a replayed encapsulation, even if the data that follows is different.

Each link has a nonzero, 32-bit *link ID*. The server chooses this ID during link setup, making sure that it is unique among all active or recently-active connections to the same server. It is never transmitted in cleartext, so it need not be random.

Key derivation, whether from the temporary key material or from Diffie-Hellman exchanges, is done with HKDF-SHA256 [42], salted with the server's public key, and produces four 128-bit AES keys: server-to-client payload key, server-to-client header key, client-to-server payload key, and client-to-server header key, in that order.

3.2.1 New Link Handshake

Link setup is loosely based on the STS protocol [18] and provides forward secrecy. Initially, the client knows the server's public key. It has no asymmetric keypair of its own, but it may have an "access token" which will identify it to the server. This token is opaque to the client, and its contents are outside the scope of this paper.

The client's first message to the server is shown in figure 2b. It begins with a randomly chosen Möller key encapsulation, followed by a special block encrypted with the temporary key material. This block has sequence number 0, and its F code is RC. Its first 32 bytes are an ECDH message on the NIST standard curve P-256 [54], derived from a source of strong randomness. Only the x-coordinate of the public point is transmitted. If the client has an access token, it follows immediately after the ECDH message.

If the server can decrypt this handshake and finds the access token (if any) acceptable, it replies with its own handshake, shown in figure 2c. This is a normal block, also encrypted with the temporary key material provided by the client. It also has sequence number 0, its F code is RS, and its contents are another ECDH message and the link ID for the new link. Once the client receives this message, both sides can complete the Diffie-Hellman exchange and derive long-lived keys for the link. Subsequent blocks are encrypted with those keys. The handshakes count as sequence number 0 in each direction.

3.2.2 Reconnection Handshake

For new connections to established links, the client's handshake needs to be as short as possible. It is shown in figure 2d. As with a new-link handshake, it begins with a randomly chosen Möller key encapsulation, but instead of a block, only a modified header follows. This header has $p = 0$, $d = 0$, and F = RC, and it carries the desired link ID in place of the sequence number. Unlike normal blocks with $p = 0$ and $d = 0$, the GCM authentication tag is omitted. The client may transmit blocks, encrypted with the appropriate link keys, immediately after this handshake (that is, in the same cover-protocol message).

3.3 Rekeying

Rekeying is very similar to key derivation for a new link, but uses blocks rather than special handshake messages. Rekeying resets the sequence number but does not change the link ID. Either peer may initiate a rekeying cycle at any time by transmitting an RKI block. Peers are *required* to rekey before the sequence number wraps around, and encouraged to rekey considerably sooner.

An RKI block's data section is simply an ECDH message on curve

P-256. The recipient of this message responds with an RKR block, whose data section is another ECDH message. Upon receipt of RKR, both peers derive new link keys from the Diffie-Hellman exchange, just as they would have for a new link. It is a protocol error to transmit any blocks after RKI until receipt of RKR, or to transmit blocks using the old keys after transmitting RKR.

3.4 Link Termination

When both sides have sent and received a FIN block, the link is closed; however, both sides must remember the link ID for some time, to guard against replay attacks or delayed block arrival. It is a protocol error to transmit a DATA block with a nonzero d field after transmitting a FIN block; however, DATA blocks with $d = 0$ may still be sent, and other function codes may be used if appropriate. For instance, the RKI block requires a response, even if the recipient has already transmitted a FIN.

4. STEGANOGRAPHY

In this section, we describe two proof-of-concept steganography modules: one that duplicates the packet sizes and timings of encrypted peer-to-peer protocols, and one that mimics HTTP. These modules illustrate the flexibility and feasibility of the StegoTorus framework. However, they are not expected to resist sophisticated, targeted attacks that might be launched by a nation-state adversary. To underscore this, for each module we also describe potential attacks and the level of sophistication each requires.

More diverse and resilient modules are under development, both by us and the larger community, as the arms race continues. The StegoTorus client can be configured to use whichever modules the adversary has not yet blocked; therefore, ultimately, the adversary will have to detect and block traffic generated by *all* of the steganography modules in order to block StegoTorus.

4.1 Embed Module

The *embed* steganography module conceals Tor traffic within an encrypted, peer-to-peer cover protocol, such as the popular Skype and Ventrilo protocols for secure voice over IP. (Ventrilo is not strictly a peer-to-peer protocol, but its users typically set up their own servers, so there is no small, stable set of server IPs that could be whitelisted.) Since the audio payload of each packet is encrypted, we can substitute our own encrypted data without fear of payload inspection. This leaves cleartext headers, packet sizes, and inter-packet timings as the characteristics visible to the censor.

4.1.1 Packet Traces

This module relies on a database of *packet traces*, pre-recorded sequences of packet sizes and timings from real sessions of the cover protocol. Client and server match the recorded packet sizes exactly, and timings to the nearest millisecond. If there is no data available when a packet should be sent, they will fabricate padding-only blocks to maintain the deception.

The server does not maintain its own database of traces; instead, the client transmits its chosen trace to the server as a special control message, immediately after link setup. Some of these traces are distributed with the software, but users are encouraged to capture their own use of the cover protocol, so that the censor cannot block StegoTorus by pattern-matching against the distributed set of traces. (Packet timings seen by the client may or may not correspond to packet timings as actually transmitted by the server. For greater realism, one should capture a trace from both ends, but we have not implemented this yet.)

Potential Attacks: Some VoIP protocols permit an eavesdropping adversary to learn much about the speech being transmitted, just from packet sizes and timings [68]. Therefore, if traces are reused too often, the censor might become suspicious of users apparently having the exact same conversation over and over again.

4.1.2 Application Headers

The packet trace does not attempt to capture application headers, as these may depend on the substituted contents. Instead, the *embed* module includes emulation code for each potential cover protocol. Unfortunately, neither the Skype nor the Ventrilo protocol has a public specification, necessitating reverse engineering. To date this has been done by hand, but we are investigating the possibility of automating the process [12, 15, 44].

Potential Attacks: If the censor has access to the true protocol specification for a protocol we have reverse engineered, they may be able to detect deviations on our part.

Even if the censor doesn't have this information, it might choose to block *all* apparent VoIP protocols, or all peer-to-peer traffic that appears to contain encrypted data. These are popular, but not yet so popular that this would amount to "turning off the Internet," and there are plausible political cover stories for such actions by a nation-state: preserving telephone revenue, combating copyright infringement, etc.

4.2 HTTP Module

The *HTTP* steganography module simulates unencrypted HTTP traffic. Since the censor can observe the overt content of this module's traffic, and protocol decoders for HTTP are ubiquitous, we take care to mimic "real" browser and website behavior as accurately as possible.

HTTP [28] follows a strict pattern: the client sends a *request*, waits for the server to produce a *response*, can then send another request, and so on. HTTP 1.1 allows the client to send several requests in a row without waiting for responses ("pipelining") but this is rarely used, due to server bugs [55]. Instead, clients achieve parallelism by opening multiple connections to the same server. Each request contains a "method" (GET, POST, etc) that controls what the server will do to prepare the response.

The *HTTP* module also relies on a database of pre-recorded HTTP requests and responses; we also refer to these as "traces." Like the *embed* module's traces, some are distributed with the program, and users are encouraged to record their own. Unlike *embed*, requests and responses are not organized into a temporal sequence, and client and server use independent databases. However, the server generates responses that are consistent with client requests; for instance, if a client sends a request for a PDF document, the server will produce a PDF covertext.

4.2.1 Request Generator

Normal HTTP client-to-server traffic consists almost entirely of GET requests. Unfortunately, these provide very little space to conceal our hiddentexts. Here is a typical request template from our database:

```
GET /<uri> HTTP/1.1
Accept: text/html,application/xhtml+xml,
    application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-us,en;q=0.5
Connection: keep-alive
Host: <host>
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:10.0)
    Gecko/20100101 Firefox/10.0
Cookie: <cookie>
```

Nearly all of this is boilerplate that must be sent verbatim in every request. Data can be inserted at each position marked `<...>`, but it must be properly encoded.

At present we only store hiddentext in the <*uri*> and <*cookie*> positions, which can carry arbitrary textual data. We encode the binary chopper output in a modified `base64` alphabet [39] that avoids characters with special meaning in URIs or cookies: '+' is replaced by '-', '/' by '_', and '=' by '.'. We then insert characters at random positions in the encoded string, to make it look more like a genuine URI or cookie header: for URIs we insert '/', for cookies we alternate between '=' and ';'.

The <*host*> can theoretically also carry hiddentext, but with more difficulty: <*host*> must have the form of a DNS hostname or IP address [28], and the censor could block HTTP connections where it was not a registered hostname for the IP address to which the client was connected. Presently, we do not attempt this; instead we do a reverse DNS lookup on the server's IP address, and use the first reported name in every request to that address.

Potential Attacks: To the human eye, the HTTP request generator's URIs and cookies likely look different from normal URIs or cookies. The pattern of requests that it generates is also potentially different from the pattern of requests generated by a visit to a real website. Hence, it would be possible for an adversary to build a machine classifier that can make the same judgment. The cookie string we send changes on every request, without the server sending back `Set-Cookie:` directives; this could also be a distinguisher, as a real web browser only changes cookies when instructed to. If such attacks become common, they we may be limited in our use of cookies as a carrier channel. More sophisticated cookie, URI, and request pattern generation is also possible; Infranet [27], for instance, devotes some effort to this problem. However, substantially more overhead will be required.

The `User-Agent` header identifies the browser and operating system in use. If the same client IP address consistently produces one user-agent, except during a handful of browsing sessions, that handful might attract attention. Generating a database of client requests on each user's machine ensures that we generate user-agent headers matching the browser that that user normally uses.

The censor may conduct active attacks by replaying HTTP requests; a real web server would normally produce the same response, but StegoTorus will not. To mitigate this we could place an off-the-shelf HTTP "accelerator" cache in front of the StegoTorus server so that, for a short time, replayed requests would produce the same response as the original.

4.2.2 Response Generator

HTTP responses begin with a few headers, similar to the ones shown above, but offering even less space for hiddentext. However, they continue with a "response body" which is *designed* to carry arbitrary data. That data typically conforms to some known file format, which must be consistent with the contents of the request. We have developed response generators that embed StegoTorus hiddentexts in three common file formats: JavaScript, PDF, and Flash. These data formats are complex enough to conceal hiddentexts easily, and pervasive enough that blocking them would break far too many popular websites to be politically tenable. Generators for HTML and various image, audio, and video formats are under development.

JavaScript Generator: JavaScript is a programming language, human-readable in its original form, but frequently "minified" to reduce its size on the wire. Minification involves removing all white space and replacing variable names with shorter machine-generated identifiers. There is an enormous volume of JavaScript in use on the open web: 2.5% of all bytes transferred by HTTP in early 2009 [46].

This generator picks a response containing JavaScript, scans it for identifiers and numbers, and replaces them with characters from the hexadecimal encoding of the hiddentext. To preserve syntactic validity, the encoder will not change the first character of an identifier or a number, and there is a blacklist of JavaScript keywords and built-in functions that should not be replaced. The decoder simply reverses the process. Our objective with this module is to produce syntactically valid JavaScript that cannot be trivially detected by a parser.

PDF Generator: PDF documents consist of a sequence of "objects," which define pages, images, fonts, and so on. Many of these objects will normally be compressed, using the ubiquitous "deflate" algorithm, to save space. The PDF response generator locates compressed objects within a PDF document from the HTTP response database, and replaces their contents with our hiddentexts. Chopper output is incompressible, but we apply the "deflate" transformation to it anyway, so that each modified object's contents is still superficially what it ought to be. The overall file structure is adjusted to match.

SWF Generator: Adobe (formerly Shockwave) Flash is a format for vector-graphic animations, and is also frequently used as a container for video. Flash files consist of a sequence of tagged data blocks, containing shapes, buttons, bitmaps, ActionScript byte code, etc. [1] Flash files may be compressed (CWS) or uncompressed (FWS). In the more common CWS format, the entire file (with the exception of a short initial header, but including the block framing) is compressed with "deflate." The SWF response generator uncompresses a template CWS file, replaces block contents with encrypted data, and recompresses the result.

Potential Attacks: The HTTP response generator attempts to preserve the syntactic validity of JavaScript, PDF and SWF files that it modifies. However, it does not attempt to preserve the semantics of JavaScript or the original content of PDF or SWF. Therefore, adversaries might be able to detect the use of the present HTTP generator by attempting to execute JavaScript, render PDF documents, or play back Flash animations. However, doing so at line rate on a border router would be quite challenging. The filter would have to extract HTTP response bodies of interest, reassemble packets into streams, and then parse and decode the contents of the file; all of these are expensive and complicated operations. Providing the appropriate execution environment for JavaScript requires the adversary to assemble and process all the data of the surrounding webpage, just as a browser would.

Nonetheless, we do expect that if StegoTorus comes into wide use, filtering routers will gain the ability to detect these simple schemes. In particular, a natural escalation of the arms race might involve the use of cascading detectors, where a series of fast filters select traffic to subject to more expensive analyses. If this happens, we would have to implement more sophisticated steganography.

5. DETECTION RESISTANCE

To evaluate how well StegoTorus can conceal a Tor stream, we developed two attacks upon Tor, which StegoTorus ought to defeat if it is functioning as intended. We designed these attacks to be practical in the resource-constrained environment of a perimeter filter (as described in Section 2.2.2) so they are deliberately quite simple. The first attack picks Tor streams out of other TCP streams, based on a fundamental characteristic of the Tor wire protocol that is cheap to detect. The second attack operates on known Tor streams and extracts information about the sites being visited covertly. In each case we will first describe the attack and how it fares against Tor, then discuss its effectiveness against StegoTorus. As we did for the steganography modules, we will also discuss potential improvements to the attacks that might be implemented by an adversary determined to detect StegoTorus.

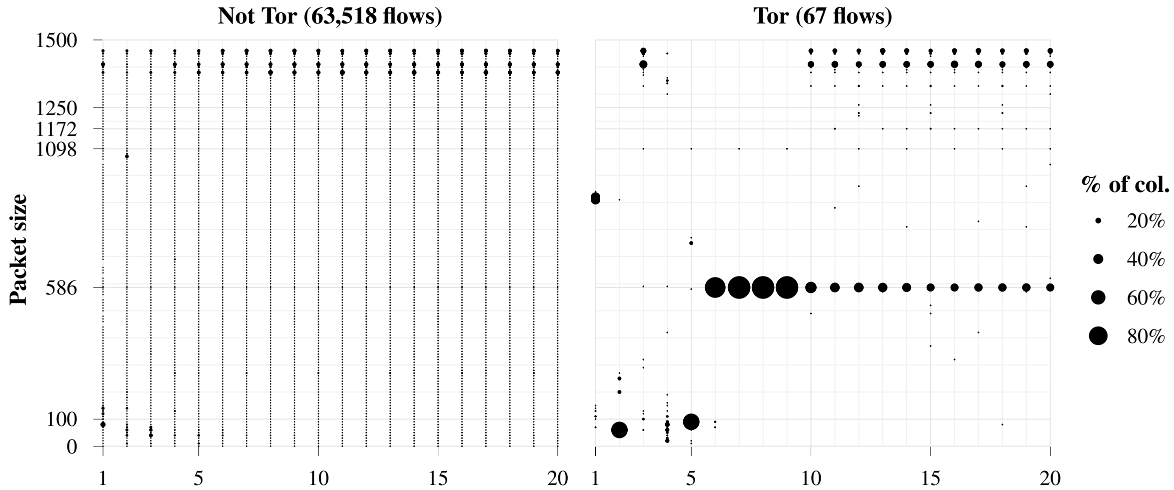


Figure 3: Payload lengths for the first 20 non-empty packets of 63,585 unidirectional flows on TCP port 443, taken from the CAIDA 2011-Chicago data set. [14] Port 443 is officially assigned to HTTP over TLS, but Tor relays are sometimes configured to accept connections on this port. Scanning for 586-byte TCP payloads identifies 67 of the flows as probable Tor traffic.

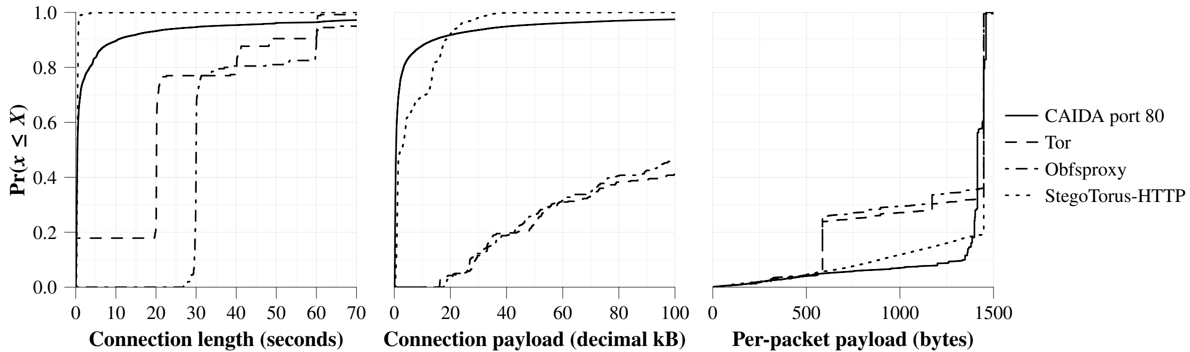


Figure 4: Empirical CDFs of connection length, total data transferred, and per-packet payload for 20 visits to each of the Alexa top ten websites, using Tor directly (dashed line), obfsproxy (dot-dash line), and StegoTorus-HTTP (dotted line). CAIDA Chicago-2011 port 80 traffic for reference (solid line).

5.1 Detecting Tor

The Tor protocol [22] sends nearly all messages in the form of "cells" with a fixed length of 512 bytes. These cells are packed into TLS 1.0 application-data records [17] on the wire. Because of the "empty record" countermeasure for a cryptographic weakness in TLS 1.0 [5, 6], the overhead of TLS encapsulation is 74 bytes per application-data record. The client will pack many cells into a record if it has enough to say, and TCP will split large records in the middle in order to transmit MTU-sized packets, but Tor nonetheless winds up transmitting many TCP packets that contain exactly one cell. These packets have a characteristic payload length of 586 bytes. A filtering router can pick Tor streams out of other traffic by counting how often they appear.

We implemented the following concrete algorithm: let τ be the adversary's current estimate of the probability that a given TCP flow is Tor, initially set to zero. Ignore packets containing only an ACK and no payload; these are best treated as neutral [57]. Otherwise, update $\tau \leftarrow \alpha\tau + (1-\alpha)\mathbf{1}_{l=586}$ where $\alpha \in (0, 1)$ is a tuning parameter and $\mathbf{1}_{l=586}$ is one if the TCP payload length l equals 586 bytes, zero otherwise. If τ rises past a threshold value T, the TCP flow is considered Tor traffic.

To do this, a perimeter filter must be capable of tracking TCP flows in realtime, and maintaining one scalar value (the estimate τ) for each; to the best of our knowledge, this is within the capabilities of modern DPI hardware. Empirically, $\alpha = 0.1$ and $T = 0.4$ identifies Tor within a few dozen packets. Figure 3 shows probable Tor flows picked out of all the port-443 traffic in the CAIDA 2011-Chicago data set [14] with this technique. (This data set only includes IP and TCP headers for each packet captured, so we are unable to confirm that the selected flows are actually Tor, or how much of the background traffic is in fact HTTPS.)

To confirm the effectiveness of this attack against vanilla Tor, we collected traffic traces from visiting the top ten Alexa sites twenty times over vanilla Tor, obfsproxy [21], and StegoTorus with the *HTTP* steganography module. In addition to non-zero TCP payload sizes for each packet, we extracted the lifetime and total data transferred (treating the two directions as independent) of each TCP stream. Figure 4 presents a qualitative comparison of these features in the form of empirical CDFs, with all TCP flows on port 80 of the CAIDA 2011-Chicago data set (again, we cannot confirm this, but port 80 traffic on the public Internet is almost surely HTTP) for reference. Tor's predilection for generating 586-byte packets is clearly seen in the rightmost panel of Figure 4, and the other panels show other characteristics that would be easy for the adversary to detect, such as a tendency for TCP connections to last exactly 20 or 30 seconds. Obfsproxy does little to alter these features.

StegoTorus fares much better. It is not perfect, but it generates empirical CDFs for all three features that are closer to the CAIDA port 80 reference than they are to either Tor or obfsproxy. In par-

ticular, it eliminates the 586-byte characteristic payload size. Still, a determined adversary with more analytic power at its disposal might be able to detect the remaining statistical differences between StegoTorus-HTTP and "normal" HTTP traffic that Figure 4 reveals. Improving our HTTP emulation will reduce these differences. If necessary, we could also implement an explicit statistical model of what HTTP traffic "should" be like.

5.2 Identifying Visits to Facebook

Once the censor has identified TCP streams as Tor traffic, they would also like to learn which sites are being accessed clandestinely. We present a simple method to determine whether a Tor user is visiting Facebook; this site has sometimes been completely blocked by government censors. It is a cut-down version of Panchenko et al. [57], which can identify accesses to a small set of censored websites within a larger Tor session. Their classifier uses a support vector machine, which is too expensive to run on a filtering router, even on a small number of streams. With careful optimization, our classifier requires a handful of probability calculations per arriving packet, plus maintenance of a sliding-window vector per stream under surveillance; this should be acceptably cheap.

Once a stream has been identified as Tor traffic, the censor maintains a pair of sequences, u_i and d_i, sliding over the last n non-empty packets observed by the filtering router. Each u_i is the cumulative sum of payload lengths for packets 1 through i sent "upward" (client to relay), and d_i is the same for packets sent "downward" (relay to client). The censor has previously observed "typical" Facebook traffic, and modeled the probability distributions $\Pr[U_i]$ and $\Pr[D_i]$ that one would expect to see if a trace were a visit to Facebook. Using this model, the censor computes

$$\log \Pr[\{u_i\}, \{d_i\} \text{ is Facebook}] = \sum_{i=1}^{n} \log \Pr[U_i = u_i] + \sum_{i=1}^{n} \log \Pr[D_i = d_i]$$

Log-probabilities are used to avoid floating-point underflow, since the per-packet probabilities can be very small. If the overall log-probability exceeds a threshold, the censor classifies the traffic as a visit to Facebook.

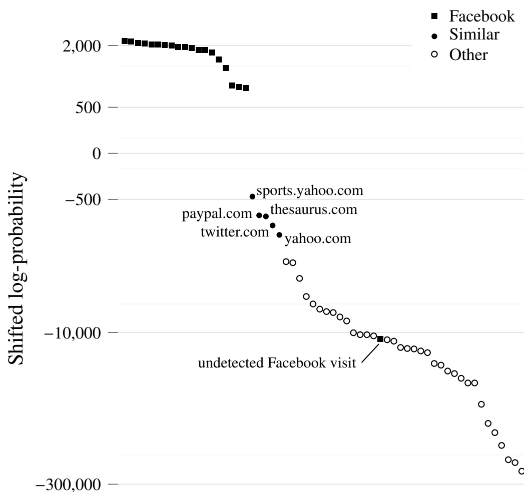


Figure 5: Log-probabilities reported by Facebook classifier, shifted to place the classification threshold at zero on the y-axis. Visits to Facebook (squares) show the shifted log-probability for the first 250 packets. Visits to non-Facebook sites (circles) show the maximum shifted log-probability observed for a 250-packet sliding window.

We trained this classifier on the first 250 packets transmitted in each direction over ten visits to the Facebook home page (login screen), and modeled the probability distributions as independent Gaussians for each position in the sequence. This is a deliberate departure from reality: U_{i+1} has a strong dependence on U_i, since they are cumulative sums, but treating them as independent makes the classifier robust to variation in the order of resources downloaded. We then tested it on 20 more visits to Facebook, plus 40 visits to other web sites chosen from Alexa's categorized directory of popular sites [3]. For all of the test visits, we browsed randomly until we had somewhere between 5,000 and 30,000 TCP packets; this resulted in a total of over 450,000 total packets and 500MB of Tor traffic. Figure 5 shows the results. Only one of the Facebook visits is not detected, and none of the other sites are misdetected as Facebook.

We augmented this attack to detect visits to nine of the top ten Alexa sites.[2] The classifier described above is intrinsically binary: site-X or not-site-X. An adversary wishing to know *which* of some set of sites was being visited would have to run a classifier for each site in parallel, suffering additional resource costs proportional to the number of sites. If the adversary cares only about visits to a fairly small number of sites, this will not be a significant problem.

For each site, we trained a classifier using the same procedure as described above for Facebook, using traces for ten visits to its front page. A traffic stream generated by a real user would not stop after loading the front page of whatever site he or she was visiting. Therefore, we adjusted the training window size for each site to ensure that the classifier did not simply learn the overall amount of data involved in loading the front page. We then tested each binary classifier on an additional ten visits to the target site, plus ten traces for each of the other eight sites.

For test runs with "vanilla" Tor, we took the best classification result obtained among four different window sizes: 50, 100, 150, and 200 packets. For test runs with StegoTorus, we added a 500-packet window, since StegoTorus-HTTP generates a much larger volume of traffic. We present classification accuracy in Table 2, as trapezoidally approximated AUC scores (area under the receiver operating characteristic curve) for Tor and StegoTorus visits to each of the nine sites. AUC scores allow evaluation of classifier effectiveness without first having to choose a tradeoff between false negatives and false positives.

Web Site	Tor	StegoTorus
Google	0.9697	0.6928
Facebook	0.9441	0.5413
Youtube	0.9947	0.4125
Yahoo	0.8775	0.7400
Wikipedia	0.9991	0.7716
Windows Live	0.9403	0.6763
Blogspot	0.9825	0.6209
Amazon	0.9841	0.8684
Twitter	0.9944	0.7366

Table 2: AUC scores for detecting visits to nine of the Alexa top ten sites' front pages, over Tor and StegoTorus.

An AUC score of 0.5 indicates a classifier performing no better than random guessing, and a score of 1 indicates perfect accuracy. Over Tor by itself, we can often obtain AUC scores better than 0.95, but over StegoTorus, the scores drop to 0.75 or less in most cases. For real-time classification of the traffic volume seen at a perimeter router, the adversary requires an AUC score very close to 1 to avoid being swamped by errors. StegoTorus does not reduce

[2] `baidu.com` was excluded because visits to this site did not exchange enough packets to perform a meaningful analysis.

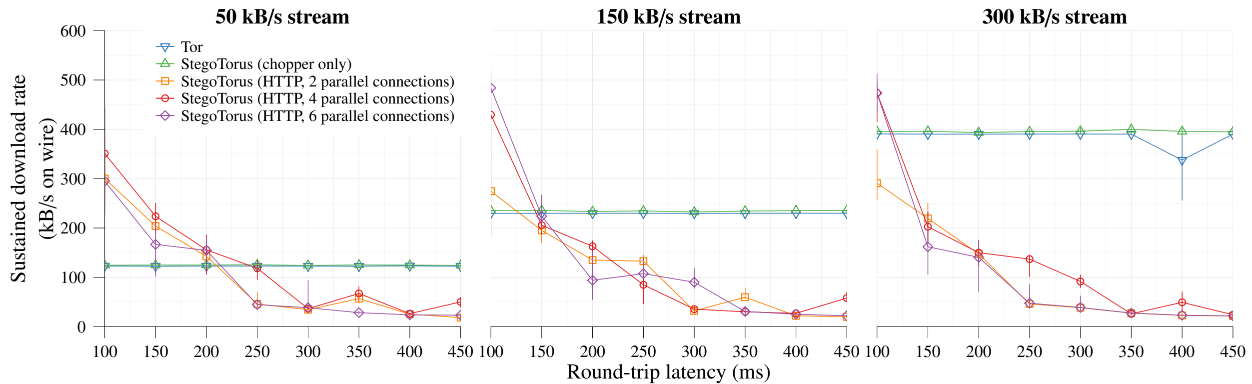


Figure 6: Overhead and resilience of StegoTorus' *HTTP* steganography module, compared to Tor alone and to StegoTorus in chopper-only operation. Each data point shows median bandwidth consumption over a 20-second interval while transferring a continuous, fixed-rate stream of traffic over a 5.6 Mbps DSL link with adjustable latency. Whiskers indicate the inter-hinge distance [49]. Tor and chopper-only StegoTorus do not suffer from latencies up to at least 450 ms. The present *HTTP* module is much more sensitive to latency, and cannot keep up with high-bandwidth streams at higher latencies.

the adversary's odds all the way to chance, but it reduces them far enough to make the attack impractical.

A determined adversary might train additional classifiers on visits to sites of interest *over StegoTorus*. However, these classifiers will be dependent on the covertext database that the adversary used for training, so StegoTorus users who generate their own covertext databases will be protected from this tactic.

6. PERFORMANCE

All performance tests were conducted using a desktop PC in California with a DSL link to the Internet (5.6 Mbit/s down, 0.7 Mbit/s up) as the client, and a virtual host in a commercial data center in New Jersey as the server. During testing, the DSL link was otherwise idle, and round-trip latency between the two machines was 85 ms. To ensure that our results reflect the performance of StegoTorus itself rather than factors beyond our control (such as the instantaneous load on the global Tor network), we configured a private Tor network entirely within the server host, and sourced all of the test files and streams from an HTTP server also running on that host.

Steganographic Expansion: We performed a series of downloads of 1,000,000-byte files and measured the amount of data actually transferred over the network by a direct HTTP connection, Tor, StegoTorus using the chopper alone, and StegoTorus with the *HTTP* module. The results are shown in Table 3. Tor itself has a small amount of overhead, and the chopper imposes a little bit more, but HTTP steganography is very expensive by comparison, increasing the amount of data sent upstream by a factor of 41, and downstream by 12. While we have not spent much time on optimizing our encoding, an expansion factor of at least eight (one byte per bit) is typical for modern steganography schemes [10], so we suspect that HTTP steganography cannot be made *that* much more efficient in the downstream direction.

Goodput: Expanding on the previous experiment, we conducted more downloads of files of various sizes, measured the time required, and computed the mean goodput (that is, application-layer throughput) achieved in the same configurations described earlier; the results are shown in Figure 7. We see that the goodput of the chopper-only configuration is comparable to Tor, and that StegoTorus-HTTP is only able to achieve goodput of roughly 27 kB/s, which is still about 4 times better than a 56 kbit/s modem. Consistent with this, we have been able to use StegoTorus-HTTP as-is for casual Web browsing; subjectively speaking, it is noticeably slower than a direct broadband connection, but not so slow that waiting for page loads becomes tedious.

	To server bytes	×	From server bytes	×
Direct	23,643	1	1,014,401	1
Tor	61,162	2.6	1,075,715	1.1
StegoTorus (chopper)	63,061	2.7	1,084,228	1.1
StegoTorus (HTTP)	966,964	41	11,814,610	12

Table 3: Mean number of bytes transferred in each direction in order to download a 1,000,000-byte file directly, over Tor, and over StegoTorus with and without HTTP steganography. The file was downloaded 32 times for each test.

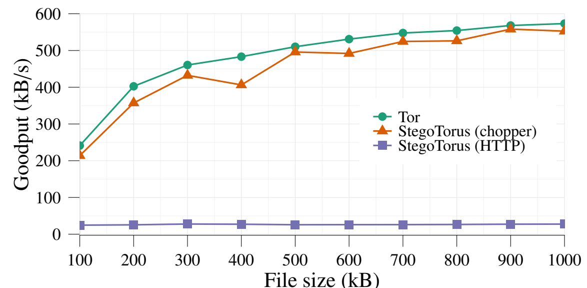


Figure 7: Mean StegoTorus (ST) and Tor goodput for 100 kB to 1 MB file transfers.

Resilience: Since StegoTorus may be used in locations with poor connectivity, we also investigated its performance as a function of network latency. We used Linux's *netem* mechanism [34] to vary the round-trip latency to the StegoTorus server from 100 to 450 ms. For comparison, packets from California take approximately 85 ms to reach New Jersey, 120 to 180 ms to reach East Asia or Australia, and 300 to 350 ms to reach India or Africa.

We configured the server to generate continuous streams of data at three different rates, thus measuring steady-state behavior, and recorded the median bandwidth consumption over a period of 20 seconds at each latency setting. The results are shown in Figure 6. Ideal behavior would be for each line to be perfectly horizontal, and as close to the "Tor" line as possible.

Both Tor and StegoTorus in chopper-only mode are robust up to 450 ms of delay, suffering no measurable performance degradation.

The *HTTP* module, however, can keep up with a 50 kB/s data stream only at latencies below 200 ms. Allowing *HTTP* to use more parallel connections increases throughput at low latencies, but does not help it keep up at high latencies. We suspect this is because each chopper block, which typically will only contain one or two Tor cells, takes a full HTTP request-response pair to transfer to the peer. Thus we are only transferring one or two Tor cells per round-trip cycle, despite consuming far more bandwidth. However, we believe that the *HTTP* module can be improved to handle high latency at least somewhat better, and plan to make this a priority for future work.

7. RELATED WORK

Here we summarize related research in the areas of blocking resistance and encrypted traffic analysis.

Address Filtering Resistance: The oldest technique for evading address filters is the use of open proxy servers, such as DynaWeb [26] and Ultrasurf [65]. However, with these services, users have to rely on the proxy administrator not to betray their browsing habits. Proxies also have publicly advertised and relatively stable IP addresses, so it is easy to block them in an address filter. While Tor relays come and go frequently, Tor's directory service is public, and there is nothing to stop a perimeter filter from blocking every entry point it lists. The obvious solution is to have many proxies whose addresses are not published; users have to find out about them via covert means. Köpsell and Hillig proposed these covert proxies in 2004 as an add-on to their AN.ON service [41]. The Tor Project calls them "bridge relays" and has deployed them extensively [19, 23].

Browser-hosted proxies [29] aim to make so many proxies available that it would be hopeless for a censor to block them all; there is still a global directory, but it is piggybacked on a cloud-storage service that is so widely used that the censor will hesitate to block it. Telex [72], Decoy Routing [40], and Cirripede [38] take a different approach to address-filtering resistance: TCP streams are covertly "tagged" to request that a router somewhere on the path to the overt destination divert the traffic to a covert alternate destination. Telex and Decoy Routing place the tag in the TLS handshake, whereas Cirripede uses the initial sequence numbers of several TCP connections. All three rely on the impenetrability of TLS to prevent the censor from making its own connections to the overt destination and comparing what it gets with the observed traffic, and may be vulnerable to large-scale traffic analysis as Tor is.

Pattern and Statistical Filtering Resistance: Infranet [27], like StegoTorus, conceals traffic that would otherwise be blocked within seemingly normal HTTP traffic. It works as a direct proxy for the browser, and does not provide Tor's anonymity guarantees; on the other hand, it can take advantage of its access to unencrypted network requests to reduce its bandwidth requirements. Dust [71] attempts to define a cryptosystem whose output is wholly indistinguishable from randomness; it is not a complete circumvention system by itself (but is under active development as a pluggable transport for Tor) and could theoretically be blocked by looking for the *absence* of any cleartext. SkypeMorph is a pluggable transport for Tor that makes Tor packet shape resemble Skype [51]. It is conceptually similar to our embed module, but lacks our generic chopper-reassembler and crypto framework. NetCamo [32] is an algorithm for scheduling transmissions to prevent traffic analysis; it is complementary to any of the above, and could also be deployed within existing relay servers to enhance their resistance to global adversaries. Collage [11] is a scheme for steganographically hiding messages within postings on sites that host user-generated content, such as photos, music, and videos. The sheer number of these sites, their widespread legitimate use, and the variety of types of content that can be posted make it impractical for the censor to block all such messages. However, it is suitable only for small messages that do not need to be delivered quickly, and it may be vulnerable to steganographic stripping [30].

Encrypted Traffic Analysis: There have been a number of studies on analyzing encrypted traffic to classify the application type [4] or the web site being visited in an encrypted HTTP stream [8, 36, 64]. These attacks usually extract some set of features based on the packet sizes, timings, and directions (essentially all of the available information when encryption is used) and use machine learning techniques to do the classification. Recently, some similar traffic analysis work has been done for clients who are using Tor. This naturally makes the task more difficult because Tor introduces two defenses [24]: combining all network traffic into one TCP stream to the first Tor router, and padding each packet to a fixed size (or a small set of sizes). Herrmann et al. [35] use a multinomial naïve Bayes classifier on the histogram of packet sizes and directions successfully against VPN technologies and OpenSSH, but they achieve under 3% accuracy against Tor while classifying on a set of 775 web sites. The work of Panchenko et al. [57], however, demonstrates that these defenses are not enough. Using support vector machines and a carefully selected feature set, they were able to achieve over 50% accuracy on the same classification task. This prompted the developers of Tor to introduce an ad hoc defense in the form of randomized pipelining [58] to defeat this type of classifier. Last, Wang et al. [67] present a potential application-level attack that involves serving malicious content and then observing a distinctive traffic pattern; although relevant, we are more interested in passive attacks that could be carried out on a large scale.

8. CONCLUSION

We described StegoTorus, a new system for improving Tor's robustness against automated protocol analysis. StegoTorus interposes on communications between the Tor client and the first Tor relay, implements a custom encryption protocol and provides a suite of steganography modules that make Tor resilient to fingerprinting attacks. Our statistical evaluations demonstrate that StegoTorus makes Tor traffic more resilient to site fingerprinting attacks and and that it resembles HTTP in the dimensions of connection length, payload size and bandwidth use. Our performance measurements indicate that our prototype system, in its HTTP steganography mode, delivers throughput of roughly 30 kB/s, which is about four times that of a 56 kbit/s modem.

Future Work: There is much room for further research on expanding and strengthening our suite of steganography modules. Beyond steganography modules we plan to explore opportunities for performance improvements and add support for redundant coding among steganography modules.

9. ACKNOWLEDGMENTS

We appreciate helpful comments during system and paper development from Arjun Athreya, Brian Caswell, Drew Dean, Bruce DeBruhl, Roger Dingledine, Pamela Griffith, David-Sarah Hopwood, George Kadianakis, Eric Kline, Patrick Lincoln, Nick Mathewson, Phil Porras, Steve Schwab, William Allen Simpson, Paul Vixie, and Mike Walker. Paul Hick and kc claffy at CAIDA [14] provided access to anonymized backbone traces. Data analysis was done in R [61] with the "ggplot2" plotting add-on [69].

This material is based on work supported by the Defense Advanced Research Project Agency (DARPA) and Space and Naval Warfare Systems Center Pacific under Contract No. N66001-11-C-4022, and by a National Science Foundation Graduate Research

Fellowship under Grant No. DGE-1147470. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the Defense Advanced Research Project Agency, Space and Naval Warfare Systems Center Pacific, or the National Science Foundation. Distribution Statement "A:" Approved for Public Release, Distribution Unlimited.

10. REFERENCES

[1] Adobe Systems Incorporated. *SWF File Format Specification (version 10)*, 2008. https://www.adobe.com/devnet/swf.html.

[2] Albrecht, M. R., Paterson, K. G., and Watson, G. J. Plaintext Recovery Attacks against SSH. In *IEEE Symposium on Security and Privacy* (2009), pp. 16–26. doi:10.1109/SP.2009.5.

[3] Alexa. Top Sites by Category. Data set, 2011. http://www.alexa.com/topsites/category.

[4] Bar-Yanai, R., Langberg, M., Peleg, D., and Roditty, L. Realtime Classification for Encrypted Traffic. In *Experimental Algorithms* (2010), vol. 6049 of *Lecture Notes in Computer Science*, pp. 373–385. doi:10.1007/978-3-642-13193-6_32.

[5] Bard, G. V. The Vulnerability of SSL to Chosen Plaintext Attack. Cryptology ePrint Archive, Report 2004/111, 2004. http://eprint.iacr.org/2004/111.

[6] Bard, G. V. A Challenging but Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL. Cryptology ePrint Archive, Report 2006/136, 2006. http://eprint.iacr.org/2006/136.

[7] Bellare, M., and Namprempre, C. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. *J. Cryptology 21*, 4 (2008), 469–491. doi:10.1007/s00145-008-9026-x.

[8] Bissias, G. D., Liberatore, M., Jensen, D., and Levine, B. N. Privacy vulnerabilities in encrypted HTTP streams. In *Privacy Enhancing Technologies* (2006), vol. 3856 of *Lecture Notes in Computer Science*, pp. 1–11. doi:10.1007/11767831_1.

[9] Black, J., and Urtubia, H. Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption. In *Proceedings of the 11th USENIX Security Symposium* (2002), pp. 327–338. http://www.usenix.org/event/sec02/full_papers/black/black_html/.

[10] Böhme, R. *Advanced Statistical Steganalysis*. Springer, 2010. http://www.springer.com/book/978-3-642-14312-0.

[11] Burnett, S., Feamster, N., and Vempala, S. Chipping Away at Censorship Firewalls with User-Generated Content. In *Proceedings of the 19th USENIX Security Symposium* (2010), pp. 453–468. http://www.usenix.org/events/sec10/tech/full_papers/Burnett.pdf.

[12] Caballero, J., Yin, H., Liang, Z., and Song, D. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *14th ACM Conference on Computer and Communications Security* (2007), pp. 317–329. doi:10.1145/1315245.1315286.

[13] Celine, H. Quoted in *Leviathan*, Robert Shea and Robert Anton Wilson, Dell Publishing, 1975.

[14] claffy, k., Andersen, D., and Hick, P. The CAIDA Anonymized Internet Traces — Equinix, Chicago, 17 Feb 2011. Data set, 2011. http://www.caida.org/data/passive/passive_2011_dataset.xml.

[15] Cui, W., Kannan, J., and Wang, H. J. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of the 16th USENIX Security Symposium* (2007), pp. 199–212. http://www.usenix.org/events/sec07/tech/cui.html.

[16] Deibert, R. J., Palfrey, J., Rohozinski, R., and Zittrain, J. Global Internet Filtering Map and Regional Summaries, 2011. http://map.opennet.net/.

[17] Dierks, T., and Allen, C. The TLS Protocol, Version 1.0. RFC 2246, 1999. http://tools.ietf.org/html/rfc2246.

[18] Diffie, W., Oorschot, P. C., and Wiener, M. J. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography 2* (1992), 107–125. http://sites.google.com/site/michaeljameswiener/STS.pdf.

[19] Dingledine, R. Behavior for bridge users, bridge relays, and bridge authorities. Tor Proposal #125, 2007. https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/125-bridges.txt.

[20] Dingledine, R. Iran blocks Tor; Tor releases same-day fix. Tor Project official blog, 2011. https://blog.torproject.org/blog/iran-blocks-tor-tor-releases-same-day-fix.

[21] Dingledine, R. Obfsproxy: the next step in the censorship arms race. Tor Project official blog, 2012. https://blog.torproject.org/blog/obfsproxy-next-step-censorship-arms-race.

[22] Dingledine, R., and Mathewson, N. Tor Protocol Specification. The Tor Project, 2003–2011. https://gitweb.torproject.org/torspec.git/blob/84ec5aca5f5735f445840f6f574842b71365bbde:/tor-spec.txt.

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

[24] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium* (2004), pp. 303–320. http://www.usenix.org/events/sec04/tech/dingledine.html.

[25] Dworkin, M. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D, 2007. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.

[26] Dynamic Internet Technology Inc. DynaWeb. Proxy service, 2002. http://www.dit-inc.us/dynaweb.

[27] Feamster, N., Balazinska, M., Harfst, G., Balakrishnan, H., and Karger, D. Infranet: Circumventing Web Censorship and Surveillance. In *Proceedings of the 11th USENIX Security Symposium* (2002), pp. 247–262. http://www.usenix.org/events/sec02/feamster.html.

[28] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext Transfer Protocol — HTTP/1.1. RFC 2616, 1999. http://tools.ietf.org/html/rfc2616.

[29] Fifield, D., Hardison, N., Ellithorpe, J., Stark, E., Dingledine, R., Porras, P., and Boneh, D. Evading Censorship with Browser-Based Proxies. In *Privacy Enhancing Technologies* (2012), vol. 7384 of *Lecture Notes on Computer Science*, pp. 239–258. https://crypto.stanford.edu/flashproxy/flashproxy.pdf.

[30] Fisk, G., Fisk, M., Papadopoulos, C., and Neil, J. Eliminating Steganography in Internet Traffic with Active Wardens. In *Information Hiding* (2003), vol. 2578 of *Lecture Notes in Computer Science*, pp. 18–35. http://www.woozle.org/~mfisk/papers/ih02.pdf.

[31] Gilmore, J. Quoted in "First Nation in Cyberspace" by Philip Elmer-Dewitt. *TIME Magazine*, December 1993.

[32] Guan, Y., Fu, X., Xuan, D., Shenoy, P. U., Bettati, R., and Zhao, W. NetCamo: Camouflaging Network Traffic for QoS-Guaranteed Mission Critical Applications. *IEEE Transactions on Systems, Man, and Cybernetics 31*, 4 (2001), 253–265. doi:10.1109/3468.935042.

[33] Haselton, B., et al. Peacefire: Open Access for the Net Generation. Web site, 1995–2012. http://www.peacefire.org/info/about-peacefire.shtml.

[34] Hemminger, S. Network Emulation with NetEm. In *linux.conf.au* (2005). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.1687&rep=rep1&type=pdf.

[35] Herrmann, D., Wendolsky, R., and Federrath, H. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-Bayes Classifier. In *Proceedings of the 2009 ACM workshop on Cloud computing security* (2009), pp. 31–42. doi:10.1145/1655008.1655013.

[36] Hintz, A. Fingerprinting websites using traffic analysis. In *Privacy Enhancing Technologies* (2003), vol. 2482 of *Lecture Notes in Computer Science*, pp. 229–233. doi:10.1007/3-540-36467-6_13.

[37] Hopper, N. J., Langford, J., and von Ahn, L. Provably Secure Steganography. In *Advances in Cryptology — CRYPTO* (2002),

vol. 2442 of *Lecture Notes in Computer Science*, pp. 119–123. doi:10.1007/3-540-45708-9_6.

[38] Houmansadr, A., Nguyen, G. T., Caesar, M., and Borisov, N. Cirripede: Circumvention Infrastructure using Router Redirection with Plausible Deniability. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), pp. 187–200. doi:10.1145/2046707.2046730.

[39] Josefsson, S. The Base16, Base32, and Base64 Data Encodings. RFC 4648, 2006. http://tools.ietf.org/html/rfc4648.

[40] Karlin, J., Ellard, D., Jackson, A., Jones, C. E., Lauer, G., Makins, D. P., and Strayer, W. T. Decoy Routing: Toward Unblockable Internet Communication. In *USENIX Workshop on Free and Open Communications on the Internet* (2011). https://db.usenix.org/events/foci11/tech/final_files/Karlin.pdf.

[41] Köpsell, S., and Hillig, U. How to Achieve Blocking Resistance for Existing Systems Enabling Anonymous Web Surfing. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society* (2004), pp. 47–58. https://gnunet.org/sites/default/files/koepsell-wpes2004_0.pdf.

[42] Krawczyk, H. Cryptographic Extraction and Key Derivation: The HKDF Scheme. Cryptology ePrint Archive, Report 2010/264, 2010. http://eprint.iacr.org/2010/264.

[43] Leech, M., Ganis, M., Lee, Y.-D., Kuris, R., Koblas, D., and Jones, L. SOCKS Protocol Version 5. RFC 1928, 1996. http://tools.ietf.org/html/rfc1928.

[44] Lin, Z., Jiang, X., Xu, D., and Zhang, X. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *15th Symposium on Network and Distributed System Security* (2008). http://www.isoc.org/isoc/conferences/ndss/08/papers/14_automatic_protocol_format.pdf.

[45] Lincoln, P., Mason, I., Porras, P., Yegneswaran, V., Weinberg, Z., Massar, J., Simpson, W. A., Vixie, P., and Boneh, D. Bootstrapping Communications into an Anti-Censorship System. In *2nd USENIX Workshop on Free and Open Communications on the Internet* (2012). https://www.usenix.org/conference/foci12/bootstrapping-communications-anti-censorship-system.

[46] Maier, G., Feldmann, A., Paxson, V., and Allman, M. On Dominant Characteristics of Residential Broadband Internet Traffic. In *Proceedings of the Ninth Internet Measurement Conference* (2009), pp. 90–102. http://www.icir.org/vern/papers/imc102-maier.pdf.

[47] Mathewson, N. Pluggable Transports for Circumvention. Tor Proposal #180, 2010–2011. https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/180-pluggable-transport.txt.

[48] Mathewson, N. Tor and Circumvention: Lessons Learned. Invited talk at the 4th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET), 2011. http://www.usenix.org/event/leet11/tech/slides/mathewson.pdf.

[49] McGill, R., Tukey, J. W., and Larsen, W. A. Variations of Box Plots. *The American Statistician 32*, 1 (1978), 12–16. http://www.jstor.org/pss/2683468.

[50] McLachlan, J., and Hopper, N. On the risks of serving whenever you surf: vulnerabilities in Tor's blocking resistance design. In *Proceedings of the 8th ACM workshop on Privacy in the electronic society* (2009), pp. 31–40. doi:10.1145/1655188.1655193.

[51] Moghaddam, H. M., Li, B., Derakhshani, M., and Goldberg, I. SkypeMorph: Protocol Obfuscation for Tor Bridges. Tech. rep., University of Waterloo, 2012. http://cacr.uwaterloo.ca/techreports/2012/cacr2012-08.pdf.

[52] Möller, B. A Public-Key Encryption Scheme with Pseudo-random Ciphertexts. In *Computer Security — ESORICS* (2004), vol. 3193 of *Lecture Notes in Computer Science*, pp. 335–351. http://www.bmoeller.de/pdf/pke-pseudo-esorics2004.pdf.

[53] Murdoch, S. J., and Danezis, G. Low-Cost Traffic Analysis of Tor. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy* (2005), pp. 183–195. doi:10.1109/SP.2005.12.

[54] NIST. Digital Signature Standard. FIPS 186-2, 2000. http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf.

[55] Nottingham, M. Making HTTP Pipelining Usable on the Open Web. Internet-Draft, 2011. http://tools.ietf.org/html/draft-nottingham-http-pipeline.

[56] Ohling, F., Schoen, S., adam or aco, et al. *How to Bypass Internet Censorship*. FLOSS Manuals, 2011. http://en.flossmanuals.net/bypassing-censorship/.

[57] Panchenko, A., Niessen, L., Zinnen, A., and Engel, T. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society* (2011), pp. 103–114. doi:10.1145/2046556.2046570.

[58] Perry, M. Experimental Defense for Website Traffic Fingerprinting. Tor Project official blog, 2011. https://blog.torproject.org/blog/experimental-defense-website-traffic-fingerprinting.

[59] Pfitzmann, A., and Hansen, M. A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management, 2010. v0.34. http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf.

[60] Price, M., Enayat, M., et al. Persian cyberspace report: internet blackouts across Iran. Iran Media Program news bulletin, 2012. http://iranmediaresearch.com/en/blog/101/12/02/09/840.

[61] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. http://www.R-project.org/.

[62] Rogaway, P. Nonce-Based Symmetric Encryption. In *Fast Software Encryption* (2004), vol. 3017 of *Lecture Notes in Computer Science*, pp. 348–359. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.2964&rep=rep1&type=pdf.

[63] Ross, A. Quoted in "Hillary Clinton adviser compares Internet to Che Guevara" by Josh Halliday. *The Guardian*, June 2011. http://www.guardian.co.uk/media/2011/jun/22/hillary-clinton-adviser-alec-ross.

[64] Sun, Q., Simon, D. R., Wang, Y.-M., Russell, W., Padmanabhan, V. N., and Qiu, L. Statistical identification of encrypted Web browsing traffic. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy* (2002), pp. 19–30. doi:10.1109/SECPRI.2002.1004359.

[65] UltraReach Internet Corp. Ultrasurf. Proxy service, 2001. http://www.ultrasurf.us/.

[66] von Ahn, L., and Hopper, N. J. Public-Key Steganography. In *Advances in Cryptology — EUROCRYPT* (2004), vol. 3027 of *Lecture Notes in Computer Science*, pp. 323–341. doi:10.1007/978-3-540-24676-3_20.

[67] Wang, X., Luo, J., Yang, M., and Ling, Z. A potential HTTP-based application-level attack against Tor. *Future Generation Computer Systems 27* (2011), 67–77. doi:10.1016/j.future.2010.04.007.

[68] White, A. M., Matthews, A. R., Snow, K. Z., and Monrose, F. Phonotactic Reconstruction of Encrypted VoIP Conversations: Hookt on Fon-iks. In *IEEE Symposium on Security and Privacy* (2011), pp. 3–18. https://www.cs.unc.edu/~amw/resources/hooktonfoniks.pdf.

[69] Wickham, H. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009. http://had.co.nz/ggplot2/book.

[70] Wilde, T. Knock Knock Knockin' on Bridges' Doors. Tor Project official blog, 2012. https://blog.torproject.org/blog/knock-knock-knockin-bridges-doors.

[71] Wiley, B. Dust: A Blocking-Resistant Internet Transport Protocol, 2010. http://blanu.net/Dust.pdf.

[72] Wustrow, E., Wolchok, S., Goldberg, I., and Halderman, J. A. Telex: Anticensorship in the Network Infrastructure. In *Proceedings of the 20th USENIX Security Symposium* (2011), pp. 459–473. http://www.usenix.org/events/sec11/tech/full_papers/Wustrow.pdf.

[73] Yeo, V. Green Dam enforcement watered down. *ZDNet Asia* (October 2009). http://www.zdnetasia.com/green-dam-enforcement-watered-down-62058509.htm.