Turbo Tunnel, a good way to design censorship circumvention protocols

David Fifield

Draft of July 21, 2020

Abstract

This is something of a position paper. In it, I advocate for the use of an interior *session and reliability protocol* in circumvention protocols. By this, I mean an abstraction that overlays a session and reliable stream interface atop some possibly unreliable or transient carrier—something that has sequence numbers, acknowledgements, and retransmission of lost data. This inner session layer provides persistent, end-to-end session state that is persistent and largely independent of the outer obfuscation layer by which it is transported.

The suggestion is simple but its consequences are far-reaching. Decoupling the user's circumvention session from any single obfuscated network connection offers advantages in modularity, blocking resistance, and even performance. We demonstrate the potential of the idea by implementing it in circumvention systems: in obfs4, making it resistant to TCP termination attacks; in meek, permitting full-duplex communication; in Snowflake, allowing sessions to migrate across temporary proxies; and in a new kind of DNS tunnel, showing how a reliable proxy connection can operate over a lossy and hard-to-use channel.

Turbo Tunnel is my name for this idea of embedding a session layer in a circumvention protocol, intended to make the concept easier to talk about. It is a design pattern, not a specific protocol or library. This paper describes my experience in implementing Turbo Tunnel designs, how it was instantiated using specific libraries and protocols, and what benefits were realized.

1 A common need

The need for a session layer separate from obfuscation has long been felt, if not consciously stated, in circumvention systems of all descriptions. In many systems, the obfuscation layer effectively does double duty, being both an instrument of evasion and a frame for the user's session. This unnecessary fusion of responsibilities causes a number of practical problems that are usually either ignored, or dealt with separately in each system as a special case. One of the contributions of the present work is to recognize a common pattern and begin to treat it systematically.

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

Figure 1: A Turbo Tunnel design inserts a session/reliability layer into the circumvention protocol stack, between the user's application-layer streams and the obfuscated network connection that performs evasion. The session/reliability layer is an internal layer, never exposed to the censor. Using QUIC, for example, does *not* mean that QUIC UDP packets are sent on the wire; instead, those packets are encapsulated and sent inside the obfuscated tunnel.

We will start by outlining a few concrete problems with existing and proposed circumvention systems, and how they are solved by a Turbo Tunnel design.

Problem: Censors can disrupt obfs4 by terminating long-lived TCP connections. obfs4, like many circumvention protocols, relies essentially on an underlying TCP connection [26 §0]. There is no separate session state; when the TCP connection ends, all end-to-end state is lost and the session must be restarted from scratch. This makes the protocol vulnerable to attacks on the underlying TCP connection. Just such an attack—termination [5] or throttling [2 §4.4] of connections after 60 seconds—took place in Iran in 2013, forcing users to constantly restart their circumvention tunnels, while leaving protocols that used shorter connections unaffected.

This problem is solved by the introduction of a separate, "virtual" session that outlives any single TCP connection. When one TCP connection is terminated or throttled, the client software may reconnect and resume the session on

a new connection, without interrupting ongoing downloads. The reliability layer takes care of retransmitting any information lost during the transition. A TCP connection ceases to be the essential backbone of a circumvention session, becoming merely a transient and replaceable carrier of bytes.

Problem: meek is half-duplex, limiting its performance. meek [8] creates a bidirectional flow by stringing together a sequence of HTTP requests and responses. Because each request–response pair is independent of the others, they must be kept in order somehow. The Tor deployment of meek maintains this order by enforcing that only one request may be outstanding at a time [8 §5]: the client must wait for a response before initiating another request. This strategy is effective but limits performance, as the tunneled connection cannot send and receive at the same time, and transfer speeds are limited more by round-trip time than bandwidth.

We can help the situation by having HTTP requests and responses contain encapsulated packets of a session protocol with sequence numbers. A client may send a request whenever it has outgoing data, without waiting for the response to its previous request. Both peers buffer and reorder incoming packets before passing their contents to an upper layer. The stream multiplexing 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, which may be slow or have poor uptime. Snowflake [21] is built on temporary browser-based proxies. Each temporary proxy connection is reliable and in-order while it lasts, but a proxy may go away at any time, leaving the user with a broken session. Because proxies are run by volunteers and are assigned to clients randomly, even a proxy that lasts a long time may, by chance, be simply too slow to use tolerably.

An independent session layer solves this problem by enabling a Snowflake client to keep a session going through a sequence of of disjoint proxy connections, switching from one to another as proxies come and go—or even to use many proxies at the same time, dividing traffic between them, as a hedge against one of them being slow. The session layer is responsible for retransmitting whatever is lost when a proxy goes down, and reassembling the interleaved packet sequences that arrive via different proxies.

Problem: DNS is an unreliable channel. DNS over HTTPS [11] has potential as a circumvention tunnel, since HTTPS encryption hides the protocol features that otherwise make DNS tunnels detectable. But it is not trivial to overlay a tunnel on DNS, because DNS is an unreliable channel. DNS over HTTPS, being built on TCP, is reliable up to the recursive resolver, but the resolver issues its own recursive queries using ordinary UDP-based DNS, which may result in reordered or dropped messages.

The basic reliability problem is solved by having DNS messages contain not just little chunks of data, but encapsulated packets of a session/reliability protocol. Contemporary DNS tunnels use custom retransmission schemes, of varying efficiency. The Turbo Tunnel approach is to treat channel reliability as a separable and solved problem. Instead of inventing a tunnel-specific retransmission protocol, we embed an existing protocol, one that is tested and debugged, and not necessarily even DNS-aware. Let there be one part of the code that handles encoding and decoding of DNS messages, and another part that builds a reliable channel on top of those messages. Let performance bottlenecks be limited to those that are inherent in the use of DNS, and not those that are accidental results of an inefficient reliability scheme.

Problems like those described above are common in circumvention protocols. None of the problems is fatal—clearly, circumvention using today's protocols is possible—but each, in some way, limits performance, blocking resistance, or, perhaps most importantly, flexibility. The design of circumvention protocols is a creative activity, requiring the ability to combine protocols in new and unexpected ways. The continuity of a separate session and reliability layer expands the design space and empowers the designer to attempt more ambitious ideas. For example, a Turbo Tunnel design makes it possible to build a meta-protocol that dynamically switches between different forms of obfuscation, even within the same session, without requiring extensive coordination by the programmer.

2 Requirements for the session layer

The essential component of a Turbo Tunnel design is an internal protocol that takes in a stream and takes care of the details of segmenting the stream into packets, attaching sequence numbers, and deciding when to retransmit unacknowledged data. Whatever the protocol, it must somehow be realized in code, ideally with a programming interface that is convenient to use.

The core requirement on a session protocol library is that it provide an option for abstracting its network operations. It must not insist on sending its own UDP datagrams, say, but should provide hooks for the calling application to send packets produced by the library in whatever way it deems appropriate. These hooks provide the linkage between the session and obfuscation layers of Figure 1. The session layer treats the obfuscation layer as an abstract packet I/O interface, and the obfuscation layer treats the session layer as an abstract producer and consumer of packets. It is worth reiterating that the raw session protocol is never exposed to the censor on the wire; it is always sent under cover of some form of blocking-resistant obfuscation.

Besides the core requirement of abstract networking, a nice-to-have feature in a session protocol is stream multiplexing. Stream multiplexing allows one end-to-end session to contain many distinct streams. With this, there can be many logical streams between the circumvention client and proxy, without requiring many separate instances of the obfuscation layer. An additional minor requirement is that the session protocol library should be well-maintained and written in Go, as Go is currently the most-used programming language among circumvention developers.

In late 2019 I did a small survey [15 #14] of candidate protocols. The survey found two protocols that are well suited to Turbo Tunnel: KCP [20] and QUIC [13], with their implementations kcp-go [24] and quic-go [4]. The differences between the two protocols need not concern us here; it is enough to know that they both provide a reliable stream abstraction over an unreliable packet-based channel. The libraries provide roughly equivalent interfaces, and in fact most of the implementations in Section 3 were done twice, once using kcp-go and once using quic-go, with only minor changes required. KCP does not conform to any stable third-party standard, but is fairly simple internally and battle-tested. KCP itself only provides a single reliable stream abstraction, but the smux [25] library can be used to overlay stream multiplexing onto KCP. Wherever KCP is mentioned in this document, it should be understood as the combination KCP+smux. QUIC is in the process of being standardized by the IETF (currently in Internet-Draft status [12]). QUIC's greatest claim is that it—in its default UDP-encapsulated form—is the basis for HTTP/3 [3], and already accounts for a notable fraction of web traffic. QUIC has built-in stream multiplexing and mandates the use of TLS.

The session protocol is a necessary component of a Turbo Tunnel design, but it is not the whole story. Also needed is a way to transmit the packets of the session layer between a circumvention client and proxy, in a way that the packets will not get blocked by a censor. That is the purpose of the obfuscation and network transport layers, the likes of which are already familiar to circumvention system developers. Generally, the process will involve *encapsulation* of the session-layer packets into some other, obfuscated protocol, which itself may be stream-oriented or packet-oriented. The details of encapsulation depend on the obfuscation layer, and there are additional possible complications such as provision for traffic shaping and padding.

3 Case studies

The need for a session layer decoupled from obfuscation has been present in my mind for a long time, but only in the last year did I begin to develop the idea actively. My process has been to implement Turbo Tunnel features into some existing circumvention systems, all of which work quite differently. 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.

3.1 obfs4

obfs4 [26] is a randomized protocol that works over TCP. There is a one-to-one relationship between client streams and obfuscated TCP connections: the obfs4 session ends when the TCP connection does.

My goal in integrating Turbo Tunnel into obfs4 was to make it resistant to TCP connection termination attacks. At the beginning of a new client session, rather than make a single TCP connection to the obfs4 server, the obfs4 client starts up an abstract, redialing packet-sending interface that connects to the obfs4 server repeatedly in a loop, establishing a new TCP connection whenever one is disconnected for any reason. The obfs4 client establishes a KCP or QUIC session atop the redialing packet interface, buffering packets when a connection is in the process of being established, or sending them immediately if a connection already exists. This KCP or QUIC session constitutes the "virtual" session mentioned in Section 1, independent of any single TCP connection. Session-layer packets are encapsulated simply by prefixing them by a 16-bit length header and concatenating them into the current TCP stream.

The obfs4 server runs the typical TCP accept loop as before, but instead of directly piping incoming connections to an upstream server, it decapsulates the packets contained within each TCP connection, and feeds them to a single, global instance of KCP or QUIC. The global KCP or QUIC instance produces "new session" and "new stream" events, and it is these virtual events that drive the forwarding to the upstream server. The obfs4 server makes no distinction between its many incoming TCP connections; each is just an interchangeable conduit for exchanging packets. Each incoming packet is associated with a *session identifier*, which is a random integer generated by the obfs4 client. When the obfs4 server has to send a packet to a certain client, identified by session identifier, it sends the packet using the TCP connection from which it most recently received a packet tagged with that session identifier.

A test through a proxy configured to terminate TCP connections after 20 seconds showed that an obfs4 session was able to persist over a sequence of TCP connections. The session layer takes care of retransmitting whatever packets were lost during each unclean TCP shutdown.

3.2 meek

meek [8], as currently deployed, transmits unstructured chunks of an underlying stream, which are simply concatenated at each end. Because there is no additional framing, it is important to keep the chunks in order. meek does this

protocol	time
direct QUIC UDP	3.7 s
TCP-encapsulated QUIC	10.6 s
traditional meek	23.3 s
meek with encapsulated QUIC	34.9 s

Table 1: Time for combined upload and download of a 10 MB file.

by enforcing a "ping-pong" communication pattern, allowing only one outstanding HTTP request at a time [8 §5].

The goal of integrating Turbo Tunnel into meek was to permit the client to send whenever it had data available, without waiting for a response to the most recent HTTP request. As with obfs4, the key task is building the adapter between the KCP or QUIC engine and the network. When the client has a packet to send, it encapsulates the packet into an HTTP body, along with any other packets that are immediately available. The encapsulation encoding is more sophisticated than it was in the obfs4 prototype, permitting arbitrary padding between packets. Each HTTP request is prefixed with a session identifier, which applies in common to all the packets in the request. The server, on receiving an HTTP request, decapsulates all the contained packets and feeds them to a global KCP or QUIC engine, which as in the obfs4 case takes responsibility for creating the virtual network events that drive upstream connections. The meek server is limited in that it cannot send data to the client whenever it wants; it must wait for an HTTP request to respond to. The logic for sending downstream data is simple. The server maintains a queue of outgoing packets, a separate queue for each known session identifier. When an HTTP request arrives bearing a certain session identifier, the server has license to include downstream data from that session identifier's queue in the corresponding HTTP response.

The meek implementation of Turbo Tunnel 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 of 10 MB. See Table 1. As far as I can tell, the cause may be a bad interaction between QUIC's congestion control and the bundling of requests done by meek, but I have not done enough tests to be sure.

3.3 Snowflake

Snowflake [21] uses a pool of temporary volunteer proxies. Proxies are not expected to remain stable over time. Until recently, there was no way to bridge a session across different proxies. If you were unlucky enough to have your proxy disappear while you were using it, the session would just die, and you would have to restart the Snowflake software.

The changes required were not dissimilar from those required for obfs4. Each temporary proxy connection is reliable and in order while it lasts, just like a TCP connection. The Snowflake client runs a loop of requesting a new temporary proxy from the central Snowflake broker, and sending packets through it while it lasts, requesting a new proxy if the current one goes away. The first thing sent on a fresh proxy connection is the random session identifier, which applies to all packets sent on that connection. The server works as in the meek implementation, feeding incoming packets into the session layer. If there are more than one ongoing proxy connection associated with the same session identifier, all those outgoing connections pull equally from the same queue of outgoing packets. This opens the door to supporting multiple simultaneous proxies per client, though we do not make use of this capability. One nice feature of the design is that while the Snowflake client and server have to upgrade in order to support Turbo Tunnel features, the temporary proxies do not need to change. They remain simple dumb pipes.

The Snowflake implementation graduated from prototype status and is now deployed to users of the alpha release of Tor Browser. I and the Tor anti-censorship team prototyped Turbo Tunnel in Snowflake using both kcp-go and quic-go. Although both worked well, we decided to use kcp-gofor the deployment, because its API had been more stable, it had fewer dependencies, and it was not coupled to a specific version of the Go standard library.

3.4 DNS-over-HTTPS tunnel

After modifying existing circumvention systems, it was time to try something new, DNS tunnels have a long history. The main idea is that a recursive DNS resolver acts like a proxy: it receives a packet from one place and forwards it somewhere else, then returns the response. DNS tunnels are generally disdained for censorship circumvention because they are easily detectable, for the fact that they tend to generate unusual DNS messages, and that they must contain the domain name of the tunnel server in plaintext. But that changes with new, encrypted forms of DNS, like DNS over HTTPS [11]. I wrote a new DNS tunnel called dnstt [7] that is built on Turbo Tunnel principles and works over DNS over HTTPS.

DNS is a query–response protocol not unlike HTTPS, so the client and server resemble those of meek. Upstream packets are creatively encoded as DNS names, and downstream packets are encoded into TXT responses. Unlike in meek, the dnstt client does not try to encapsulate more than one packet into a DNS query, because payload space is tightly constrained (only about 125 bytes are available per query). Responses are a little less constrained, allowing about 900 bytes, so the server tries to bundle multiple packets if possible. In DNS over HTTPS, each DNS query is serialized to a sequence of bytes, as if it were going to be sent as a UDP payload, but the bytes are instead sent in an HTTP POST body. The response similarly comes back in an HTTP response body.

Some sort of reliability layer is necessary in any DNS tunnel. Even DNS over HTTPS is only reliable TLS up to the

resolver	tunnel	transport	download rate
none	dnstt	UDP	187.1 KB/s
Cloudflare	dnstt	UDP	156.4 KB/s
Google	dnstt	HTTPS	135.1 KB/s
Cloudflare	dnstt	HTTPS	133.5 KB/s
Comcast	dnstt	HTTPS	66.3 KB/s
Quad9	dnstt	UDP	58.9 KB/s
Google	dnstt	UDP	43.1 KB/s
PowerDNS	dnstt	HTTPS	38.0 KB/s
Quad9	dnstt	HTTPS	30.9 KB/s
none	iodine	UDP	14.6 KB/s
Google	iodine	UDP	1.8 KB/s
Cloudflare	iodine	UDP	1.4 KB/s
Quad9	iodine	UDP	0.3 KB/s

Table 2: Rate of downloading a 10 MB file through DNS tunnels. "none" for a resolver means queries are sent directly to the tunnel server, with no intermediate recursive resolver.

recursive resolver; the recursively forwarded queries still go out over plain old unreliable UDP. Neither the ordering nor the delivery of DNS messages is guaranteed, in either direction. DNS tunnels have historically used a variety of ad hoc reliability schemes [6]. Delegating that responsibility to a dedicated session protocol not only simplifies the overall design, it permits higher performance. Table 2 shows the download performance of dnstt (over DNS over HTTPS, plus plain UDP for comparison) and iodine, the best-known classical DNS tunnel. Performance depends highly on what resolver is used, but DNS over HTTPS can outperform UDP-based DNS even on the same resolver, and dnstt is faster than iodine in every case.

4 Considerations for encapsulation

A Turbo Tunnel design, if naively applied, has the potential to make covert channels more susceptible to detection by analysis of packet sizes and timing, because the headers of the session layer impose some additional structure on the tunnelled data. While there have not been reports of this style of detection used by censors on a large scale, it is important to leave enough leeway in the design of a circumvention system to permit manipulating traffic features if necessary. That means that your scheme for encapsulating packets should allow for arbitrary padding. It also means you may have to complicate your code by, for example, not sending an encapsulated packet as soon, entire, as it is available, but perhaps delaying or splitting the send, so that the packet boundaries on the wire do not reflect the boundaries of the encapsulated packets. With stream-oriented carriers, it is always possible to delay sends, add padding, and split encapsulated packets. Packet-oriented carriers may not be able to split encapsulated packets, but they can delay large packets, bundle multiple packets into one, and send packets consisting of nothing but padding.

5 Toward a reusable library?

From the beginning, I have resisted positioning Turbo Tunnel as a ready-to-use importable library, instead proposing it as a general design pattern, a way to think about the construction of circumvention protocols. Partly this was to avoid premature generalization, of codifying a programming interface before understanding all the requirements. Indeed, many of the considerations in Section 4 became apparent only after the exercise of implementing the design separately for obfuscation protocols that differ widely in nature.

Accordingly, the integrations of Section 3 are all tailored to the needs of the underlying protocol, sometimes borrowing code from each other but not sharing a single external Turbo Tunnel module. I am skeptical of whether there can be a truly pluggable "libturbotunnel," particularly in a field like censorship circumvention that often demands access to low-level protocol details and breaking abstractions. There are details, like the encoding of session identifiers into protocol messages, that defy easy factorization into a library.

Nevertheless, in the experience of repeated implementation, certain common patterns have emerged that are amenable to modularization and may form the basis of a reusable library. The two main abstractions are QueuePacketConn and RemoteMap, which have proved to be useful in every integration with only minor changes. QueuePacketConn is an adapter transforms the "push" interface provided by kcp-go and quic-go into a "pull" interface. The WriteTo method stores outgoing packets in a queue, from which another part of the code may process them at its own pace. (On the client, perhaps batching several packets into one network operation; on the server, perhaps waiting for an incoming HTTP request or DNS query to respond to.) The ReadFrom method does not touch the network, but only draws from a queue of incoming packets, which is filled by a network-aware part of the code as it received encapsulated packets. QueuePacketConn depends on RemoteMap, which manages a mapping of remote addresses (i.e., session identifiers) to outgoing queues. When a peer needs to send packets to a certain session identifier (for example, when an HTTP request arrives bearing that session identifier and the server needs to fill a response body), it pulls packets belonging to that session by looking up the queue in the map. RemoteMap is the abstraction that enables connection migration and separate simultaneous channels.

kcp-go and quic-go are fully adequate for implementing a Turbo Tunnel design, though, not being originally intended for this purpose, they impose a little bit of added implementation friction. If I were to invent a new protocol for the specific purpose of being an encapsulated session layer, I would design the interface somewhat differently. Here is a rough list of desired features:

- A "pull" interface, not a "push" interface. Rather than calling a WriteTo callback whenever it wants to send a packet, the session library should buffer the stream of outgoing data and only segment it into packets on demand, when calling code requests a packet. The session library would still be responsible for deciding what actually goes in the packet (a data-carrying packet, an empty acknowledgement, or a keepalive, say), and it would be the caller's responsibility to poll the session library frequently enough to ensure that packets hit the network in a somewhat timely manner. This kind of interface would remove the need for the QueuePacketConn adapter.

  If the session library does anything like round-trip time estimation, the send time should be based on the time when the packet is produced for the caller. QueuePacketConn may falsely inflate the round-trip times seen by kcp-go and quic-go, because it buffers packets locally for a time before sending them.

- Variable maximum packet size per pull. kcp-go permits setting a global MTU, a maximum length which no packet will exceed. However, it would be more convenient if the maximum packet size were not a static global parameter, but could be set per pull. A good interface would be "give me a packet of length at most *n* bytes, or, if none is available, return immediately with nothing." An example of where this would be useful is in the DNS tunnel server. Each DNS query may support a different maximum response size. The maximum response size is conveyed with the query, depends on the recursive resolver, and cannot be known in advance. Because the kcp-go MTU is a global setting, the best one can do is set a conservative limit that is likely to be within the maximum size of *every* resolver. A per-packet size limit would allow adapting downstream packet sizes to each query and making better use of the available space.

  Related to this point, the minimum required packet size should be on the small size, at most 100 bytes, say. QUIC requires the client to send a single packet of at least 1200 bytes during the handshake, as a defense against traffic amplification [12 §8.1]. 1200 bytes is too much to fit in a single DNS request, for example, which would require some form of fragmentation beyond QUIC's own reassembly if QUIC were used for that purpose; and in a Turbo Tunnel design, traffic amplification attacks may not apply, as QUIC packets are likely to be encapsulated in some protocol other than UDP.

- No built-in cryptography. Coupling the session layer with a mandatory layer of end-to-end encryption and authentication is not inherently a bad idea: it means that it is safe to transmit even plaintext protocols through the circumvention system while trusting any intermediaries only with covertness, not with data security. But the cryptographic facilities of KCP and QUIC are either unsuited to the task, or awkward to work with. kcp-go supports an optional layer of symmetric-key encryption, which is not useful in the common circumvention setting of a single proxy server shared by mutually untrusting clients: they all know each other's key. QUIC mandates the use of TLS for every connection, which overall is a good thing, but in the circumvention context it can be burdensome to set up. There's no Let's Encrypt for exotic obfuscated circumvention protocols, so one must manually generate keys and certificates, and specially configure the client to pin a server public key, for example. In dnstt, I used kcp-go with its encryption layer disabled. In its place I substituted a Noise protocol [16], which is more secure than what kcp-go provides, and easier to set up than QUIC TLS.

- Few dependencies. Every added dependency is a burden on maintenance. Tor Browser uses a reproducible build system that requires enumerating every dependency of every sub-project and writing a build script for it. The deployment of Snowflake in Turbo Tunnel actually patches out unused cryptographic and error-correction code from kcp-go, solely to eliminate dependencies and ease maintenance.

6 Related work

Session-like layers have appeared many times in circumvention systems, usually out of necessity in systems that are not built on a reliable channel like TCP. We will cite a few examples to highlight the common elements, notably session identifiers and sequence numbers, that inform the Turbo Tunnel idea. Code Talker Tunnel builds a reliable channel atop UDP by attaching sequence and acknowledgement numbers [14 §6.2]. OSS similarly embeds sequence and acknowledgement numbers into HTTP URLs [9 §4], in order to support retransmission of lost data when an HTTP request fails. The StegoTorus chopper breaks a stream into sequence number–tagged packets, which may be sent over disparate steganographic channels and arrive out of order [23 §3]; however each channel must itself be reliable, as StegoTorus does not do retransmission. Conflux proposes to improve the performance of Tor by splitting traffic across multiple simultaneous circuits; to permit reassembly it introduces a new cell type tagged with a session identifier and sequence number [1 §3]. meek includes a session identifier with each HTTP request to allow disambiguating multiple clients in the absence of metadata like the remote IP address, but punts on the issue of sequencing by strictly serializing HTTP requests and responses [8 §5]. The Camouflage system splits traffic across multiple cover channels: different TCP streams may be assigned to different cover channels, but each stream can use only one cover channel at a time [27 §3]. The recent deployment of TapDance prepended

each covert flow with a session identifier [22 §3.2], to enable the central proxy to concatenate a sequence of short-lived flows into a long-lived session.

Some circumvention systems currently support tunneling through QUIC, optionally obfuscated, with UDP as the network transport. Psiphon can run each QUIC packet through a stream cipher before sending [18]. V2Ray can transform each QUIC packet to resemble some other UDP-based protocol, such as SRTP, DTLS, or WeChat video [17]. These uses of QUIC for circumvention may be viewed as somewhat restricted implementation of Turbo Tunnel, with QUIC serving as the session and reliability layer, and lightweight packet-by-packet obfuscation on top.

MASQUE [19] is a proposal to colocate covert proxy servers with existing web servers, over HTTP/2 (TLS/TCP) or HTTP/3 (QUIC/UDP). Despite the use of QUIC, MASQUE is not really an example of the Turbo Tunnel idea. The key difference is that MASQUE puts QUIC on the outside of the protocol stack, not the inside, and makes QUIC itself act as the obfuscation layer, with the goal of blending in with normal web traffic. Alternatively, MASQUE could be regarded as an optimization in which the same protocol (QUIC) works across three layers: session/reliability, obfuscation, and network transport (refer to Figure 1). The observation is that not only is QUIC a convention session protocol, it also makes a good cover protocol, because of its encrypted-by-default nature and its increasing use on the Internet. Collapsing three layers into one avoids the overhead of encapsulating QUIC packets into some other protocol. The efficiency comes at the loss of some flexibility: the obfuscation of MASQUE is not "pluggable"; the only option for obfuscation is as web traffic.

History and availability

The Turbo Tunnel idea was developed in a series of posts to the Net4People BBS circumvention discussion forum [15]:

#9	Aug. 2019	Manifesto.
#14	Oct. 2019	Protocol evaluation and obfs4.
#21	Dec. 2019	meek.
#30	Apr. 2020	DNS 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 a private branch.
https://gitweb.torproject.org/pluggable-transports/meek.git/log/?h=turbotunnel

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

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/

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 discussion with Cecylia Bocovich, Arlo Breault, Roger Dingledine, ValdikSS, Vinicius Fortuna, Student Main, Eric Wustrow, Sergey Frolov, and Loup Vaillant-David. This work was shepherded for FOCI by Michalis Polychronakis. The Turbo Tunnel implementations have benefited from the use of third-party packages: kcp-go [24] and smux [25] by xtaci; quic-go [4] by Lucas Clemente, Marten Seemann, and others; and the Flynn implementation [10] of the Noise protocol framework [16].

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, June 2020. https://github.com/lucas-clemente/quic-go.

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

[6] David Fifield. Survey of techniques to encode data in DNS messages, May 2018. https://trac.torproject.org/projects/tor/wiki/doc/DnsPluggableTransport/Survey.

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

[8] 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/.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

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

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

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

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

[19] David Schinazi. MASQUE obfuscation. Internet-Draft draft-schinazi-masque-obfuscation-02, April 2020. https://tools.ietf.org/html/draft-schinazi-masque-obfuscation-02.

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

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

[22] 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.

[23] 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.

[24] xtaci. kcp-go, April 2020. https://github.com/xtaci/kcp-go.

[25] xtaci. smux, April 2020. https://github.com/xtaci/smux.

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

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