Turbo Tunnel, a good way to design censorship circumvention protocols

David Fifield

Abstract

This paper advocates for the use of an interior *session and reliability layer* in censorship circumvention systems—some protocol that provides a reliable stream interface over a possibly unreliable or transient carrier protocol, with sequence numbers, acknowledgements, and retransmission of lost data. The inner session layer enables persistent end-to-end session state that is largely independent of, and survives disruptions in, the outer obfuscation layer by which it is transported.

The idea—which I call Turbo Tunnel—is simple, but has many benefits. Decoupling an abstract session from the specific means of censorship circumvention provides more design flexibility, and in some cases may increase blocking resistance and performance. This work motivates the concept by exploring specific problems that a Turbo Tunnel design can solve, describes the essential components of such a design, and reflects on the experience of implementation in the obfs4, meek, and Snowflake circumvention systems, as well as a new DNS over HTTPS tunnel.

1 A common need

My purpose in writing this paper is to make the case that censorship circumvention systems should incorporate an interior session and reliability layer, independent of the obfuscation layer that provides blocking resistance. Circumvention systems of all kinds stand to benefit from a design that decouples session and obfuscation, whether in increased performance or better resistance to blocking. More importantly, the flexibility afforded by such a design facilitates new and daring circumvention designs that are not easy to realize today.

The need for a session layer separate from obfuscation has been felt, if not stated, by the developers of diverse circumvention systems over many years. In most systems, the obfuscation layer does double duty, being both a vehicle of evasion and a frame for the user's session. This unnecessary fusion of responsibilities causes a number of practical problems, major and minor, which, when not ignored, are typically dealt with uniquely within each system. One of the contributions of this work is to call attention to a common challenge. The solution I propose is to insert a dedicated session/reliability protocol—not necessarily tailored to circumvention—into the middle of the circumvention protocol stack, a design pattern I call Turbo Tunnel. See Figure 1.

user streams	e.g. HTTP, chat, Tor, VPN
session/reliability	e.g. KCP, QUIC, SCTP
obfuscation	e.g. obfs4, meek, Snowflake, FTE

Figure 1: A Turbo Tunnel design uses a session/reliability layer in the middle of the protocol stack, between the user's application streams and the obfuscated network connection responsible for evasion. The session/reliability layer is an internal layer, not exposed to the censor. Using QUIC, for example, does not mean that QUIC UDP packets are observable on the wire; instead, those packets are encapsulated and transmitted inside the blocking-resistant obfuscated tunnel.

To motivate the discussion, let us first outline a few specific problems facing circumvention systems and see how a Turbo Tunnel design can help. These examples will be treated in more detail in Section 3.

Problem: Censors can disrupt obfs4 by terminating TCP connections. obfs4, like many circumvention protocols, relies essentially on an underlying TCP connection [30 §0]. There is no "session" apart from the TCP connection; when it ends, all end-to-end state is lost and a new connection must be started from scratch. This characteristic makes obfs4 vulnerable to attacks on the underlying TCP connection, like the ones that occurred in Iran in 2013, where certain connections were dropped [7] or throttled [2 §4.4] after 60 seconds. Protocols like HTTP that use short connections were hardly affected, while circumvention tunnels had to be constantly restarted. The attack has not recurred, but it remains a potential danger to protocols like obfs4 that rely on a single long-lived TCP connection.

The problem is solved by the introduction of a separate, "virtual" session that outlives any one TCP connection. When one obfs4-protected TCP connection is terminated or throttled, the client may reconnect and resume the session on another. The session protocol provides continuity by retransmitting any information lost during the transition. A TCP connection ceases to be the indispensable backbone of a session, becoming merely a transient and replaceable carrier of bytes.

Problem: meek is half-duplex. meek creates a bidirectional stream by stringing together a sequence of HTTP requests and responses [10 §5]. meek already has the notion of a session: each HTTP request is tagged with a session identifier so that the server can associate requests from the same client. Since requests are independent at the HTTP layer, they must somehow be kept in order. The Tor deployment of meek maintains ordering by permitting only one outstanding request: the client waits for the response to its previous request before making another, giving rise to a "ping-pong" communication pattern in which only one side may send at a time.

The situation is helped by having HTTP requests and responses contain encapsulated packets of a session protocol with sequence numbers. The client may make a request whenever it has something to send. Both peers buffer and reorder incoming packets before passing their contents to an upper layer. The stream multiplexing feature of HTTP/2 means that interleaved requests do not block each other, even if they happen to use the same TCP connection.

Problem: Snowflake can use only one temporary proxy. Snowflake [24] is built on browser-based proxies. Each proxy connection is reliable and in order while it lasts, but a proxy may disappear at any time, leaving the user with a broken session. Because proxies are run by volunteers and assigned to clients randomly, even a working proxy may, by chance, simply be too slow to use tolerably.

An independent session layer solves this problem by enabling a Snowflake client to persist a session through a sequence of disjoint proxy connections—or even to use many proxies at once, splitting traffic across them, as a hedge against one of them being slow. The session layer retransmits whatever is lost when a proxy disappears, and reassembles interleaved packet sequences that arrive over different proxies.

Problem: DNS does not guarantee order nor delivery. DNS over HTTPS [14] has potential as a circumvention tunnel, because HTTPS encryption hides the protocol features that otherwise make DNS tunnels detectable. But any DNS tunnel requires some form of sequencing layer, because the DNS protocol is unreliable. DNS over HTTPS is reliable only up to the first recursive resolver: after that, the recursive queries use plain old UDP-based DNS, which may drop or reorder messages.

Contemporary DNS tunnels use a variety of custom reliability schemes [9]. The Turbo Tunnel approach is to treat reliability as a separable and solved problem. DNS messages can contain encapsulated packets of an existing, tested and debugged, session/reliability protocol. One part of the code can be devoted purely to encoding and decoding DNS messages, while another part builds a reliable channel on top of them, similar to how TCP builds on IP.

The DNS example hints at what is perhaps the greatest advantage of a Turbo Tunnel design: more freedom to the developer. The design of circumvention systems is a creative activity, demanding the flexibility to combine protocols in original and unexpected ways. Separating the concerns of session management and censorship evasion reduces conceptual complexity, empowering the designer to attempt more ambitious ideas. Imagine, for example, a system that permits switching between different obfuscation strategies, without losing end-to-end session state.

2 Implementing a Turbo Tunnel design

The essential component of a Turbo Tunnel design is an internal protocol that takes care of segmenting an outgoing stream into packets, attaching sequence numbers, and deciding when to retransmit unacknowledged data; and on the receiving side, acknowledging received data and reassembling packets back into a stream. The protocol must, of course, be realized in code as some sort of library. The core requirement on the library is that it must permit abstracting its network operations. It must not, for example, insist on sending its own UDP datagrams, but should provide hooks for the calling application to send and receive the library's packets in whatever way is appropriate for the obfuscation layer. The session layer uses the obfuscation layer as a censor-resistant network interface: whenever it would do a socket operation, it calls into the obfuscation layer instead.

In late 2019 I did a small survey [19 #14] to find protocols and libraries suited to implementing Turbo Tunnel designs. I focused on libraries written in Go, because it is commonly used among circumvention developers. The survey turned up two good candidate protocols, KCP [23] and QUIC [16], with implementations in the kcp-go [28] and quic-go [4] libraries. How the protocols work hardly concerns us; what matters is they provide the right interfaces. For our purposes they are basically interchangeable, and in fact most of the implementations in Section 3 were done twice, once with kcp-go and once with quic-go. KCP does not conform to any external standard, but is fairly simple and battle-tested. By itself, KCP provides a single reliable stream, but the related smux [29] library adds support for multiple streams. QUIC is in the process of being standardized [15]. In its usual UDP-encapsulated form, QUIC is the basis of HTTP/3 [3], and already accounts for a notable fraction of web traffic. QUIC is complex, has built-in support for multiple streams, and mandates the use of TLS.

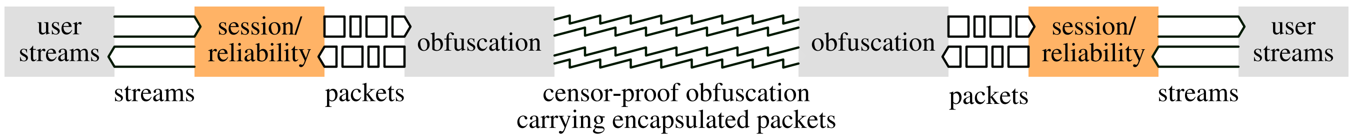

Figure 2: An end-to-end Turbo Tunnel design. The session/reliability layer transforms outgoing streams into packets, and incoming packets back into streams. The obfuscation layer is responsible for encapsulating the session/reliability layer's packets and transporting them in a way that will not be blocked by a censor.

Building a Turbo Tunnel–based circumvention system is a matter of marrying some form of obfuscation with a session/reliability library like kcp-go or quic-go. Essentially, it's a lot of glue code; see Figure 2 for how the pieces fit. The user's streams are fed into the session layer, not directly into the obfuscation layer. The session layer produces discrete packets, which must be *encapsulated* into the obfuscation layer such that they may be recovered at the other end. The details of encapsulation depend on the particulars of the obfuscation channel, which may itself be packet-oriented, stream-oriented, or something else entirely. The encapsulation must also associate with each packet a *session identifier*, a unique value that enables the recipient to distinguish packets belonging to different sessions, analogous to the four-tuple in TCP. The session identifier, being decoupled from any network address, enables roaming by the client, as in Mosh [27 §2.2] and WireGuard [5 §II-A]. The details of roaming depend on the obfuscation, but generally, a server receiving upstream packets tagged with a certain session identifier from a particular network address assumes that downstream packets for that session may be sent to that same network address.

Naively implemented, a Turbo Tunnel design may increase a circumvention system's susceptibility to traffic analysis; that is, detection based on packet sizes and timing. The headers of the session layer impose additional structure on the tunnelled data, and the sending of acknowledgement and keepalive packets may result in distinctive timing patterns. Geddes et al. showed how effective this kind of attack can be, identifying acknowledgement packets through timing and selectively interfering with them to disrupt a circumvention tunnel [13 §4.3]. This style of detection has not, so far, been a serious problem for deployed circumvention systems, but it is wise to allow room in the design for manipulating of traffic features. This means, for example, that the packet encapsulation scheme should allow for padding. It may also mean not sending a complete packet as soon as it is available, but delaying sends and consolidating or splitting packets so that the on-the-wire packet boundaries do not reflect the packet boundaries of the session layer.

The case studies of the next section have source code available that provides examples of implementing Turbo Tunnel. I have also prepared a simplified example that focuses only on the essential elements at https://www.bamsoftware.com/papers/turbotunnel/example/.

3 Case studies

My process for developing the Turbo Tunnel concept has been to implement it in disparate circumvention systems. The process culminated in the creation of a new DNS tunnel built to take advantage of the possibilities of DNS over HTTPS, with high performance compared to other DNS tunnels. The descriptions here are only sketches; see the "history and availability" section for links to source code and more details.

3.1 obfs4

obfs4 [30] is a randomized protocol that works over TCP. There is a one-to-one relationship between user streams and obfuscated TCP connections: the obfs4 session begins and ends where the TCP connection does.

My goal was to make obfs4 resistant to TCP termination attacks. To this end, the client of the Turbo Tunnel version of obfs4 does not open just one TCP connection for each new user session, but rather starts a loop that connects to the obfs4 server and reconnects whenever a connection is terminated for any reason. The outgoing packets produced by the session layer are sent over whatever TCP connection is current (or buffered temporarily during reconnection). Packets are encapsulated by prefixing them with a 16-bit length header and concatenating them on the current TCP connection. The client generates a random session identifier and sends it at the beginning of all its connections. By convention, all packets on a single connection share the same session identifier.

The obfs4 server runs a typical TCP accept loop as always, but instead of piping incoming connections directly to an upstream server, it decapsulates the contained packets and feeds them into a single, global instance of the session protocol. The session protocol produces "new session" and "new stream" events, which are what drive upstream forwarding. The obfs4 server makes no distinction between its many incoming TCP connections; each is an interchangeable conduit for exchanging packets. When the server needs to send a packet for a given session identifier, it sends the packet on the TCP connection from which it most recently received a packet tagged with that session identifier.

Testing through a proxy that terminates connections after 20 seconds showed that a session could persist, despite constant disconnections.

protocol	time
TCP-encapsulated QUIC	10.6 s
traditional meek	23.3 s
meek with encapsulated QUIC	34.9 s

Table 1: Time for simultaneous upload and download of a 10 MB file over different varieties of meek, with TCP-encapsulated QUIC for comparison.

3.2 meek

meek builds a tunnel over a chain of HTTP requests and responses [10 §5]. As currently deployed in Tor, meek transmits unstructured chunks of data in each request or response body, each endpoint simply concatenating the chunks it receives. Because there is no framing to enforce the order of chunks, meek uses a half-duplex communication style, with the client and server taking turns sending data.

The meek Turbo Tunnel integration permits the client to send (i.e., make an HTTP request) any time it has data available, without waiting for the response to its previous request. When the client has a packet to send, it encapsulates the packet into an HTTP request body, along with any other packets that are immediately available. The HTTP request also contains the client's session identifier, which applies to all packets in the request. The server, on receiving an HTTP request, decapsulates all the packets it contains and feeds them into its own session layer, which, as in obfs4, creates the virtual network events to drive upstream connections. The logic for sending downstream data is simple. The server maintains a queue of outgoing packets for each session identifier. When an HTTP request arrives bearing a certain session identifier, the server is entitled to send data addressed to that session identifier in the corresponding HTTP response.

The meek implementation was a success as far as permitting the client to send data at any time, but disappointingly decreased performance in a test of bulk upload and download. See Table 1. The cause of the performance loss is uncertain, but informal experiments show that the performance is sensitive to changes in parameters such as the maximum HTTP body size and the number of request threads.

3.3 Snowflake

Snowflake [24] circumvents address-based blocking using temporary volunteer proxies. Proxies are not expected to remain constantly online. Until recently, there was no way to bridge a session from one proxy another. If your proxy disappeared while you were using it, the session would die, and you would have to restart Snowflake.

The changes required for Turbo Tunnel in Snowflake were similar to what was required in obfs4. The Snowflake client runs a loop of requesting a temporary proxy from the central Snowflake broker, exchanging packets through it until it stops working, then requesting a new proxy. As in obfs4, each new proxy connection begins with a session identifier; and as in meek, for each session identifier the server maintains a queue of outgoing packets. If two or more proxy connections have the same session identifier, they all draw simultaneously from the same queue of outgoing packets on the server. This opens the door to using more than one Snowflake proxy at a time, though this feature is not yet implemented.

The Snowflake implementation has graduated from prototype status and is now deployed to users in the alpha release of Tor Browser. The Tor anti-censorship team evaluated Turbo Tunnel in Snowflake using both kcp-go and quic-go. Both worked well, but the team decided to deploy the kcp-go version, based on considerations of API stability and number of dependencies.

3.4 DNS over HTTPS tunnel

The last Turbo Tunnel experiment is not a modification of an existing system, but an entirely new piece of software, a DNS tunnel called dnstt [8] that can use DNS over HTTPS [14]. DNS tunnels are generally disdained for censorship circumvention because they are considered easily detectable: they generate unusual DNS messages, and each message must contain the domain name of the tunnel server in plaintext. But DNS over HTTPS changes everything,everything, because its messages are encrypted.

DNS is a query–response protocol not unlike HTTP, so the architecture of dnstt is like that of meek. Upstream packets are encoded as DNS names, and downstream packets are contained in TXT responses. Because payload space is tightly constrained in DNS queries (only about 140 bytes are available [9]), the dnstt client does not try to bundle more than one packet per query. Responses allow about 900 bytes of payload, so the server tries to bundle multiple outgoing packets if possible. DNS over HTTPS uses the same message format as UDP-based DNS, but the messages are sent in HTTPS bodies rather than in UDP datagrams. dnstt uses KCP. QUIC is not directly usable because it can require sending packets of up to 1200 bytes [15 §8.1], too large to fit in a DNS message.

Neither the ordering nor the delivery of DNS messages is guaranteed, in either direction. The session protocol in dnstt guarantees that lost packets will be retransmitted and packets put in order before reassembly. Delegating these responsibilities to a dedicated session protocol simplifies the design of the system and permits higher performance. It depends on the resolver, but dnstt achieved download speeds of 130 KB/s using the Google and Cloudflare DNS over HTTPS resolvers, and 30 KB/s using the Quad9. For comparison, iodine [6], the best-known classical DNS tunnel, did not exceed 2 KB/s of download with the UDP resolvers of the same operators.

4 Toward a reusable library?

I have resisted positioning Turbo Tunnel as an importable library, preferring to present it as a general design pattern or a way to think about circumvention protocols. This was to avoid the risk of settling on a programming interface before first understanding all the requirements. The integrations of Section 3 each were each done as if starting from scratch, sometimes borrowing previously written code but not making use of a shared Turbo Tunnel module. With the experience of having implemented the same idea several times, I am skeptical of whether a truly modular "libturbotunnel" is possible. Circumvention designs often demand breaking abstractions and accessing low-level protocol details. Some aspects of the interaction between the session and obfuscation layers, like the encoding of session identifiers into protocol messages, defy easy factorization into a library.

Nevertheless, a few common patterns have emerged that are amenable to modularization and may form the basis of a reusable library. The two main abstractions that have proved useful in every integration are (to use their dnstt names) QueuePacketConn and RemoteMap. QueuePacketConn is an adapter that transforms the "push" interface of kcp-go and quic-go into a "pull" interface. The WriteTo method of QueuePacketConn stores outgoing packets in a queue, so that the obfuscation layer may process them at its own pace. (Perhaps batching several packets into one send operation, for example.) The ReadFrom method of QueuePacketConn draws from a queue of incoming packets, which is replenished by the obfuscation layer as encapsulated packets arrive. RemoteMap manages a mapping of session identifiers to outgoing queues. It is used in servers to buffer outgoing packets for each ongoing session until there is an opportunity to send them. (For instance, the meek server must hold onto packets until it gets an HTTP request to which it may respond, and the dnstt server must similarly wait for a DNS query.)

kcp-go and quic-go are adequate for the task of implementing a Turbo Tunnel design, but they require a fair amount of adaptation to make them work in the way required. A session protocol designed for embedding in a circumvention system would work somewhat differently. Here is a rough list of features I found myself wishing for while working on implementations:

- A "pull" interface, not a "push" interface. Instead of calling a WriteTo callback whenever it wants to send a packet, the session library could buffer the stream of outgoing data, only breaking it up on demand, when the calling code requests a packet. The session library would still decide what actually goes in each packet—how to set the acknowledgement field, for example. It would be the caller's responsibility to poll the session library frequently enough to put packets on the network in a timely manner. Such a "pull" interface would remove the need for the QueuePacketConn adapter.

- A variable maximum size per packet. kcp-go permits setting a global maximum packet size, which is handy for limited-space contexts like DNS messages. It would be even more useful if the maximum size were not global, but could be specified for each packet as it is requested. A good interface would be "give me a packet of at most *n* bytes, or, if none is available, return immediately with nothing." A motivating case for this feature is DNS, in which different resolvers may have different packet length limits. The best one can do with a global maximum packet size is set it conservatively to a low value that is appropriate for all resolvers. An adaptive limit would allow more efficient use of space.

- No built-in cryptography. End-to-end encryption and authentication are, generally speaking, good features to have in a session protocol, but the cryptographic facilities of KCP and QUIC are not a good match for the Turbo Tunnel model. kcp-go supports an optional layer of symmetric-key encryption, which is unfortunately not useful in the common circumvention setting of a single proxy server accessed by mutually untrusting clients: they all know each other's key. QUIC mandates TLS for every connection, which is burdensome to set up and provide a user interface for, which is needless when the user-level protocol is something like Tor that provides its own end-to-end security.

- Few dependencies. Every added dependency is a burden on maintenance. The deployment of Snowflake in Tor Browser patches out unused cryptographic and error-correction code from kcp-go, solely to eliminate dependencies and ease maintenance.

5 Related work

Session-like layers have appeared many times in circumvention systems, usually out of necessity in those that are not built on a reliable channel like TCP. Code Talker Tunnel builds a reliable channel atop UDP by including sequence and acknowledgement numbers [18 §6.2]. OSS similarly embeds sequence and acknowledgement numbers into HTTP URLs [11 §4]. The StegoTorus chopper breaks a stream into packets that may be sent over disparate steganographic channels and arrive out of order [26 §3]; however each channel must itself be reliable, as the chopper does not do retransmission. Conflux proposes to improve the performance of Tor by splitting traffic across multiple simultaneous circuits; circuits are associated with a session identifier, and Tor cells contain a sequence number to permit reassembly at the exit router [1 §3]. The Camouflage system splits traffic across multiple cover channels: different streams may be assigned to different cover channels, but each stream can use only one cover channel at a time [31 §3]. TapDance prepends a session identifier to each covert flow [25 §3.2] that enables a central proxy to concatenate a sequence of short-lived flows into one long-lived session.

Some circumvention systems support tunnelling over QUIC, with optional obfuscation. Psiphon can run QUIC packets through a stream cipher before sending, so that they are not identifiable as QUIC [22]. V2Ray can transform QUIC packets to resemble other UDP-based protocols, like SRTP and DTLS [21]. These applications of QUIC may be viewed as limited implementations of Turbo Tunnel, where the session/reliability layer is QUIC, and the obfuscation layer is lightweight packet-by-packet transformation.

MASQUE [17] is a proposal to colocate proxy servers with web servers over HTTP/2 (TLS/TCP) or HTTP/3 (QUIC/UDP), such that proxy traffic looks like ordinary web traffic. Despite the use of QUIC, MASQUE is not really an example of the Turbo Tunnel idea—the key difference is that it puts QUIC on the outside of the protocol stack, not the inside. To put it in terms of this paper, MASQUE uses QUIC as an obfuscation layer, not a session/reliability layer. However, thanks to the fact that QUIC works well for both purposes, MASQUE could be adapted into a Turbo Tunnel design in two ways. First, one could run some other session protocol through the MASQUE tunnel, treating MASQUE as just another obfuscated proxy. Alternatively, one could encapsulate MASQUE's QUIC packets into some other obfuscation layer (obfs4, say, or even another instance of MASQUE) and use MASQUE as an inner session layer.

History and availability

Resources related to this paper, including a worked example of converting a client–server system to a Turbo Tunnel design, are available at https://www.bamsoftware.com/papers/turbotunnel/.

The Turbo Tunnel idea was developed in a series of posts to the Net4People BBS circumvention discussion forum [19]:

#9	Aug. 2019	Manifesto
#14	Oct. 2019	Protocol evaluation and obfs4
#21	Dec. 2019	meek
#30	Apr. 2020	DNS over HTTPS tunnel
#35	May 2020	Snowflake

obfs4. The changes to obfs4 remain in private branches.
https://gitlab.torproject.org/dcf/obfs4/tree/reconnecting-kcp
https://gitlab.torproject.org/dcf/obfs4/tree/reconnecting-quic

meek. The changes to meek remain in private branches.
https://gitweb.torproject.org/pluggable-transports/meek.git/log/?h=turbotunnel-kcp
https://gitweb.torproject.org/pluggable-transports/meek.git/log/?h=turbotunnel-quic

Snowflake. Turbo Tunnel–enabled Snowflake is part of the alpha release of Tor Browser since version 9.5a13 for desktop and 10.0a1 for Android. The Turbo Tunnel code has been merged into the main development branch.
https://www.torproject.org/download/alpha/
https://gitweb.torproject.org/pluggable-transports/snowflake.git/

dnstt. The home page has downloads and documentation.
https://www.bamsoftware.com/software/dnstt/

Acknowledgements

Much of my Turbo Tunnel research and development was conducted under a contract with Xiao Qiang and the Counter-Power Lab at the University of California, Berkeley. I am grateful for communication with Cecylia Bocovich, Arlo Breault, Roger Dingledine, Vinicius Fortuna, Sergey Frolov, Ian Goldberg, Rod Hynes, Trevor Perrin, David Schinazi, Student Main, Loup Vaillant-David, ValdikSS, Zack Weinberg, and Eric Wustrow. This paper was shepherded for FOCI by Michalis Polychronakis. My implementations have benefited from the use of several free software packages, notably kcp-go [28] and smux [29] by xtaci; quic-go [4] by Lucas Clemente, Marten Seemann, and others; and the Flynn implementation [12] of the Noise protocol framework [20].

References

[1] Mashael Alsabah, Kevin Bauer, Tariq Elahi, and Ian Goldberg. The path less travelled: Overcoming Tor's bottlenecks with traffic splitting. In *Privacy Enhancing Technologies Symposium*. Springer, 2013. https://www.cypherpunks.ca/~iang/pubs/conflux-pets.pdf.

[2] Simurgh Aryan, Homa Aryan, and J. Alex Halderman. Internet censorship in Iran: A first look. In *Free and Open Communications on the Internet*. USENIX, 2013. https://www.usenix.org/conference/foci13/workshop-program/presentation/aryan.

[3] Mike Bishop. Hypertext Transfer Protocol version 3 (HTTP/3). Internet-Draft draft-ietf-quic-http-29, June 2020. https://tools.ietf.org/html/draft-ietf-quic-http-29.

[4] Lucas Clemente, Marten Seemann, et al. quic-go, July 2020. https://github.com/lucas-clemente/quic-go.

[5] Jason A. Donenfeld. WireGuard: Next generation kernel network tunnel. In *Network and Distributed System Security*. The Internet Society, 2017. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/wireguard-next-generation-kernel-network-tunnel/.

[6] Erik Ekman, Bjorn Andersson, et al. iodine, June 2014. https://code.kryo.se/iodine/.

[7] Nima Fatemi. Iran. tor-dev mailing list, May 2013. https://lists.torproject.org/pipermail/tor-dev/2013-May/004787.html.

[8] David Fifield. dnstt, May 2020. https://www.bamsoftware.com/software/dnstt/.

[9] David Fifield. Survey of techniques to encode data in DNS messages, July 2020. https://www.bamsoftware.com/software/dnstt/survey.html.

[10] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-resistant communication through domain fronting. *Privacy Enhancing Technologies*, 2015(2), 2015. https://www.bamsoftware.com/papers/fronting/.

[11] David Fifield, Gabi Nakibly, and Dan Boneh. OSS: Using online scanning services for censorship circumvention. In *Privacy Enhancing Technologies Symposium*. Springer, 2013. https://www.bamsoftware.com/papers/oss.pdf.

[12] Flynn. Go implementation of the Noise protocol framework, March 2018. https://github.com/flynn/noise.

[13] John Geddes, Max Schuchard, and Nicholas Hopper. Cover your ACKs: Pitfalls of covert channel censorship circumvention. In *Computer and Communications Security*. ACM, 2013. https://www-users.cs.umn.edu/~hopper/ccs13-cya.pdf.

[14] Paul Hoffman and Patrick McManus. DNS queries over HTTPS (DoH). RFC 8484, October 2018. https://tools.ietf.org/html/rfc8484.

[15] Jana Iyengar and Martin Thomson. QUIC: A UDP-based multiplexed and secure transport. Internet-Draft draft-ietf-quic-transport-27, February 2020. https://tools.ietf.org/html/draft-ietf-quic-transport-27.

[16] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC transport protocol: Design and Internet-scale deployment. In *SIGCOMM*. ACM, 2017. https://dl.acm.org/doi/10.1145/3098822.3098842.

[17] Multiplexed Application Substrate over QUIC Encryption (MASQUE) working group, February 2020. https://datatracker.ietf.org/wg/masque/about/.

[18] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. SkypeMorph: Protocol obfuscation for Tor bridges. In *Computer and Communications Security*. ACM, 2012. https://www.cypherpunks.ca/~iang/pubs/skypemorph-ccs.pdf.

[19] Net4People BBS. https://github.com/net4people/bbs.

[20] Trevor Perrin. The Noise protocol framework, revision 34, July 2018. https://noiseprotocol.org/noise.html.

[21] Project V. QUIC, November 2018. https://www.v2fly.org/en/configuration/transport/quic.html.

[22] Psiphon. ObfuscatedPacketConn, April 2020. https://github.com/Psiphon-Labs/psiphon-tunnel-core/blob/v2.0.11/psiphon/common/quic/obfuscator.go.

[23] skywind3000. KCP - a fast and reliable ARQ protocol, January 2020. https://github.com/skywind3000/kcp/blob/1.7/README.en.md.

[24] Snowflake. https://snowflake.torproject.org/.

[25] Benjamin VanderSloot, Sergey Frolov, Jack Wampler, Sze Chuen Tan, Irv Simpson, Michalis Kallitsis, J. Alex Halderman, Nikita Borisov, and Eric Wustrow. Running refraction networking for real. *Privacy Enhancing Technologies*, 2020(3), 2020. https://petsymposium.org/2020/files/papers/issue4/popets-2020-0073.pdf.

[26] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. StegoTorus: A camouflage proxy for the Tor anonymity system. In *Computer and Communications Security*. ACM, 2012. https://www.frankwang.org/files/papers/ccs2012.pdf.

[27] Keith Winstein and Hari Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *Annual Technical Conference*. USENIX, 2012. https://www.usenix.org/conference/atc12/technical-sessions/presentation/winstein.

[28] xtaci. kcp-go, July 2020. https://github.com/xtaci/kcp-go.

[29] xtaci. smux, July 2020. https://github.com/xtaci/smux.

[30] Yawning Angel and Philipp Winter. obfs4 (the obfourscator). https://gitlab.com/yawning/obfs4/-/blob/obfs4proxy-0.0.11/doc/obfs4-spec.txt.

[31] Apostolis Zarras. Leveraging Internet services to evade censorship. In *Information Security Conference*. Springer, 2016. https://dke.maastrichtuniversity.nl/zarras/files/Camouflage.pdf.