Snowflake, a censorship circumvention system using temporary WebRTC proxies

(Draft October 3, 2023)

Cecylia Bocovich Arlo Breault David Fifield Serene Xiaokang Wang

Authors are listed alphabetically.

Abstract

Snowflake is a system for circumventing Internet censorship. Its blocking resistance comes from the use of numerous, ultralight, temporary proxies ("snowflakes"), which accept traffic from censored clients using peer-to-peer WebRTC protocols and forward it to a centralized bridge, which does the real work of directing traffic to its destination. The temporary proxies are lightweight enough to be implemented in JavaScript, in a web page or browser extension, making them vastly cheaper to set up than a traditional proxy or VPN server. The proxies do not need to have stable network addresses, nor be continually online—even the disappearance of an in-use proxy does not mean the end of a circumvention session, as its client will switch to another on the fly, invisibly to upper network layers.

Snowflake has been deployed with success in Tor Browser and Orbot for several years. It has been significant for circumvention during high-profile network disruptions, including in Russia in 2021 and Iran in 2022. In this paper, we explain the composition of Snowflake's many parts, give a history of deployment and attempts to block it, and reflect on implications for circumvention generally.

1 Introduction

Censorship circumvention systems—systems to enable network communication despite interference by a censor—may be characterized on multiple axes. Some systems imitate a common network protocol; others try not to look like any protocol in particular. Some distribute connections over numerous proxy servers; others concentrate on a single proxy that is, for one reason or another, difficult for a censor to block. What all circumvention systems have in common is that they strive to increase the *cost* to the censor of blocking them—whether that cost be in research and development, human resources, and hardware; or in the inevitable overblocking that results when a censor tries to selectively block some connections but not others. Snowflake, the subject of this paper, is a circumvention system that uses thousands of temporary proxies and makes switching between them easy and fast. On the spectrum of imitation to randomization, Snowflake falls on the side of imitation; on the scale from diffuse to concentrated, it is diffuse. What most sets Snowflake apart is that it pushes the idea of distributed, disposable proxies to an extreme: its proxies can run in a web browser and censored clients communicate with them using WebRTC.

WebRTC is a suite of protocols intended for real-time communication applications in web browsers [1]. Video and voice chat are typical examples of WebRTC applications. Snowflake exchanges WebRTC data formats in the course of establishing a connection, and uses WebRTC protocols for traversal of NAT (network address translation) and communication between clients and proxies. Crucially for Snowflake, WebRTC APIs are available to JavaScript code in web browsers, meaning it is possible to implement a proxy in a web page or browser extension. WebRTC is also usable outside a browser, which is how we implement the Snowflake client program and alternative, command line–based proxies.

As is usual in circumvention research, we assume a threat model in which *clients* reside in a network controlled by a *censor*. The censor has the power to inspect and interfere with traffic that crosses the border of its network; typical real-world censor behaviors include inspecting IP addresses and hostnames, checking packet contents for keywords, blocking IP addresses, and injecting false DNS responses or TCP RST packets. The client wants to communicate with some *destination* outside the censor's network, possibly with the aid of third-party *proxies*. The censor is motivated to block the specific contents of the communication, or even the destination itself. The censor is aware of the possibility of circumvention, and therefore seeks to block not only direct communication, but also indirect communication by way of a proxy or circumvention system. We consider circumvention accomplished when the client can re-

liably reach any proxy, because the proxy, outside the censor's control, can forward the client's communication to any destination. (In Snowflake, we separate the roles of temporary *proxies* and a stable long-term *bridge*, but the idea is the same.) Working in the client's favor is the fact that the censor is presumed to derive benefit from permitting some forms of network access: the censor does not trivially "win" simply by shutting down all communication, but must be selective in its blocking decisions in order to optimize some objective of its own. The art of censorship circumvention is forcing the censor into a dilemma of overblocking or underblocking, by making circumvention traffic difficult to distinguish from traffic that the censor prefers not to block.

Snowflake originates in two earlier projects: flash proxy and uProxy. Flash proxy [9], like Snowflake, used a model of untrusted, temporary JavaScript proxies in web browsers; then, the link between client and proxy was WebSocket rather than WebRTC. (WebSocket still finds use in Snowflake, but on the proxy–bridge link, not the client–proxy link.) Flash proxy was deployed in Tor Browser from 2013 to 2016, but never saw much use, probably because the reliance on WebSocket, which does not have built-in NAT traversal like WebRTC does, required users to perform a cumbersome port forwarding procedure. WebRTC was at the time an emerging technology, and while it had been considered as a future transport protocol for flash proxy, we decided to start Snowflake as an independent project. uProxy [36], in one of its early incarnations, pioneered the use of WebRTC proxies for circumvention. uProxy's proxies were browser-based, but its trust and deployment models were different from flash proxy's and Snowflake's. Each censored client would arrange, out of band, for a personal acquaintance, outside the censor's network, to run a proxy in their web browser. The trust relationship was necessary to prevent misuse, because the browser proxies fetched destination content directly, which means activity by the client would be attributed to the proxy operator. A uProxy proxy was expected to be persistent and always online; clients did not change proxies on the fly. uProxy supported protocol obfuscation: the communications protocol was fundamentally WebRTC, but the contents of packets could be transformed so as not to resemble WebRTC. This obfuscation was possible because uProxy ran as a privileged browser extension with access to real sockets. Since Snowflake uses ordinary unprivileged browser APIs, its WebRTC can only look like WebRTC; on the other hand, because of that, Snowflake proxies are even easier to deploy. Like flash proxy, uProxy was active in the years 2013–2016.

Among existing circumvention systems, the one that is most similar to Snowflake is MassBrowser [25]. It features multiple circumvention techniques, one of which is proxying though volunteer proxies, called buddies. MassBrowser's architecture is similar to Snowflake's: there is a centralized component that coordinates connections between clients and buddies, corresponding to a piece in Snowflake called the broker; buddies play the same role as our proxies. The trust model is intermediate between Snowflake's and uProxy's. Buddies preferentially operate as one-hop proxies, as in uProxy, but are not limited to proxying only for trusted friends. To deter misuse, buddies specify a policy of what categories of content they are willing to proxy. Buddies also support forwarding to a Tor bridge, as in Snowflake, but this option is used only as a last resort, in keeping with MassBrowser's principle of prioritizing blocking resistance and performance over privacy and anonymity. An innovation in MassBrowser not present in Snowflake is client-to-client proxying: clients may act as buddies for other clients, the logic being that what is censored for one client may not be censored for another. The buddy software is a standalone application, not constrained by a web browser environment, and can, like uProxy, use protocol obfuscation on the client–buddy link.

Protozoa [2] and Stegozoa [12] show ways of building a point-to-point covert tunnel over WebRTC, the former by replacing the encrypted parts of encoded media frames with its own ciphertexts, the latter using video steganography. Designs like these might serve as alternatives for the link between client and proxy in Snowflake. Significantly, where Snowflake now uses WebRTC data channels, Protozoa and Stegozoa are built around WebRTC media streams, which may be an advantage in blocking resistance. We will have more to say on this point in Section 3.

It is not our purpose to disproportionately emphasize the limitations of other circumvention systems and the advantages of Snowflake. Circumvention research is a cooperative enterprise, and we recognize and support our colleagues who are pursuing and maintaining their own designs. While challenges remain, today's circumvention systems by and large serve their intended purpose, and are a vital element of day-to-day Internet access for many people. With Snowflake, we have explored a different point in the design space, one with its own advantages and disadvantages. We acknowledge that Snowflake will be a better choice in some censorship environments and worse in others; indeed, one of the ideas we hope to convey is that blocking resistance can be meaningfully understood only in relation to a particular censor and its resources, costs, and motivations. In this paper we present the design of Snowflake, discuss various challenges and considerations, and reflect on over three years of deployment. As of October 2023, Snowflake supports an estimated average 30,000 concurrent users and transfers around 30 TB of circumvention traffic per day.

2 How it works

A Snowflake proxy connection proceeds in three phases. First, there is rendezvous, in which a client indicates its need for circumvention service and is matched with a temporary proxy. Rendezvous is facilitated by a central server called the broker. Then, there is connection establishment, where the client and its assigned proxy connect to each other with WebRTC, using

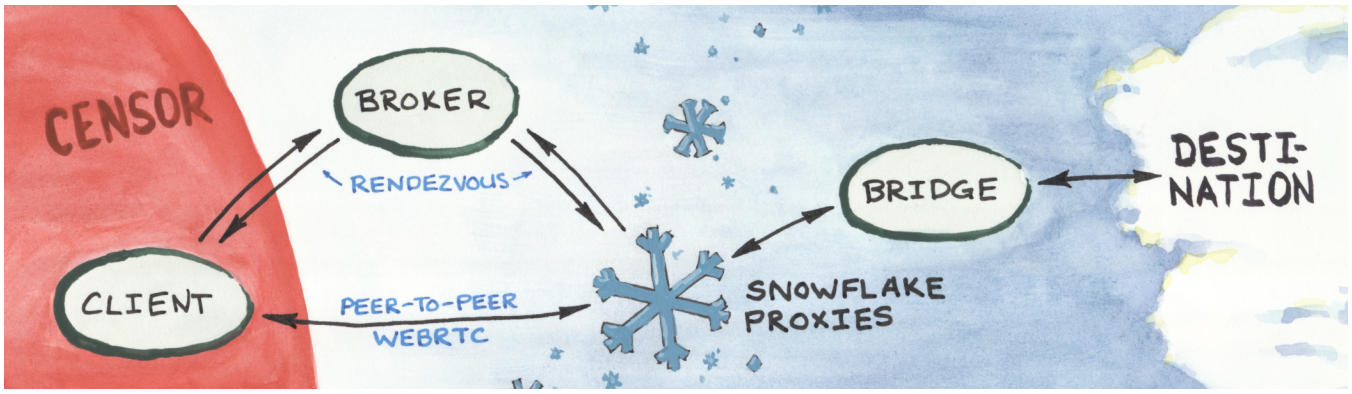

Figure 1: Architecture of Snowflake. The client contacts the broker through a special channel with high blocking resistance. The broker assigns the client a compatible proxy from among those currently polling. The client and proxy connect to one another using WebRTC. The proxy connects to the bridge, then begins copying traffic between the client and the bridge. Session state is established at the client and the bridge, so that it may be resumed on a different proxy if interrupted.

information exchanged during rendezvous. Finally, there is data transfer, where the proxy ferries data between the client and the bridge. The bridge takes responsibility for directing the client's traffic to its eventual destination (in our case, by feeding it into the Tor network). Figure 1 illustrates the process.

These phases repeat as needed, as temporary proxies go offline. A circumvention session is not tied to any single proxy. A client builds a session over a series of proxies, switching to a new one whenever the current one stops working. State variables stored at the client and the bridge ensure the session can pick up where it left off. The change of proxies is invisible to the applications using Snowflake (except, perhaps, for a brief delay while rendezvous takes place again): the Snowflake client presents an abstraction of a single, uninterrupted connection.

It does not avail a censor to block the broker or bridge, because Snowflake clients never contact either directly. Clients reach the broker over an indirect rendezvous channel. Access to the bridge is always mediated by a temporary proxy.

2.1 Rendezvous

A session begins with a client sending a rendezvous message to the broker. There is an ambient population of proxies constantly polling the broker to check for new clients in need of service. The broker matches the client with one of the currently available proxies, subject to considerations such as compatibility of NATs.

The client's rendezvous message is a bundle of data that the broker users to match the client with a proxy, and that the proxy will need in order to make a connection with the client. The essential element is a Session Description Protocol (SDP) *offer* [28], which contains the information necessary for a WebRTC connection, such as the client's external IP addresses and cryptographic details to secure a later key exchange. The broker forwards the client's SDP offer to the proxy, and the proxy sends back an SDP *answer*, containing its share of connection details. The broker forwards the proxy's SDP answer back to the client. The client and the proxy then connect to each other directly. In WebRTC terms, this offer/answer exchange is called "signaling," and the broker here acts as a signaling server. To gather the information necessary to construct an offer or answer, clients and proxies communicate with third-party STUN servers before contacting the broker. We additionally use STUN servers to identify the NAT type of clients. We will say more about how this information is used in Section 2.2. The contacting of STUN servers is a normal and expected part of WebRTC, though there are fingerprinting considerations that we discuss in Section 3.

Communication with the broker uses a "long-polling" model. An example is shown in Figure 2. Proxies poll the broker periodically, by making an HTTPS request to a designated URL path. The broker does not respond immediately to a proxy poll request, but holds the connection idle for a few seconds to see if a client rendezvous message will arrive. If not, the broker sends a response saying "no clients" and the proxy goes to sleep until it is time for its next poll. If a client does arrive, the broker sends its SDP offer to the proxy in the response to the proxy's poll request. As it is now too late for the proxy to send more information in the same HTTPS exchange, the proxy sends its SDP answer back to the broker in a second HTTPS request. All this happens while the client waits for a response to its initial rendezvous message. The broker responds to the client with the proxy's SDP answer, simultaneously sending an acknowledgement to the proxy. At that point rendezvous is finished and the client and the proxy connect to one another.

The client needs to use an indirect channel, resistant to blocking, when communicating with the broker. What is needed, essentially, is a miniature circumvention system to bootstrap into the full system. What makes the rendezvous facet of circumvention different from general circumvention are its different,

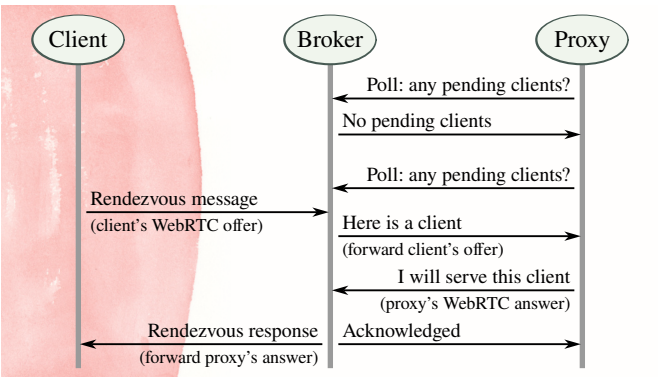

Figure 2: The long-polling communication model of Snowflake rendezvous. Proxies poll periodically to check for new clients. When the broker makes a match, the proxy gets the client's SDP offer, then immediately re-connects to send back its SDP answer. It all happens during one round trip from the client's perspective. Not shown here is the indirect channel used by the client to access the broker through the censor's zone of control (shaded background).

generally more lenient, requirements and assumptions, which permit a larger solution space. Because rendezvous accounts for only a small fraction of total communication volume, and it happens only infrequently, it can afford to use techniques that would be too slow, expensive, or complicated for real-time or bulk data transfer. Another nice feature is that rendezvous is separable and modular: more than one method can be used, and the methods do not necessarily have to bear any relation to the circumvention techniques of the main system. While the assumption of WebRTC permeates Snowflake's design, its rendezvous modules are independent. We currently support two rendezvous methods in Snowflake:

Domain fronting In this method, the client does an HTTPS exchange with the broker by an intermediary web service such as a CDN, taking care to change the externally visible hostname (the TLS Server Name Indication, or SNI) from that of the broker to some other "front domain" [10]. The CDN routes the HTTPS request to the broker according to the HTTP Host header, which remains unmodified under the TLS encryption. The well-known drawback of domain fronting is the high financial cost of CDN bandwidth. Because we use it only for rendezvous, the cost is much more manageable than in a system that uses domain fronting for all its data transfer.

AMP cache AMP is a framework for web pages written in a restricted dialect of HTML. Part of this framework is a free-to-use cache server [27]. The cache fetches origin web pages on demand, which means that it is effectively as a restricted sort of HTTP proxy. If rendezvous messages are encoded to conform to AMP requirements, they can be sent to the broker via the cache server. Rendezvous through the AMP cache is not easily blocked without blocking the cache server as a whole. This rendezvous method still technically requires domain fronting, because the AMP cache protocol would otherwise expose the upstream server's hostname in the TLS SNI, but it enlarges the set of usable intermediary web services and front domains.

Anything that can be persuaded to convey a rendezvous message of about 1500 bytes indirectly to the broker, and return a response of about the same size, might work as a rendezvous module. For example, encrypted DNS (DNS over TLS or DNS over HTTPS) would serve: the client encodes its rendezvous message into a series of DNS queries for hostnames whose authoritative resolver is the broker itself, equipped with a module to reassemble the rendezvous message and send a response in the form of DNS responses. A third-party recursive resolver acts as an intermediary for the broker, while DNS encryption hides the broker's DNS zone from the censor.

Rendezvous is not unique to Snowflake. Other examples of rendezvous in circumvention include the DEFIANCE Rendezvous Protocol [20 §3], the facilitator interaction in flash proxy [9 §3], and the registration proxy in Conjure [13 §4.1]. A key property of the listed systems is that they do not rely on preshared secret information. The client needs only to acquire the necessary software; whatever additional information is required to establish a circumvention session is exchanged dynamically, at runtime. A corollary of the no-secret-information property is that an adversary—the censor—is at no special disadvantage in attacking the system. The censor may download the client software, run it, study its network connections—and the system must maintain its blocking resistance despite this. This stands in contrast to other systems in which, preliminary to making a connection, a client must acquire some secret, such as a password or proxy IP address, through an out-of-band channel presumed to be unavailable to the censor, and the system's blocking resistance depends on keeping that information hidden from the censor. The disadvantage of a separate rendezvous step is that it is one more thing to get right. Not only the main circumvention channel but also the rendezvous must resist blocking: the system is only as strong as the weaker of the two.

2.2 Peer-to-peer connection establishment

Now the client and the proxy connect to each other directly. Even in the absence of censorship, making a direct connection between two Internet peers is not always easy, because of NAT (network address translation) and firewalls. Snowflake clients and proxies alike run in diverse networks that have varying NATs and ingress policies. Fortunately for us, WebRTC is designed with this use case in mind, and has built-in support for traversing NAT, namely ICE (Interactive Connectivity Establishment) [19], a procedure for testing candidate pairs of

peer network addresses to find a pair that works. ICE makes use of STUN (Session Traversal Utilities for NAT) [29] and third-party STUN servers that, among other services, enable a host to learn its own external IP addresses. The first part of ICE has already taken place at the beginning of rendezvous, when the client and proxy contacted STUN servers to gather external address candidates and included them in their respective SDP offer and answer.

There is no guarantee that any two hosts will be able to make a connection using the facilities of STUN alone. Some address mapping and filtering setups are simply incompatible. In the case of an incompatible pairing, ICE would normally fall back to using TURN (Traversal Using Relays around NAT) [31], which is a kind of UDP proxy. Such a fallback would be problematic for Snowflake, because the TURN relays themselves would become a focus of blocking by the censor. But Snowflake has an advantage most WebRTC applications do not. Most WebRTC applications want to connect *a particular* pair of peers, whereas we are satisfied when a client can connect to *any* proxy. Snowflake clients and proxies self-assess their NAT type and report it in interactions with the broker. The broker takes NAT compatibility into account when matching and avoids cases that would require a fallback to TURN.

Two factors are relevant to a Snowflake client or proxy's ability to make a peer-to-peer connection: how its NAT maps internal IP–port combinations to external ports, and how its firewall filters incoming packets. For our purposes, it suffices to condense the combinations of NAT mapping and firewall filtering into the following well-known variations:

Full cone The same internal IP–port pair always maps to the same external port. Any remote host may send a packet to an internal IP address and port by sending a packet to the mapped external port.

Restricted cone Like full cone, but incoming packets are allowed only if there has recently been an outgoing packet to the same remote IP address.

Port-restricted cone Like restricted cone, but incoming packets are allowed only if there has recently been an outgoing packet to the same remote IP–port pair.

Symmetric The external port depends on both the internal IP–port pair and the remote IP–port pair. Incoming packets are allowed only if there has recently been an outgoing packet to the same remote address.

Table 1 shows the pairwise compatibility of NAT variations. As the incompatible cases always involve a symmetric NAT, we further simplify matching by categorizing the variations into the two types *unrestricted* (works with most other NATs) and *restricted* (works only with more permissive NATs). Unrestricted proxies may be matched with any client; restricted proxies may be matched only with unrestricted clients. The broker prefers to match unrestricted clients with restricted proxies, in order to conserve unrestricted proxies for the clients that need them. Symmetric NAT is always considered restricted, but port-restricted cone NAT differs depending on the peer: for proxies it is restricted, but for clients it is unrestricted. The asymmetric categorization is an approximation to further help conserve unrestricted proxies for clients with symmetric NATs. Though it creates the potential for an incompatible match between a symmetric proxy and a port-restricted cone client, port-restricted cone proxies are common in practice, and are compatible with port-restricted cone clients. In case of a connection failure, clients re-rendezvous and try again.

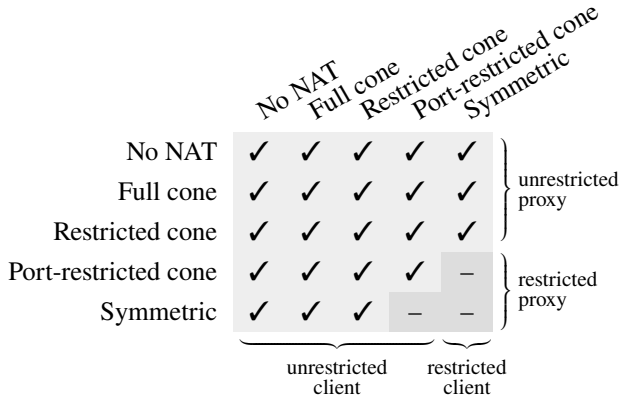

Table 1: Pairwise compatibility of NAT variants, using the facilities of STUN alone (no fallback to TURN). The incompatible cases are when one peer's NAT is symmetric and the other's is symmetric or port-restricted cone. Note the asymmetry in what NAT variants we consider "restricted" in client and proxy.

Check style guide to see if caption should be before or after a table.

Clients and proxies self-assess their NAT type and send it to the broker in their rendezvous messages. Clients use the NAT behavior discovery feature of STUN [22]. Not all STUN servers support NAT behavior discovery, but those whose addresses we ship with the Snowflake client do. Proxies cannot use the same technique, because the necessary STUN features are not available to JavaScript code in web browsers. Instead we adapt a technique from MassBrowser [25 §V-A]: we run a centralized, always-on WebRTC testing peer behind a simulated symmetric NAT. Proxies try to connect to this peer; if the connection succeeds, its type is unrestricted, otherwise it is restricted. Clients and proxies retest their NAT type periodically, to account for potential changes in their local networking environment. If a client or proxy is unable to determine its NAT type for some reason, it reports the type "unknown," which the broker conservatively treats as if it were restricted.

Figure 3 shows that unrestricted proxies form a relatively small fraction of the proxy population. In absolute terms, there are enough, thanks in large part to the volunteers who run the command-line version of the Snowflake proxy on networks

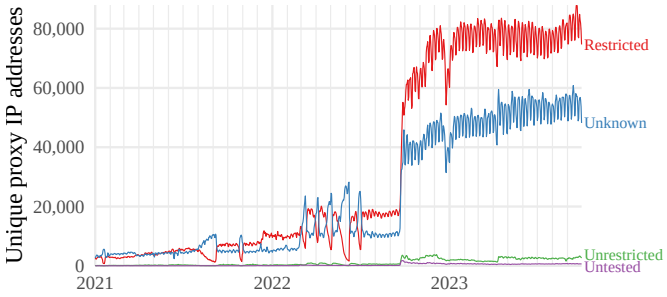

Figure 3: Proxy NAT types, in unique IP addresses per day. The places in 2021 and 2022 where there is an increase in the "unknown" NAT type and a decrease in the other types were the result of temporary operational problems with the testing peer that proxies use to assess their NAT type.

unencumbered by NAT. Though stable, long-term proxies are against the ethos of Snowflake, it has proved useful, as a matter of practicality, to sacrifice a measure of address diversity for better NAT compatibility in a common case. We can estimate how many tries it takes a client to be matched with a proxy, on average, by counting failed and successful rendezvous attempts at the broker, under the assumption that clients repeat rendezvous attempts until getting a match. In July 2023, unrestricted clients almost always got a match on the first attempt, while restricted clients needed an average of 1.07 attempts (standard deviation 0.05).

At the same time as the proxy makes a connection to its assigned client, it also connects to the bridge. For this connection we use the WebSocket protocol [24], which offers a TCP-like, point-to-point, client–server connection layered on HTTPS. The choice of protocol for the proxy–bridge link is arbitrary, and could be changed without affecting the rest of the system. The protocol does not need to be blocking-resistant; it just needs to be available to JavaScript code in web browsers. WebRTC would serve for this link too.

2.3 Data transfer

No complicated processing takes place at the proxy. The main value of a Snowflake proxy is its IP address: it gives the client a peer to connect to that is not on the censor's address blocklist. Having provided that, the proxy assumes a role of pure data transfer.

Snowflake uses a stack of nested protocol layers. We will walk though each layer and describe its purpose.

UDP DTLS SCTP	WebRTC data channel	ephemeral, per proxy
KCP smux	Turbo Tunnel	persistent, per session
Tor protocol application streams		

This is the stack for the client–proxy link, which is the place where WebRTC is used, and which is exposed to observation by the censor (Figure 1). The stack for the proxy–bridge link is the same, but with WebSocket in place of the WebRTC data channel at the top. The layers marked "ephemeral" are skimmed off and replaced as proxies come and go. The layers marked "persistent" are instantiated once in each circumvention session, hold long-term state, and are end-to-end between client and bridge.

The connection between a client and its proxy is a WebRTC data channel [18], which provides a way to send arbitrary binary messages between peers. A data channel is its own stack of three protocols: UDP for network transport, DTLS (Datagram TLS) for confidentiality and integrity, and SCTP (Stream Control Transmission Protocol) for delimiting message boundaries and other features like congestion control. Working UDP port numbers will have been discovered using ICE in the previous phase. The peers authenticate one another at the DTLS layer using certificate fingerprints that were exchanged during rendezvous [17 §5.1].

Data channels are well-suited to Snowflake's needs. (The specification even lists circumvention as a use case [18 §3.2].) But data channels are not the only option: WebRTC also offers *media streams* for unreliable transport of real-time audio and video. Which of these is used may be a fingerprinting vector. We will take up this topic in Section 3.

If clients only ever used one proxy, a WebRTC data channel alone would be sufficient. But a Snowflake proxy might disappear at any moment, and when that happens, its data channel goes with it. If the client was in the middle of a long download, for example, it should be possible to resume the download without interruption after rendezvousing with a new proxy. For this we need a shared notion of session state that exists at the client and the bridge, not tied to any temporary proxy. A lack of session continuity across proxy failures had been an unsolved problem in flash proxy [9 §5.2].

We adopt the Turbo Tunnel design pattern [7] and insert a userspace session and reliability protocol between the ephemeral proxy data channels and the client's own application streams. This part of the protocol stack outlives any single proxy; it belongs to the client and the bridge. Its primary function is to attach sequence numbers and acknowledgements to packets of data, so that both ends know what parts of the data stream need to be retransmitted after a temporary loss of proxy connectivity. The client tags its traffic with a random session identifier string that remains consistent throughout a session, which the bridge uses to index a map of session variables. For the inner session layer we use a combination of KCP [34] and smux [37]. KCP provides reliability, and smux detects the end of idle sessions and terminates them. KCP and smux have shown their worth in other deployments, and are easy to program, but there is nothing about them on which we depend essentially. Any other transport protocol that provides the necessary features and can be implemented in userspace would do,

such as QUIC, TCP, or (another layer of) SCTP. We prototyped successfully with QUIC before deciding on KCP/smux.

Finally, we must specify some concrete protocol so that the client can tell the bridge what remote destination to access on its behalf. In our deployment, this role is played by the Tor protocol. After stripping away the WebRTC and Turbo Tunnel containers, the Snowflake bridge feeds the client's data stream into a local Tor bridge. Almost anything would work here, with the caveat that it should be end-to-end secure between the client and bridge, to prevent inspection or tampering by curious or malicious proxies—Snowflake proxies are "untrusted messengers" in the sense of Feamster et al. [6 §3]. Integration with Tor has the nice feature that not even the Snowflake bridge is trusted to see the plaintext or destination of client traffic, let alone the temporary proxies. Using Tor also has some drawbacks, which we will comment on in Section 4.4 and Section 6.

3 Protocol fingerprinting

Snowflake leans heavily into the "address blocking" side of circumvention, but the "content blocking" part matters too. The goal, as always, is to make circumvention traffic difficult to distinguish from other traffic the censor cares not to block. Snowflake is inherently tied to WebRTC, and can only be effective against a censor that is not willing to block WebRTC protocols wholesale. But even within that scope, there are many variations in *how* WebRTC is implemented and used, which, if not carefully considered, might enable a censor to selectively block only Snowflake, while leaving other uses of WebRTC undisturbed. Unfortunately for the circumvention developer, the richness of WebRTC protocols affords a large attack surface for fingerprinting. Not only that, WebRTC leaves the details of signaling—in which peers exchange information needed to set up a connection, corresponding to Snowflake rendezvous—unspecified [1 §3], leaving every application to invent its own mechanism.

As WebRTC is designed for the web, most implementations of WebRTC are embedded in web browsers, and are not easily removed from that context. Snowflake originally used a WebRTC library extracted from Chromium, but that eventually proved unworkable for cross-platform deployment. Since 2019, Snowflake has used Pion [30], an independent implementation of WebRTC not tied to any browser. This is both good and bad. The good is greater agility and less development friction, and a working relationship with upstream developers that enables us to get fingerprinting-related changes made; we would not be where we are today without it. The bad is that the WebRTC fingerprint of Pion does not automatically match that of the mainly browser-originated WebRTC that Snowflake aims to blend in with.

The following is a list of fingerprinting concerns that bear on Snowflake, together with how we have tried to address them. The existence of a fingerprinting vulnerability does not automatically invalidate a circumvention system: censorship and circumvention are a dialog, and even among demonstrable vulnerabilities, some are more and some are less practical for a censor to take advantage of. The important thing is to have a solid foundation; minor flaws may be patched up as necessary.

Selection of STUN servers It is not unusual for a WebRTC application to use STUN, but the choice of what STUN servers to use is up to the application. Running dedicated STUN servers just for Snowflake would not work, because a censor would experience no collateral harm in simply blocking them by IP address. Our deployment uses a pool of public STUN servers that are used in applications other than circumvention, filtered for those that support the NAT behavior discovery feature described in Section 2.2. The client chooses a random subset of servers from the pool when it makes a connection; this is because not every STUN server is accessible under every censor.

Format of STUN messages STUN is most often deployed over plaintext UDP, which leaves its messages open to inspection and potential fingerprinting. STUN messages consist of a fixed header followed by a variable-length list of ordered attributes [29 §5]. What attributes appear, and their order, depends on the STUN implementation and how the application uses it.

We have not done anything in particular to disguise STUN messages. Though plaintext UDP is the most common, STUN specifies other transports, including encrypted ones like DTLS. These may be options for Snowflake in the future—of course, only if they are common enough that their use does not stick out on its own.

Rendezvous Because the rendezvous methods of Section 2.1 are modular, each one needs a separate justification as to why it should be difficult to block. Besides that, they must be implemented in a way that does not expose accidental distinguishers. For example, the domain fronting and AMP cache rendezvous methods use HTTPS, which is TLS, which means TLS fingerprinting is a concern [10 §5.1]. Snowflake, like many circumvention systems, uses the uTLS package [14 §VII] to get a client TLS fingerprint that is randomized or that imitates common browsers. See Section 5.2 for an account of when domain fronting rendezvous was briefly blocked in Iran, because we were slow in activating uTLS.

Though each rendezvous method may be difficult to block in itself, a censor might combine a low-confidence detection of rendezvous with features from other phases of the Snowflake data exchange to strengthen its guess.

DTLS The outermost layer of a WebRTC data connection, the protocol directly exposed to a censor, is DTLS (Datagram TLS) over UDP. DTLS is an adaptation of TLS [33 §1]

to the datagram setting, and therefore inherits the fingerprinting concerns of TLS [14]. TLS/DTLS fingerprinting may involve, for example, inspecting ClientHello messages to see what ciphersuites and extensions are used, and their order. It may be that a certain combination is specific to a particular implementation of a circumvention system, and may therefore be blocked at low cost.

Due to practical considerations, Snowflake's defenses to DTLS fingerprinting are not very robust, and are reactive rather than proactive. In the realm of TLS one may use uTLS, but there is as yet no equivalent of uTLS for DTLS. The present way of altering DTLS fingerprints in Snowflake is to submit a pull request upstream to Pion whenever a fingerprint feature used for blocking is identified. Section 5.1 documents how this has happened twice already, in response to blocking in Russia.

Data channel or media stream Besides data channels, WebRTC offers *media streams*, in line with its intended purpose of enabling real-time audio and video communication. Though both are encrypted, data channels and media streams are externally distinguishable because they use different containers. Data channels use DTLS, and media streams use DTLS-SRTP; that is, the Secure Real-Time Transport Protocol with a DTLS key exchange [32 §4.3].

Data channels are a closer match to Snowflake's communication model: media streams are meant to contain encoded audio and video, not arbitrary binary data. But the use of DTLS rather than DTLS-SRTP could become a significant feature if most other WebRTC applications use media streams. Although it would be less convenient, it would be possible to adapt the WebRTC link between the client and its proxy to use a media stream rather than a data channel, either by modulating binary data into a well-formed encoded audio or video signal in the manner of, say, Stegozoa [12 §3.3], or by directly replacing the ciphertext of SRTP packets, as in Protozoa [2 §4.4].

Fifield and Gil Epner [8] studied the network traffic of WebRTC applications, with the goal of revealing fingerprinting pitfalls that might affect Snowflake, which was then in early development. Frolov et al. [14 §V-C] observed that the unprotected TLS fingerprint of domain fronting rendezvous was distinctive, and introduced the uTLS package that Snowflake now uses to protect it. MacMillan et al. [23] focused on the DTLS handshake, comparing Snowflake to three other WebRTC applications. They correctly anticipated features of the Pion DTLS handshake that would later actually be used to block Snowflake in Russia; see more details in Section 5.1.

Chen et al. [4] combined features of rendezvous and DTLS in order to reduce false positives. Their classifier first prefilters by looking for DNS queries for STUN servers in the Snowflake client's default pool and the default front domain used in domain fronting rendezvous. While a single DNS query is not strong evidence of Snowflake, several related queries sent within a short time are more deserving of attention. They then apply a machine learning classifier to features of any subsequent DTLS handshake. The authors acknowledge that DTLS fingerprinting is fragile, as the DTLS fingerprint is, in principle, controllable by the application. The DNS prefilter may perhaps be mitigated by alternative rendezvous methods (Section 2.1), or by smarter selection of STUN servers by the client.

4 Experience

Snowflake has now been in operation for a few years. In lieu of a forward-looking evaluation, here we will take a look back at the history of our deployment and reflect on the experience.

4.1 Client counts and bandwidth

Snowflake became available to end users gradually, reflecting a long development process. Development began in late 2015, and deployment in 2017, but the system only really became usable in 2020. It began to attract large numbers of users (enough to merit a censor's attention) in 2022, following generalized network blocking events in Russia and Iran.

Snowflake shipped in the alpha release series of Tor Browser before graduating to the stable series. It was first released for GNU/Linux in Tor Browser 7.0a1 on 2017-01-24, and for macOS in Tor Browser 7.5a4 on 2017-08-08. But we hit a roadblock in attempting to prepare releases for other platforms: the Chromium-derived WebRTC library we had used to that point presented major difficulties in Tor Browser's cross-compiling, reproducible build environment. What let us resume making progress was a switch to Pion WebRTC [30] in 2019. With it, we were able to release Snowflake for Windows in Tor Browser 9.0a7 on 2019-10-01, and for Android in Tor Browser 10.0a1 on 2020-06-02.

While at this point Snowflake was available on every platform supported by Tor Browser, it was not yet comfortably usable. There were two important parts missing: no NAT type matching (Section 2.2) meant that a client could not always connect to its assigned proxy; and a lack of persistent session state (Section 2.3) meant that even if a proxy connection was successful, the client's session would end once that proxy disappeared. For these reasons, by early 2020, the average number of concurrent users had not risen above 40. The Turbo Tunnel session persistence feature became available to users in Tor Browser 9.5a13 on 2020-05-22. The client part of NAT behavior detection was released with Tor Browser 10.0a5 on 2020-08-19, and the necessary proxy support was added on 2020-11-17. After these changes, Snowflake became practical for daily browsing and the number of users began to grow into 2021.

This brings us to Figure 4, which shows the number of Snowflake users since 2021. The number of users is estimated

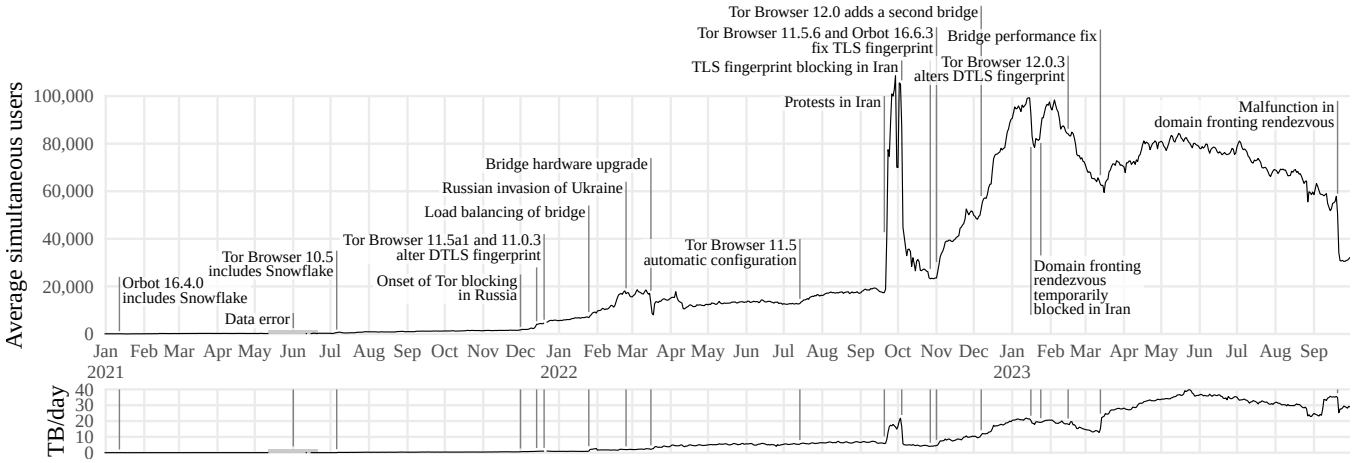

Figure 4: Estimated average simultaneous Snowflake users and bandwidth by day. The values at the far left of the graph, in early January 2021, are about 60 users and 4 GB/day.

using the aggregate statistics reported by the Tor bridge and the formulas of Tor Metrics [21]. Tor Metrics graphs are frequently misinterpreted. Be aware: the chart does not show a count of unique clients, but rather the *average number of concurrent clients* per day. For example, the value of 12,000 on 2022-05-01 means that, on average, 12,000 clients were using the service at any point in time on that day. The contribution of a client is independent of the number of temporary proxies it uses over the course of a session.

Snowflake's growth began in earnest when it became part of default installations. Orbot, a mobile app that provides a VPN-like Tor proxy, added a Snowflake client in version 16.4.0 on 2021-01-12. Snowflake graduated to Tor Browser's stable series in Tor Browser 10.5 on 2021-07-06, becoming a third built-in circumvention option alongside meek and obfs4. Being part of a stable release meant that it was easily available to all Tor users, not a self-selected group of alpha users. The number of users steadily increased over the next five months, reaching almost 2,000 by December 2021.

Network censorship events may have the contrary effects of either increasing or decreasing the number of users of a circumvention system. The user count will decrease if the system does not have enough resistance to prevent itself from being caught up in the blocking; but increase if it remains unblocked as one of a diminished number of ways to reach the outside world. Two such censorship events, one in Russia and one in Iran, had the effect of increasing the number of Snowflake users by multiples. (Because they also, in part, threatened Snowflake itself, we will have more to say about them in Section 5.)

On 2021-12-01, some ISPs in Russia deployed measures to block most forms of access to Tor, including Snowflake [38]. The measures varied in their effectiveness; in the case of Snowflake, blocking was triggered by a particular feature of the DTLS handshake, which we were able to mitigate in new releases within a few weeks. Over the next two months the total number of Snowflake users quadrupled, with most of the new users coming from Russia. The sudden demand temporarily overwhelmed the bridge, and for a few weeks Snowflake was almost unusably slow. It forced us to rearchitect the bridge for better scaling [11], as well as move the bridge to a powerful dedicated server and upgrade its network link from 1 Gbps to 10 Gbps. By May 2022, about 70% of Snowflake users were in Russia. The count of users in Russia got an additional small boost, visible in the graph, starting on 2022-07-14, when Tor Browser 11.5 added the Connection Assist feature, which automatically enables circumvention options when needed. Another DTLS blocking signature was reported on 2022-06-20; we did not get to fixing it until Tor Browser 12.0.3 on 2023-02-15. By that time the global user count had come to be dominated by users from Iran.

And Orbot 17 on. . .

The next event that had a large effect on Snowflake usage was the nationwide protests in Iran that started on 2022-09-21. The government imposed periodic network shutdowns and additional network blocking, severe even by the standards of a country already notorious for Internet censorship [3]. Internet users in Iran turned to circumvention systems that continued working despite the new restrictions, Snowflake among them. Adoption was rapid: on 2022-09-20, Iran accounted for only 1% of Snowflake users; on 2022-09-24 it was 67%. The massive influx of users had us scrambling to test and deploy performance improvements over the succeeding days. About two weeks later, on 2022-10-04, usage dropped almost as quickly as it had risen. The drop was caused by the censors in Iran blocking a TLS fingerprint used in rendezvous. After we released fixes for the TLS fingerprinting issue, the user count in Iran began to recover going into 2023. One of the emergency performance optimizations we had deployed in late September turned out to be a mistake and actually harmful for performance. The problem was masked, for a time, by the

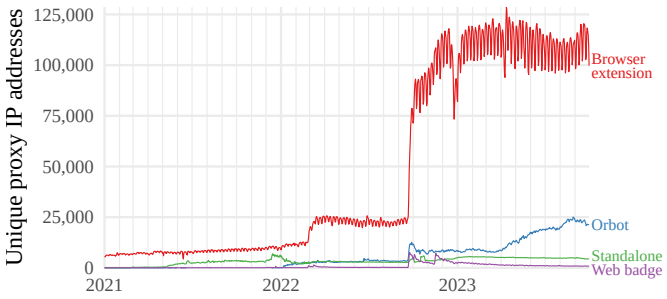

Figure 5: Unique proxy IP addresses per day, by proxy type. The two large steps visible in the graph correspond to the invasion of Ukraine by Russia in late February 2022, and protests in Iran beginning late September 2022, at which times there were campaigns to encourage people to install the browser extension. We have omitted occurrences of an unknown proxy type, of which there were fewer than 50.

great demand for the service, but starting in February 2023 the user count started to decline. We reverted the erroneous change in mid-March, and then the count started to recover again. Around this time, there was also a span of about a week, and irregular brief intervals thereafter, during which the domain used for domain fronting rendezvous was blocked in some ISPs in Iran, an evident attempt at blocking that was, however, not sustained.

Throughout most of this history, we ran the backend bridge on a single server, upgrading and optimizing it as needed. But as the bridge began to reach its hardware capacity, and performance improvements became harder to achieve, we deployed a second bridge to share the load. We discuss the challenges and design considerations of doing so in Section 4.4. The second bridge was made available to users in Tor Browser 12.0 on 2022-12-07. By July, the second bridge supported about 18% of Snowflake users.

As of 2023-09-30, Snowflake had transferred 10 PB of circumvention data. By this we mean goodput: Tor TLS traffic inside the tunnel, ignoring WebRTC, WebSocket, and KCP/smux overhead. At that time, about 1% of all Tor users (26% of bridge users) used Snowflake to connect to Tor.

Revisit this when Orbot 17 hits the Play Store. Caveat: front domain change of 2023-09-20 affected snowflake-02 unequally.

Discuss 2023-09-20 front domain change.

As of 2023-08-01, relay users are currently inflated due to #59.

4.2 Number and type of proxies

Snowflake's effectiveness depends on its proxies, of which there are several types. The primary type is the web browser extension, which, once installed, works in the background while the browser is running. There is also a "web badge" version of the proxy that does not require installation. It uses the same JavaScript code as the extension, but runs in an ordinary web page. Some people leave a browser tab idling on the web badge page, rather than install a browser extension. Apart from the web-based proxies, we provide a standalone, command-line proxy that does not require a browser. This version is convenient to install on a rented VPS, for example. Running a long-term proxy at a fixed IP address is somewhat at odds with Snowflake's goal of proxy address diversity and agility, but these standalone proxies are valuable because they tend to have less restrictive NATs, making them compatible with more clients. Finally, Orbot, a mobile app for accessing Tor, besides being able to *use* Snowflake for circumvention, can also *provide* Snowflake proxy service to others, a feature called "kindness mode."

Figure 5 shows the daily counts of each proxy type. Browser extension proxies predominate, representing about 81% of 130,000 daily IP addresses. For comparison, there were about 1,900 of the more traditional style of Tor bridge at this time. The difference is attributable to the relative ease of running a Snowflake proxy versus a Tor bridge—though the comparison is not quite direct, because Tor bridges have better defenses against enumeration and blocking than do Snowflake proxies.

Update percentage before submission.

It was not clear at the outset that it would even be possible to attract enough proxies to make Snowflake meaningfully blocking resistant and support a reasonable number of users. Initial growth in the number of proxies depended on our developing new and easier ways to run one, while later growth was driven by intentional advocacy and outreach. In the early days, circa 2017, the only round-the-clock proxy support was a few standalone proxies, run by us for the benefit of alpha tester clients. The browser extension became available in mid-2019. In the latter half of 2019, additional proxy capacity came when Cupcake, a browser extension for flash proxy with an existing user base, was repurposed for Snowflake. Orbot's Snowflake proxy feature was added in version 16.4.1 in February 2021. (In Figure 5, Orbot is counted among the standalone proxies until January 2022, when it got its own proxy type designation.)

It is worth reflecting briefly on the greater popularity of the browser extension compared to the web badge. The latter had been envisioned as the primary source of proxies in flash proxy, the idea being that people's browsers would automatically become proxies while reading sites that had the flash proxy badge installed, unless they checked an option to prevent it. We decided, early on, that flash proxy's opt-out permission had been a mistake, and that Snowflake would be only opt-in. In order to run a proxy, a person must to take a positive action such as installing a browser extension or activating a toggle on a web page. Our initial worry that this policy would reduce the number of proxies turned out to be unfounded. People find an informative, interactive proxy control panel more appealing than a nondescript badge graphic, and install the browser extension in greater numbers than ever used the web badge in flash proxy.

4.3 Proxy churn

The size of the proxy pool is not the only measure of its quality. Also important is its "churn," the rate at which it is replenished with fresh proxy IP addresses. Churn determines how hard a

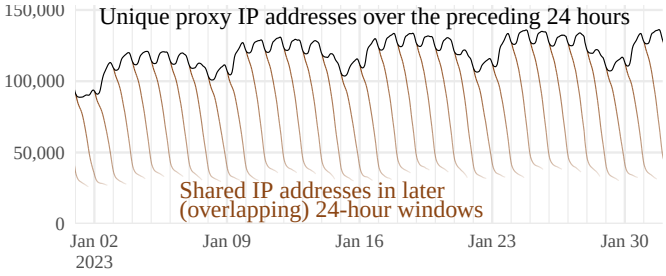

Figure 6: Proxy pool churn in January 2023. The dark upper line shows the number of unique proxy IP addresses in a 24-hour window starting at the point indicated. The lighter descending lines show how many of the same IP addresses remain in the pool, at 1-hour intervals up to 40 hours later. It takes about 20 hours for 50% of the proxy pool to turn over.

censor would have to work to keep a blocklist of proxy IP addresses up to date; or alternatively, how quickly a momentarily complete blocklist would lose effectiveness.

We ran an experiment to measure churn. Every hour, the broker logged a record of the proxy IP addresses it had seen in the past hour. To avoid storing real proxy IP addresses, each record was not a transparent list, but a HyperLogLog++ sketch [16], a probabilistic data structure for estimating the number of distinct elements in a multiset. We additionally hashed proxy IP addresses with a secret string before adding them to a sketch, to prevent their recovery from our published data. A sketch supports two basic operations: count and merge. Given a sketch X, we may compute an approximate count $|X|$ of its unique elements, and given two sketches X and Y, we may merge them into a new sketch representing the union $X \cup Y$. The quantity we are interested in, the size of the intersection of two sketches, is computed using the formula $|X|+|Y|-|X \cup Y|$. Such a computation estimates how many IP addresses are shared across two samples of the proxy pool.

Figure 6 visualizes the results of the churn experiment. We merged consecutive sketches over a 24-hour window to serve as a reference, then computed the size of its intersection with other windows of the same size, offset by $+1, +2, \ldots, +40$ hours. After 1 hour, the shifted window still has, on average, 97.3% of addresses in common with the reference; after 12 hours the fraction has fallen to 68.8%; by the time 24 hours have elapsed, only 38.2% of proxy IP addresses are ones that had been seen in the previous day.

4.4 Multiple bridges

In the abstract model of Figure 1, the bridge is a single, centralized entity. It *can* be centralized because it is never accessed directly, but only via temporary proxies. Unlike more traditional static proxy systems, Snowflake does not benefit, in terms of blocking resistance, from having multiple bridges. For scalability reasons, though, it is useful for "the" bridge to be realized as multiple servers, each handling a fraction of client traffic.

Our deployment now uses two bridges. Generalizing from one bridge to two required changes to the messages exchanged between clients, proxies, and the broker. Unfortunately, the fact of multiple bridges cannot be made fully transparent to clients, for technical reasons related to Tor. In our design, the client informs the broker of what bridge it wants to use, the broker conveys the choice to the proxy, and the proxy connects to the client's chosen bridge. This is in contrast to other imaginable designs where the choice of bridge is made by the broker or the proxy. We will discuss design considerations and tradeoffs.

Update on publication.

One minor difficulty is distributing the Turbo Tunnel layer. Recall from Section 2.3 that Snowflake has the notion of an end-to-end session between a client and the bridge, independent of temporary proxy connections that carry it. This is made possible by extensive state stored at the bridge: a table of clients, reassembly buffers, transmission queues, timers, and so on. While it is certainly possible to instantiate one such bundle of state variables per bridge, a session begun in one instance must remain with that instance—no other has the context necessary to make the packets of the session meaningful. This difficulty might be resolved by hashing the client's session identifier string to index a consistent bridge per session, as long as the set of bridges does not change too frequently.

There is another difficulty that is harder to work around. A Tor bridge is identified by a long-term identity public key. If, on connecting to a bridge, the client finds that the bridge's identity is not the expected one, the client will terminate the connection [5 §4.2]. The Tor client can configure at most one identity per bridge; there is no way to indicate (with a certificate, for example) that multiple identities should be considered equivalent. This constraint leaves two options: either all Snowflake bridges must share the same cryptographic identity, or else it must be the client that makes the choice of what bridge to use. While the former option is possible to do (by synchronizing identity keys across servers), every added bridge would increase the risk of compromising the all-important identity keys. Our vision was that different bridge sites would run in different locations with their own management teams, and that any compromise of a bridge site should affect that site only.

These considerations led us to a multi-bridge design in which clients have awareness of (at least a subset of) all bridges, and it is the client that chooses which bridge will be used for a particular session. The client includes a bridge identity string in its rendezvous message to the broker (Section 2.1); then the broker maps the identity to the WebSocket URL of the corresponding bridge, and conveys that URL to the proxy that's chosen to serve the client. We rely on clients choosing uniformly to equalize load across bridges. A consequence is that every bridge must meet a minimum performance standard: we cannot, say, centrally assign 20% of clients to one and 80% to another according to their relative capacity. Another drawback is that there is currently no way to instruct Tor to connect to only

one of the bridges it knows about (short of rewriting the configuration file): if two bridges are configured, Tor starts two sessions through Snowflake, each doing its own rendezvous, which is wasteful and makes for a more conspicuous network fingerprint. Still, this is the best solution we have found, given the constraints. A deployment not based on Tor would have more flexibility.

A client-chooses design risks misuse by clients, if not handled carefully. Clients should only be able to select from a limited set of known bridges, not cause proxies to connect to arbitrary destinations—otherwise the tens of thousands of Snowflake proxies might be weaponized to attack third parties. The client's bridge selection in its rendezvous message is represented not as an IP address or hostname, but as a hash of the bridge's public identity key. The broker maps the identity to a WebSocket URL by consulting its own local database of known bridges, and rejects rendezvous messages that refer to an unknown bridge. After the broker tells the proxy what WebSocket URL to connect to, the proxy does its own check, verifying that the hostname in the URL is a subdomain of a known suffix reserved for Snowflake bridges. So there are two independent safeguards against misuse.

5 Notable blocking attempts

In Section 4.1 we saw how Snowflake's user counts have at times been affected by the blocking actions of censors. Now we take a closer look at selected censorship events. The effect of censorship has usually been to increase rather than decrease the number of Snowflake users. This is no paradox: as censorship intensifies, users are displaced from less resilient to more resilient systems. Snowflake's blocking resistance has not in every case been an unqualified success, though, and here we also reflect on missteps and persistent challenges.

The examples of this section are taken from Russia, Iran, China, and Turkmenistan, and are selected for being significant and instructive. Common lessons are that lines of communication and a good working relationship with affected users are invaluable in quickly understanding and reacting to blocking; and that blocking resistance can only be understood in relation to a censor, because every censor's cost calculus is different.

Snowflake is blockable by any censor that is willing to block WebRTC. We would not claim otherwise. Indeed, we believe this is how circumvention systems should be presented: not by arguing their unblockability in absolute terms, but by laying out what actions by the censor would suffice to block it—or more to the point, *what sacrifices a censor would have to make* in order to block it. Some censors may be able to make those sacrifices; others may not. Advancing the state of the art of censorship circumvention consists in pushing blocking beyond the capabilities of more and more censors.

5.1 Blocking in Russia

Snowflake, along with other common ways of accessing Tor, was blocked in a subset of ISPs in Russia on 2021-12-01 [38]. The event was evidently coordinated and targeted, as it happened suddenly and affected multiple Tor-related protocols. Besides Snowflake, a portion of Tor relays and bridges, as well as some servers of the circumvention transports meek and obfs4, were blocked, at least temporarily. As might be expected, the attempt to block various blocking-resistant protocols was less than totally successful, and its ultimate effect was to substantially increase the number of users accessing Tor via circumvention transports, Snowflake among them.

We had the advantage of established relationships with developers and users in Russia, one of whom, through manual testing, found the traffic feature that was being used to distinguish Snowflake. It was DTLS fingerprinting, of the kind cautioned about in Section 3. Specifically, it was the presence of a supported_groups extension in the DTLS Server Hello message produced by Pion. The extension being present in Server Hello was a bug in itself, but it also afforded the censor a feature to match on that would detect DTLS connections with a Pion implementation in the server role, without affecting other forms of DTLS. The process of finding the flaw, fixing it, and shipping new releases of Tor Browser took a few weeks, after which the user count rose quickly: from the beginning to the end of December 2021, the number of users in Russia grew from about 400 to about 4,000 (Figure 7). Snowflake was to become a significant tool amid the general intensification of censorship in Russia following the invasion of Ukraine in February 2022.

The Server Hello supported_groups distinguisher used to detect Snowflake in Russia had been discovered and documented by MacMillan et al. [23 §3] already in 2020. We might have avoided this blocking event by proactively fixing the known distinguisher—but it was not necessarily the wrong call not to have done so. In a project like Snowflake, there is always more to do than time to do it; one must consider the opportunity cost of preempting specific blocking that may not come to pass. In this case, a reactive approach by us was enough: the loss was minor, and we were able to patch the problem quickly. Even in the ISPs where the blocking rule was present, it did not succeed at blocking 100% of Snowflake connections, because of the way it targeted a quirk of the Pion implementation of Server Hello. When the DTLS server role in the WebRTC data channel was played by a web browser proxy, not a standalone proxy or Snowflake client, the feature would not be present.

In May 2022 we got a report of a new detection rule, this time keyed on not just the presence, but the *contents* of the supported_groups extension, now at a byte offset suggesting that it targeted the Client Hello message, not Server Hello. The presence of a supported_groups extension in Client Hello is not at all unusual, but the specific groups offered by Pion's implementation differed from those of common browsers. Despite

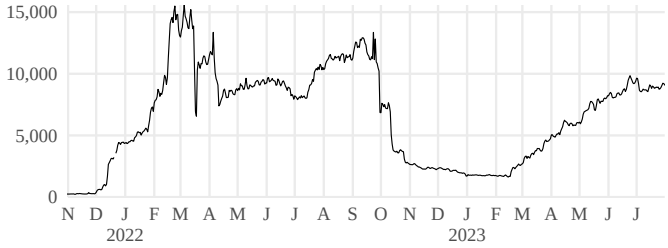

Figure 7: Snowflake users in Russia (average concurrent). The attempted blocking of Tor-related transports in December 2021 led to Snowflake's first surge in usage. The decrease in September–October 2022 coincided with an even larger influx of users from Iran.

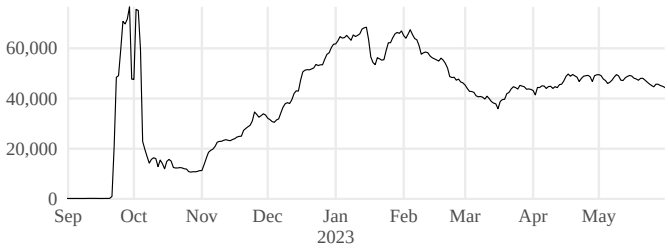

Figure 8: Snowflake users in Iran. Heightened censorship beginning in September 2022 caused Iran to become the single biggest source of Snowflake users. The drop in October 2022 was the result of TLS fingerprint blocking, which interfered with rendezvous and took some time to mitigate.

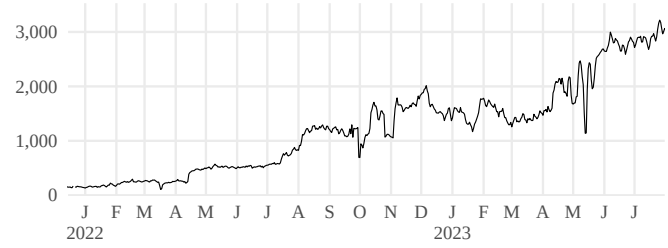

Figure 9: Snowflake users in China. Though no sustained blocking is evident, disruption of domain fronting rendezvous for three days in May 2023 briefly depressed user numbers.

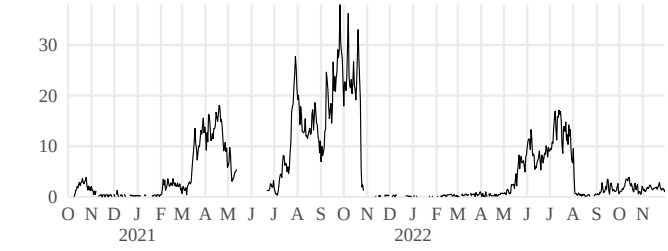

Figure 10: Snowflake users in Turkmenistan. Though there have never been many Snowflake users in Turkmenistan, blocking events are evident on 2021-10-24 and 2022-08-03.

our being able to confirm the existence of the new blocking rule, testers reported that Snowflake continued to work—which may have something to do with the fact that the Snowflake client does not always play the client role in DTLS. If the Snowflake client is the DTLS server, and the DTLS client is a browser proxy, then the byte pattern looked for by the blocking rule does not appear. We developed a mitigation, but by the time we had prepared a testing release in July 2022, the new rule had apparently been removed and replaced by another. We can only speculate as to motivations, but it may be that the censor found the old rule to have too many false positives, or simply not to be effective enough.

The detection rule that replaced supported_groups in Client Hello looked for the presence of a Hello Verify Request message. Hello Verify Request is an anti-denial-of-service feature in DTLS, in which the server sends a random cookie to the client, and the client sends a second Client Hello message, this one containing a copy of the cookie [33 §5.1]. Sending Hello Verify Request is not an error (it is a "MAY" in the RFC), but because the Pion implementation in Snowflake sent it, and major browsers did not, it was a reliable indicator of Snowflake connections. (Those, at least, in which the DTLS server role was played by a Snowflake client or standalone proxy.) This distinguisher had also been anticipated by MacMillan et al. [23 §3] in 2020. The first reports of this blocking rule were in July 2022; but as you can see in Figure 7, it had no apparent immediate effect. It is hard to say whether the drastic decline in October 2022 was a consequence of this rule, or some other, unidentified one. That decline coincided with an explosion of users from Iran, which temporarily affected the usability of the whole system. We deployed a mitigation to remove the Hello Verify Request message from Snowflake, regrettably, only in February 2023, after which the number of users in Russia began to recover.

The example of Snowflake in Russia demonstrates some of the difficulty of censorship measurement. The answer to a question like "Does Snowflake work in Russia?" is not a simple yes or no. It may depend on the date, the ISP, and even such factors as which endpoint plays the DTLS server role.

5.2 Blocking in Iran

In late September 2022, users from Iran became the majority of Snowflake users almost overnight, only to fall just as quickly two weeks later. See Figure 8. The cause of the rise was extraordinary new network restrictions amid mass protests [3]; the cause of the decline was TLS fingerprint blocking, which stopped Snowflake rendezvous from working. Specifically, it was a block of one of the TLS fingerprints of the crypto/tls package in the standard library of the Go programming language (the language in which the Snowflake client is written).

We say "one of" the fingerprints because there are several the package may output. This fact prevented the user count from falling all the way to zero, but also complicated our efforts to

identify the cause of the decline. We isolated two variables that were relevant in this case: the version of the Go standard library, and the presence or absence of hardware-accelerated AES. Between Go 1.17 and Go 1.18, the crypto/tls package stopped declaring support for the old protocol versions TLS 1.0 and 1.1 in the Client Hello message [15]. And if the operating environment has hardware-accelerated AES (common on desktop platforms, not so common on mobile), the crypto/tls package prioritizes AES ciphersuites; otherwise it prioritizes ChaCha ones. Of the four possible TLS fingerprints resulting from the combination of these two variables, only one was blocked, namely the one from Go 1.17, without AES acceleration. As it happens, it was mainly Orbot that was affected, because at the time it used a Snowflake client compiled with Go 1.17, and it runs on mobile platforms that are less likely to have AES acceleration. Tor Browser was relatively unaffected, because it either ran on desktops with AES acceleration, or on mobile platforms with the newer version of the Go standard library whose TLS fingerprint was not being matched. But evidently Orbot is more used in Iran than Tor Browser, because the decline was so drastic.

That simple TLS fingerprinting worked to block Snowflake rendezvous was negligence on our part. Anticipating such an event, we had already implemented TLS camouflage using uTLS in the Snowflake client, but failed to turn it on by default. Activating the feature required only a small configuration change, but we had to wait for new releases of Tor Browser and Orbot to get it into the hands of users: see the September–November 2022 interval in Figure 8.

The fact that only one (albeit the most common) of the Snowflake client's TLS fingerprints was blocked could be a sign of carelessness on the part of the censor. On the other hand, it is not certain that the TLS fingerprint blocking of October 2022 was meant to disrupt Snowflake specifically. Go is a popular language for implementing circumvention systems. Snowflake may have been caught up in blocking that was intended for another system.

After repairing the TLS fingerprinting flaw, the number of users from Iran gradually recovered to near its former peak. We are aware of only minor disruptions after this time. The default front domain used in rendezvous was blocked (by TLS SNI) in some ISPs between 2023-01-16 and 2023-01-24. We confirmed the block using data from the OONI censorship measurement platform. A reduction in users is visible at this time. AMP cache rendezvous was a successful workaround while the block was in effect. After the block was lifted, OONI measurements in the following weeks showed isolated cases of failure to connect to the front domain. These may have been further attempts at blocking. If they were, they did not have much of an effect.

5.3 Blocking in China

The user count graph from China, Figure 9, does not show any drastic changes like others we have discussed so far. There are a modest but respectable number of Snowflake users in China. Though there have been no singular, sustained events, we have seen evidence of short-term or tentative blocking of Snowflake in China.

In May 2019, when Snowflake was still only in alpha release, a user in China reported a failure to connect. Investigation revealed that the cause was IP address blocking of the few Snowflake proxies that existed at the time. Rendezvous worked, and the STUN exchange worked, but client and proxy could not establish a connection. We confirmed the evidence by temporarily running a new proxy at a different IP address: clients in China could connect when they happened to be assigned that proxy by the broker. This was back before the web browser extension proxy existed, and the only consistent proxy support was standalone proxies we developers ran ourselves at a static IP address. This problem went away as the proxy pool grew in size.

Later that month, we discovered another form of blocking in China: that of the default STUN server, of which there was only one configured at the time. The solution to this problem was to add more STUN servers and select a subset of them to use on each rendezvous attempt. Curiously, it seems that when the STUN server was blocked, the standalone proxies that had been blocked earlier that month became unblocked.

The next incidents we are aware of did not occur until 2023, recent enough to be in the scope of Figure 9. On May 12, 13, and 14, a few users reported problems with domain fronting rendezvous. We could not get systematic measurements, but some reports suggested that the blocking was triggered by multiple (two or three) HTTPS connections with the same TLS SNI to certain IP addresses within a short time. It is possible that Snowflake was not the main target of this blocking behavior, and was affected only as a side effect. If it indeed had to do with Snowflake, our best guess is that it was aimed at the multiple rendezvous mentioned in Section 4.4—though such a policy would certainly also affect a large number of non-Snowflake connections. The number of users from China was about halved during these three days. On May 15, this blocking went away and user counts returned to normal.

Also in May 2023, one user reported evident throttling (artificial reduction in speed by packet dropping) of TLS-in-DTLS connections, based on packet size and timing features. Such a policy would affect Snowflake, because it transports Tor TLS inside DTLS data channels. Reportedly, adding some padding to the first few packets to disrupt the size and timing signature was enough to prevent throttling. Our own speed tests run at the time did not show evidence of throttling, with or without added padding. There was no obvious reduction in the number of users. It may have been a localized, ISP-specific phenomenon.

5.4 Blocking in Turkmenistan

There have never been more than a few tens of Snowflake users in Turkmenistan. Even so, it has happened at least twice that the number of users dropped suddenly to zero, as you can see in Figure 10. We attributed the drops to multiple causes: filtering of the default broker front domain by DNS and TCP RST injection, and blocking of certain UDP port numbers commonly used by STUN.

Turkmenistan is a particularly challenging environment for circumvention. Though unsophisticated, the censorship there is more severe and indiscriminate than in other places we have seen. The fraction of the population that has access to the Internet is relatively small, which makes it hard to communicate with volunteer testers and lengthens testing cycles. We have been able to mitigate Snowflake blocking in Turkmenistan, but only partly, and after protracted effort.

The drop on 2021-10-24 was caused by blocking of the default broker front domain. We determined this by taking advantage of a feature of the Turkmenistan firewall, namely its bidirectionality. Nourin et al. [26 §2] provide more details; we will state just the essential information here. Among the censorship techniques used in Turkmenistan are DNS response injection and TCP RST injection. DNS queries for filtered hostnames receive an injected response containing a false IP address; TLS handshakes with a filtered SNI receive an injected TCP RST packet that tears down the connection. Conveniently for analysis, it works in both directions: packets that *enter* the country are subject to injection just as those that exit it are. By sending probes into the country from outside, we found that the default broker front domain was blocked at both the DNS and TLS layers. It was some time—not until August 2022—before we got confirmation from testers that an alternative front domain worked to get around the block of the broker.

The increase in the number users from May to August 2022 visible in Figure 10 was caused by a partial unblocking of the broker front domain on 2023-05-03. We realized this only in retrospect, by looking at logs from Censored Planet [35], a censorship measurement system that had continuous measurements of the domain at that time, in one autonomous system in Turkmenistan. There was a clear shift from RST responses to successful TLS connections on that date. DNS measurements are available only after that date, so they do not show a change, but they also showed no blocking. The unblocking evidently permitted some users to connect as before. But it must not have been nationwide, because as late as 2022-08-18, some users reported that RST injection was still in place for them (though DNS injection had ceased).

There was yet another layer to the blocking. Even if they could contact the broker (at the default or an alternative front domain), clients could not then establish a connection with a proxy. Further testing revealed blocking of the default STUN port, UDP 3478. A client that cannot communicate with a STUN server cannot find its own ICE candidate addresses (Section 2.2), which makes most WebRTC proxy connections fail. (The exceptions are proxies without NAT and ingress filtering. While there are some such proxies, censorship in Turkmenistan also outright blocks large swaths of the IP address space, including data center address ranges where those proxies tend to run.) As chance would have it, the NAT discovery feature we rely on for testing the NAT type of clients requires STUN servers to open a second, functionally equivalent listener on a different port [22 §6] (commonly port 3479). Changing to those alternative STUN port numbers let some users connect to Snowflake again. Specifically, STUN servers on port 3479 worked in AGTS, one of two major affected ISPs. The workaround did not work in Turkmentelecom, the other major ISP, where port 3479 was blocked. Though we do not have continuous measurements to be sure, we suspect that the STUN port blocking began on 2022-08-03 and precipitated the drop seen there in Figure 10.

The blocking techniques just described are crude, surely resulting in significant overblocking—but they nevertheless offer greater challenges to circumvention than the more considered blocking of Russia and Iran. We highlight this to make the point that blocking resistance cannot be defined in absolute terms, but only in relation to a particular censor and its predilections. Censors differ not only in resources (time, money, equipment, personnel), but also in their tolerance for the social and economic harms of overblocking. Circumvention can only respond to and act within these constraints. The government of Turkmenistan has evidently chosen to prioritize political control over a functioning network, to an extreme degree. To paraphrase one of our collaborators: "What they have in Turkmenistan can hardly be called an Internet." In a network already heavily damaged by oppressive policy, the marginal harm caused by the clumsy blocking of this or that circumvention system is relatively small. This explains the sense in which a resource-poor censor can "afford" certain blocking actions that a richer, more capable censor cannot.

Check that this holds water with someone who knows TM politics.

6 Future work

A natural extension of Snowflake would be to have it access systems other than Tor—ordinary VPNs, for example. Tor has its benefits: an existing user base, a standard (pluggable transports) for integrating circumvention modules, and exit nodes separate from entry nodes, which relieve the circumvention developer of the concerns associated with actually exiting traffic to its destination. But Tor has drawbacks as well, notably its lower speed and a lack of support for UDP and other non-TCP protocols. Nothing inherently ties Snowflake to Tor, and it might easily be adapted to other systems. One question is whether every Snowflake-like deployment should manage its own pool of proxies, or if proxies can somehow be shared. Building Snowflake's population of proxies has been a substantial undertaking in itself—for every project to have to repeat

the process from scratch would be a regrettable duplication of effort. There is no reason why one proxy might not serve multiple projects, the client expressing its preference in the same way it now signals which Tor bridge to use (Section 4.4). But there would be design issues to work out. While some proxy operators may be happy to donate bandwidth to a free-to-use project like Tor, they may need more incentive than altruism to help a commercial VPN. A shared deployment would impose additional friction on development (making it harder to alter the proxy protocol, for example). Rather than retrofit the current Tor-based proxies with support for other systems, a next-generation proxy pool might be designed from the ground up with multiple cooperating projects in mind. If it proved successful, the Tor deployment could migrate to it.

The Turbo Tunnel reliability layer of Section 2.3 was necessary for providing a continuous session abstraction over a sequence of unreliable proxies. But it might do even more: in particular, it should be possible for a client to multiplex its traffic over multiple proxies not just sequentially, but in parallel. (Something like multipath TCP.) Sequence numbers in the inner reliability layer would ensure a reliable stream, even when proxies have different lifetimes and performance characteristics. Multiplexing could increase performance by using the sum of the bandwidths of the individual proxies, and reduce variability by hedging against the client being assigned one very slow proxy. Using two or more proxies at once would eliminate the brief pause for re-rendezvous between consecutive proxies that now occurs. Our experiments with multiplexing have so far not shown enough benefit to justify the change, though it may be a matter of tuning. And of course, analysis would be required to determine whether simultaneous WebRTC connections form a distinctive network fingerprint.

Availability

The project web site, https://snowflake.torproject.org/, has links to source code and instructions for installing the proxy browser extensions.

Add Git clone URL or similar for the paper itself. Say it shows how to reproduce our figures. Must also include the churn logs of Section 4.3.

Acknowledgements

The Snowflake project has been made possible by the cooperation and support of many people and organizations. We want to thank particularly: Chris Ball, Griffin Boyce, Roger Dingledine, Sean DuBois, Arthur Edelstein, Mia Gil Epner, gustavo gus, J. Alex Halderman, Haz Æ 41, Jordan Holland, Armin Huremagic, Ximin Luo, Kyle MacMillan, Ivan Markin, meskio, Prateek Mittal, Erik Nordberg, Linus Nordberg, Vern Paxson, Sukhbir Singh, Aaron Swartz, ValdikSS, Philipp Winter, Censored Planet, China Digital Times, Greenhost, Guardian Project, Mullvad VPN, the Net4People BBS and NTC forums, OONI, the Open Technology Fund, Pion, the Tor Project, financial donors, and volunteers everywhere who run Snowflake proxies.

References

[1] Harald T. Alvestrand. Overview: Real-time protocols for browser-based applications. RFC 8825, January 2021. https://www.rfc-editor.org/info/rfc8825.

[2] Diogo Barradas, Nuno Santos, Luís Rodrigues, and Vítor Nunes. Poking a hole in the wall: Efficient censorship-resistant Internet communications by parasitizing on WebRTC. In *Computer and Communications Security*. ACM, 2020. https://www.gsd.inesc-id.pt/~nsantos/papers/barradas_ccs20.pdf.

[3] Simone Basso, Maria Xynou, Arturo Filastò, and Amanda Meng. Iran blocks social media, app stores and encrypted DNS amid Mahsa Amini protests, September 2022. https://ooni.org/post/2022-iran-blocks-social-media-mahsa-amini-protests/.

[4] Junqiang Chen, Guang Cheng, and Hantao Mei. F-ACCUMUL: A protocol fingerprint and accumulative payload length sample-based Tor-Snowflake traffic-identifying framework. *Applied Sciences*, 13(1), 2023. https://www.mdpi.com/2076-3417/13/1/622.

[5] Roger Dingledine and Nick Mathewson. Tor protocol specification, March 2023. https://spec.torproject.org/tor-spec.

[6] Nick Feamster, Magdalena Balazinska, Winston Wang, Hari Balakrishnan, and David Karger. Thwarting web censorship with untrusted messenger discovery. In *Privacy Enhancing Technologies*. Springer, 2003. http://nms.csail.mit.edu/papers/disc-pet2003.pdf.

[7] David Fifield. Turbo Tunnel, a good way to design censorship circumvention protocols. In *Free and Open Communications on the Internet*. USENIX, 2020. https://www.bamsoftware.com/papers/turbotunnel/.

[8] David Fifield and Mia Gil Epner. Fingerprintability of WebRTC. *CoRR*, abs/1605.08805, 2016. https://arxiv.org/abs/1605.08805.

[9] David Fifield, Nate Hardison, Jonathan Ellithorpe, Emily Stark, Roger Dingledine, Phil Porras, and Dan Boneh. Evading censorship with browser-based proxies. In *Privacy Enhancing Technologies*. Springer, 2012. https://crypto.stanford.edu/flashproxy/flashproxy.pdf.

[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 and Linus Nordberg. Running a high-performance pluggable transports Tor bridge. In *Free and Open Communications on the Internet*, 2023. https://www.bamsoftware.com/papers/pt-bridge-hiperf/.

[12] Gabriel Figueira, Diogo Barradas, and Nuno Santos. Stegozoa: Enhancing WebRTC covert channels with video steganography for Internet censorship circumvention. In *Asia CCS*. ACM, 2022. https://dl.acm.org/doi/10.1145/3488932.3517419.

[13] Sergey Frolov, Jack Wampler, Sze Chuen Tan, J. Alex Halderman, Nikita Borisov, and Eric Wustrow. Conjure: Summoning proxies from unused address space. In *Computer and Communications Security*. ACM, 2019. https://jhalderm.com/pub/papers/conjure-ccs19.pdf.

[14] Sergey Frolov and Eric Wustrow. The use of TLS in censorship circumvention. In *Network and Distributed System Security*. The Internet Society, 2019. https://tlsfingerprint.io/static/frolov2019.pdf.

[15] Go 1.18 release notes: TLS 1.0 and 1.1 disabled by default client-side, March 2022. https://go.dev/doc/go1.18#tls10.

[16] Stefan Heule, Marc Nunkesser, and Alex Hall. HyperLogLog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Extending Database Technology*. ACM, 2013. https://research.google/pubs/pub40671/.

[17] Christer Holmberg and Roman Shpount. Session Description Protocol (SDP) offer/answer considerations for Datagram Transport Layer Security (DTLS) and Transport Layer Security (TLS). RFC 8842, January 2021. https://www.rfc-editor.org/info/rfc8842.

[18] Randell Jesup, Salvatore Loreto, and Michael Tüxen. WebRTC data channels. RFC 8831, January 2021. https://www.rfc-editor.org/info/rfc8831.

[19] Ari Keränen, Christer Holmberg, and Jonathan Rosenberg. Interactive Connectivity Establishment (ICE): A protocol for network address translator (NAT) traversal. RFC 8445, July 2018. https://www.rfc-editor.org/info/rfc8445.

[20] Patrick Lincoln, Ian Mason, Phillip Porras, Vinod Yegneswaran, Zachary Weinberg, Jeroen Massar, William Simpson, Paul Vixie, and Dan Boneh. Bootstrapping communications into an anti-censorship system. In *Free and Open Communications on the Internet*. USENIX, 2012. https://www.usenix.org/conference/foci12/workshop-program/presentation/lincoln.

[21] Karsten Loesing. Counting daily bridge users. Technical Report 2012-10-001, The Tor Project, October 2012. https://research.torproject.org/techreports/counting-daily-bridge-users-2012-10-24.pdf.

[22] Derek MacDonald and Bruce Lowekamp. NAT behavior discovery using session traversal utilities for NAT (STUN). RFC 5780, May 2010. https://www.rfc-editor.org/info/rfc5780.

[23] Kyle MacMillan, Jordan Holland, and Prateek Mittal. Evaluating Snowflake as an indistinguishable censorship circumvention tool. *CoRR*, abs/2008.03254, 2020. https://arxiv.org/abs/2008.03254.

[24] Alexey Melnikov and Ian Fette. The WebSocket protocol. RFC 6455, December 2011. https://www.rfc-editor.org/info/rfc6455.

[25] Milad Nasr, Hadi Zolfaghari, Amir Houmansadr, and Amirhossein Ghafari. MassBrowser: Unblocking the censored web for the masses, by the masses. In *Network and Distributed System Security*. The Internet Society, 2020. https://www.ndss-symposium.org/ndss-paper/massbrowser-unblocking-the-censored-web-for-the-masses-by-the-masses/.

[26] Sadia Nourin, Van Tran, Xi Jiang, Kevin Bock, Nick Feamster, Nguyen Phong Hoang, and Dave Levin. Measuring and evading Turkmenistan's Internet censorship. In *The International World Wide Web Conference*. ACM, 2023. https://dl.acm.org/doi/abs/10.1145/3543507.3583189.

[27] OpenJS Foundation. How AMP pages are cached. https://amp.dev/documentation/guides-and-tutorials/learn/amp-caches-and-cors/how_amp_pages_are_cached [cited 2023-06-10].

[28] Marc Petit-Huguenin, Suhas Nandakumar, Christer Holmberg, Ari Keränen, and Roman Shpount. Session Description Protocol (SDP) offer/answer procedures for Interactive Connectivity Establishment (ICE). RFC 8839, January 2021. https://www.rfc-editor.org/info/rfc8839.

[29] Marc Petit-Huguenin, Gonzalo Salgueiro, Jonathan Rosenberg, Dan Wing, Rohan Mahy, and Philip Matthews. Session Traversal Utilities for NAT (STUN). RFC 8489, February 2020. https://www.rfc-editor.org/info/rfc8489.

[30] Pion WebRTC. https://github.com/pion/webrtc.

[31] Tirumaleswar Reddy, Alan Johnston, Philip Matthews, and Jonathan Rosenberg. Traversal Using Relays around NAT (TURN): Relay extensions to Session Traversal Utilities for NAT (STUN). RFC 8656, February 2020. https://www.rfc-editor.org/info/rfc8656.

[32] Eric Rescorla. WebRTC security architecture. RFC 8827, January 2021. https://www.rfc-editor.org/info/rfc8827.

[33] Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. The Datagram Transport Layer Security (DTLS) protocol version 1.3. RFC 9147, April 2022. https://www.rfc-editor.org/info/rfc9147.

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

[35] Ram Sundara Raman, Prerana Shenoy, Katharina Kohls, and Roya Ensafi. Censored Planet: An Internet-wide, longitudinal censorship observatory. In *Computer and Communications Security*. ACM, 2020. https://censoredplanet.org/censoredplanet.

[36] uProxy. https://www.uproxy.org/.

[37] xtaci. smux, February 2023. https://github.com/xtaci/smux.

[38] Maria Xynou and Arturo Filastò. Russia started blocking Tor, December 2021. https://ooni.org/post/2021-russia-blocks-tor/.