Snowflake, a censorship circumvention system using temporary WebRTC proxies

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. The temporary proxies are simple enough to be implemented in JavaScript, in a web page or browser extension, making them much cheaper to run than a traditional proxy or VPN server. The large and changing pool of proxy addresses resists enumeration and blocking by a censor. The system is designed with the assumption that proxies may appear or disappear at any time. Clients discover proxies dynamically using a secure rendezvous protocol. When an in-use proxy goes offline, its client switches 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 a significant circumvention tool 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 blocking attempts, and reflect on implications for circumvention generally.

1 Introduction

Snowflake is a censorship circumvention system, a system to enable network communication despite interference by a censor. Its blocking resistance comes from a large pool of low-cost, temporary proxies that varies over time and offers a censor no fixed target for blocking. The core research purpose of this paper is to investigate experimentally to what extent a circumvention system that makes the tradeoffs Snowflake does can be effective against contemporary censors. We will present the design of the system, listing the many challenges of circumvention and showing how Snowflake addresses them. We will explain how Snowflake solves the technical challenge of providing a good user experience when proxies are individually unreliable. We will document the reactions of national censors through case studies in Russia, Iran, China, and Turkmenistan over more than three years of deployment. On the way, we will provide quantitative evaluations of various facets of the system, including the number of clients served, and the size and composition of the proxy pool.

Censorship circumvention systems 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. On the spectrum of imitation to randomization, Snowflake falls on the side of imitation; on the scale of diffuse to concentrated, it is diffuse. Snowflake's defining quality is that it pushes the idea of distributed, disposable proxies to an extreme.

WebRTC is a suite of protocols intended for real-time communication applications on the web [1]. Video and voice chat are typical applications. Snowflake exchanges WebRTC data formats in the course of establishing a connection, and uses WebRTC protocols to traverse of NAT (network address translation) and to connect 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 and 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 contents of the client's communication, or the destination itself. The censor knows of the possibility of circumvention, and therefore seeks to block not only direct communication with the destination, but also indirect communication by way of a proxy or circumvention system. Circumvention is accomplished when the client can reliably reach any proxy, because a proxy, being outside the censor's control, can then 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.) The censor derives benefit from permitting some forms of network access: it cannot trivially "win" 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 [10], like Snowflake, used untrusted temporary JavaScript proxies in web browsers forwarding to a central bridge, but the link between client and proxy was WebSocket rather than WebRTC, which was then an emerging technology. Flash proxy was deployed from 2013 to 2016, but never saw much use, probably because WebSocket, which lacks the built-in NAT traversal of WebRTC, required clients to do complicated port forwarding. uProxy [38], 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 model was different from flash proxy's and Snowflake's. Censored clients would arrange, out of band, for an acquaintance outside the censor's network to run a proxy in their browser [39]. A personal trust relationship was necessary to prevent misuse, since browser proxies fetched destination content directly—meaning client activity would be attributed to the proxy, and the proxy might inspect the client's traffic. Clients did not change proxies on the fly. uProxy supported protocol obfuscation: its packets could be transformed to resemble something other than WebRTC. This was possible because of uProxy's implementation as a privileged browser extension with access to real sockets. Because Snowflake uses ordinary unprivileged APIs, its WebRTC can only look like WebRTC; on the other hand, for the same reason, Snowflake proxies are 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 [24], which offers proxying though volunteer proxies, called buddies. MassBrowser's architecture is similar to Snowflake's: there is a central component that coordinates connections between clients and buddies, which corresponds to a piece in Snowflake called the broker; buddies are like our proxies. Its 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. The buddy software is not constrained by a web browser environment, and can, like uProxy, use protocol obfuscation on the client–buddy link.

Other circumvention systems have used WebRTC, though without Snowflake's focus on numerous proxies. Protozoa [2] and Stegozoa [12] demonstrate point-to-point covert tunnels over WebRTC, the former by directly replacing encoded media with its own ciphertexts, the latter using video steganography. Significantly, where Snowflake now uses WebRTC data channels, Protozoa and Stegozoa use WebRTC media streams, which may have advantages for blocking resistance. We will say more on this point in Section 3. TorKameleon [40] is a WebRTC-based system with the dual goals of resisting blocking and complicating traffic correlation attacks. It uses a recent draft programming interface called WebRTC Encoded Transforms to support Protozoa-like embedding of data within media streams, without invasive browser modifications.

Our goal in this paper is to provide a realistic assessment of Snowflake, and neither to exaggerate its advantages, nor disproportionately emphasize the limitations of other systems. Circumvention research is a cooperative enterprise, and we recognize and support our colleagues who design and maintain systems of their own. With Snowflake, we have tried to explore a different point in the design space, and by this exploration widen the scope of effective circumvention techniques. 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 censor and its particular resources, costs, and motivations.

As of March 2024, Snowflake supports an estimated 35,000 average concurrent users at an average total transfer rate of 2.7 Gbit/s, which works out to around 29 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 proxy connect to each other with WebRTC, using information exchanged during rendezvous. Finally, there is data transfer, where the proxy transports data between the client and the bridge. The bridge is responsible 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 come and go. Proxy failure is not an abnormal condition—it happens, for example, when a browser running a proxy is closed. A client builds a circumvention session over a sequence of proxies, switching to a new proxy whenever the current one

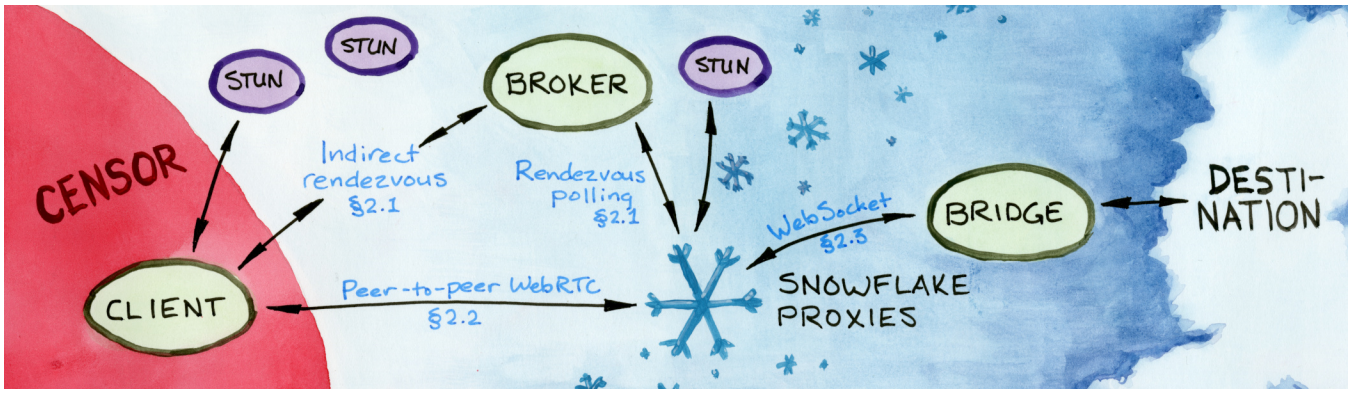

Figure 1: Architecture of Snowflake. The client contacts the broker through an indirect rendezvous channel with high blocking resistance. The broker matches the client with one of the proxies that are currently polling. The client and proxy connect to one another using WebRTC. The proxy connects to the bridge, then begins copying traffic in both directions. If the proxy disappears, the client does another rendezvous and resumes its session with a new proxy.

stops working. State variables stored at the client and the bridge let the session pick up where it left off. The change of proxies is invisible to applications using Snowflake (except for a brief delay for re-rendezvous). The Snowflake software presents an abstraction of one 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 the client sending a rendezvous message to the broker. An ambient population of proxies constantly polls the broker to check for clients in need of service. The broker matches the client with an available proxy, taking into account factors like NAT compatibility.

The client's rendezvous message is a bundle of data that the broker will need in order to match the client with a proxy, and that the proxy will need in order to connect to the client. The most important part of the rendezvous message is a Session Description Protocol (SDP) *offer* [27], which contains the information needed for a WebRTC connection, such as the client's external IP addresses and cryptographic data to secure a later key exchange. The broker gives the client's offer to a currently polling proxy, which sends back an SDP *answer* with its share of connection details. The broker forwards the proxy's answer to the client, and client and proxy then connect to one other directly. In WebRTC terms, this offer/answer exchange is called "signaling" [1 §2.2], and here the broker acts as a signaling server. To gather the information for an SDP offer or answer, clients and proxies communicate with third-party servers, called STUN servers, before contacting the broker. We will say more about how STUN is used in Section 2.2. Connecting to STUN servers is a normal part of WebRTC, though there are fingerprinting considerations that we cover in Section 3.

Interaction with the broker uses a "long-polling" model, shown abstractly in Figure 2. Proxies poll the broker periodically, using ordinary HTTPS requests. The broker holds proxy polling requests open for a few seconds, waiting for a client rendezvous message. If none arrives, the broker sends a response that says "no clients" and the proxy goes to sleep until its next poll. When a client does arrive, the broker responds to the proxy's poll request with the client's SDP offer. The proxy re-connects to the broker to send back its SDP answer. The broker sends the SDP answer to the client and an acknowledgement to the proxy. At this point rendezvous is finished: client and proxy have what they need to connect.

Proxies may contact the broker directly, because they are assumed to be uncensored. But clients must use an indirect, blocking-resistant channel, because any direct connection to the broker would be easily blocked by a censor. What is needed, essentially, is a miniature circumvention system to bootstrap the full system. If clients have access to a bootstrap rendezvous method that is good enough to reach the broker, why is a more extensive circumvention system needed at all? The answer is that the restricted scope of rendezvous admits a wider range of solutions than general circumvention. Techniques that would be too slow or expensive for high-volume or interactive circumvention may yet be suited to rendezvous, because rendezvous happens infrequently and transmits only small amounts of data.

The nice thing about rendezvous that it is modular and separable. More than one method may be used, and the methods need not have anything in common with the main system. Anything that can be persuaded to convey a message of about 1,500 bytes indirectly to the broker, and return a response of about the same size, may work as a Snowflake rendezvous module. Snowflake now supports three rendezvous methods:

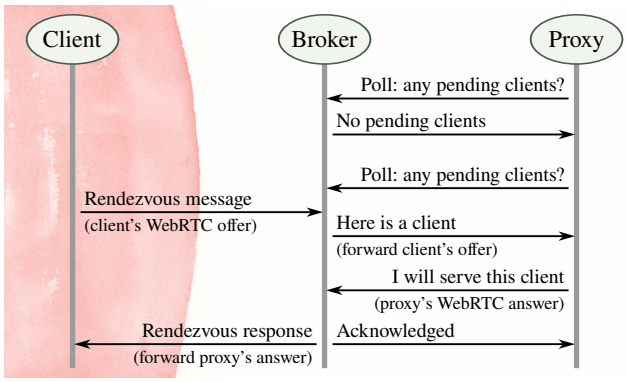

Figure 2: Information exchange in Snowflake rendezvous. When the broker makes a match, the proxy receives the client's SDP offer, then re-connects to send back its SDP answer. It all happens during one round trip from the client's point of view. Not shown is the indirect channel the client uses to access the broker.

Domain fronting In this method, the client does an HTTPS exchange with the broker through an intermediary web service such as a content delivery network (CDN), setting the externally visible hostname (the TLS Server Name Indication, or SNI [6 §3]) to a "front domain" different from the broker's. The CDN routes the HTTPS request to the broker according to the the HTTP Host header, which, under TLS encryption, reflects the actual hostname of the broker [11]. A censor cannot easily block domain-fronted rendezvous without also blocking unrelated connections to the front domain, which should be selected to have high value to the censor. The well-known drawback of domain fronting is the high cost of CDN bandwidth, but this is not a big problem when it is used only for rendezvous.

AMP cache AMP is a framework for web pages written in a restricted dialect of HTML. Part of the framework is a free-to-use cache server [26]. The cache fetches AMP-conformant pages on demand, making it effectively a restricted sort of HTTP proxy. We have a module that encodes rendezvous messages to conform to AMP requirements, allowing them to be exchanged with the broker via the AMP cache.[1] This rendezvous method is not easily blocked without blocking the cache server as a whole. It still technically requires domain fronting, because the AMP cache protocol normally exposes the broker's hostname in the TLS SNI, but it enlarges the set of usable intermediaries and front domains.

SQS (Simple Queue Service) Amazon SQS is a message queuing service designed for communication between microservices. Snowflake has the ability (new at the time of this writing) to use a message queue as a one-way communication channel.[2] Clients write into a public queue, which the broker reads from. Responses are sent back through dynamically created, single-use queues. All communication is indirect, via SQS servers.

Rendezvous is not unique to Snowflake. Other examples of rendezvous are the DEFIANCE Rendezvous Protocol [20 §3], the facilitator interaction in flash proxy [10 §3], and the registration proxy in Conjure [13 §4.1]. A key property of Snowflake and the mentioned systems is that their blocking resistance does not rely on preshared secret information. Whatever information is needed to establish a circumvention session is obtained dynamically at runtime. This is in contrast to other systems in which, before making a connection, the client must acquire some secret, such as an IP address or password, through an out-of-band channel—and blocking resistance depends on keeping that information secret. A corollary of the no-secret-information property is that an adversary is at no special disadvantage in attacking the system. There is no out-of-band channel which real clients have access to but the censor does not. The censor may pose as a client, download the software, study its network connections—and the system must maintain its blocking resistance despite this. The disadvantage of a separate rendezvous step is that it is one more thing to get right. Both the main circumvention channel and the rendezvous must resist blocking: the combination is only as strong as the weaker of the two.

2.2 Peer-to-peer connection establishment

After rendezvous, the client and its assigned proxy connect to one another directly. Even in the absence of censorship, making a connection between two Internet peers is not trivial, because of possible interference by NAT (network address translation) and firewalls. Snowflake clients and proxies run in diverse networks with varying NATs and ingress policies. Fortunately for us, WebRTC is designed with this use case in mind. It has built-in support for NAT traversal in the form of ICE (Interactive Connectivity Establishment) [19], a procedure for testing combinations of peer addresses until finding one that works. ICE uses third-party STUN (Session Traversal Utilities for NAT) servers [28] that, among other services, let a host discover its own external IP addresses. The first part of ICE happened 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. After rendezvous, the peers try pairs of candidate addresses until they are able to establish a connection.

There is no guarantee that two hosts will be able to connect using the facilities of STUN alone. Some combinations of address mapping and filtering are simply incompatible. In such a

[1] https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/merge_requests/50

[2] https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/merge_requests/214

	No NAT	(EI, EI)	(EI, AO)	(EI, AP)	(ED, ED)	
No NAT	✓	✓	✓	✓	✓	Unrestricted proxy
(EI, EI)	✓	✓	✓	✓	✓	
(EI, AO)	✓	✓	✓	✓	✓	
(EI, AP)	✓	✓	✓	✓	–	Restricted proxy
(ED, ED)	✓	✓	✓	–	–	
	Unrestricted client				Restricted client	

Table 1: Compatibility of NAT variations, assuming the use of STUN only (no fallback to TURN). Variations are represented by (*mapping*, *filtering*) behavior pairs. The table uses codes to indicate behaviors: EI (endpoint-independent), AO (address-dependent), AP (address- and port-dependent), and ED (either AO or AP). The incompatible cases are when one peer has ED mapping and the other has ED mapping or AP filtering. Note the asymmetry in what NAT variations are considered "restricted" in client and proxy.

case, ICE would normally fall back to relaying traffic through a TURN (Traversal Using Relays around NAT) server [30], a kind of UDP proxy. A fallback to TURN would be problematic for Snowflake, because the TURN servers themselves would become targets of blocking. But Snowflake has an advantage other WebRTC applications do not: most applications want to connect *a particular* pair of peers, whereas we are happy to connect a client to *any* proxy. Snowflake clients and proxies self-assess their NAT type and report it to the broker, which then avoids making matches that would require TURN.

Table 1 shows the compatibility of NAT variations. A NAT variation is defined by a combination of address mapping and filtering behaviors [21 §3]. Endpoint-dependent mapping means that the address a local endpoint is translated to depends on the remote endpoint's IP address (and possibly port). Endpoint-dependent filtering means that incoming packets from a remote endpoint are allowed only if outgoing packets have already been sent to that endpoint. As the incompatible cases always involve endpoint-dependent address mapping (sometimes called symmetric NAT), we further categorize 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. Endpoint-dependent mapping is always considered restricted, but the type of NATs with address- and port-dependent filtering differs depending on the peer: for proxies it is restricted, but for clients it is unrestricted. This is a heuristic to conserve unrestricted proxies for clients with endpoint-dependent NAT mapping. Though

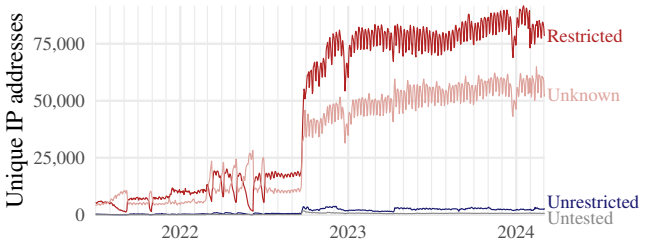

Figure 3: Proxy NAT types, in unique IP addresses per day. The places in 2021 and 2022 where "unknown" displaced other types were caused by operational problems with the NAT type testing peer.

it creates the potential for an incompatible match between a client with an address- and port-dependent filter and a proxy with endpoint-dependent mapping, this case is rare in practice, and if it happens, a client can re-rendezvous and try again.

Clients use the NAT behavior discovery feature [21] of STUN to self-assess their NAT type. Proxies cannot use the same technique, because the necessary STUN features are not exposed to JavaScript. For proxies, we adapt a technique from MassBrowser [24 §V-A] and run a centralized, always-on WebRTC testing peer behind a simulated NAT with endpoint-dependent mapping.[3] If a proxy can connect to the testing peer, its type is unrestricted; otherwise it is restricted. If a client or proxy is unable to determine its NAT type, it reports the type "unknown," which the broker conservatively treats as restricted.

Unrestricted proxies are a relatively small fraction of the proxy population, as Figure 3 shows. In absolute number, though, there are enough for restricted clients at current levels of use. The broker counts the number of unmatched (U) and matched (M) client rendezvous requests per day. If we assume that clients attempt rendezvous repeatedly until getting a match, then the number of attempts per success is $(U+M)/M$. Under this assumption, as of March 2024, the average client needs 1.01 rendezvous attempts to find a compatible proxy.

As the proxy negotiates the WebRTC connection with its client, it also connects to the bridge using WebSocket [23]. Unlike the client connection, the bridge connection presents no challenges: it's just HTTPS to an open port on a server. The choice of WebSocket for this link is arbitrary, and another protocol might be substituted in its place. It does not need to be blocking-resistant (because we are already outside the censor's zone of control), it only needs to be available to JavaScript code in web browsers.

2.3 Data transfer

No complicated processing occurs at the proxy. The main value of a Snowflake proxy is its IP address: it gives the

[3] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/40013

client a peer to connect to that is not on the censor's address blocklist. Having provided that, the proxy assumes a role of data transfer, copying data upstream from client to bridge and downstream from bridge to client.

Snowflake uses a stack of nested protocol layers. Figure 4 shows the stack for the link between the client and the proxy. This is the link that uses WebRTC, the one that is exposed to the censor. The stack for the proxy–bridge link is the same, but with WebSocket replacing WebRTC in the outermost layer. Layers at the top of the diagram are "outer," closer to the network; ones at the bottom are "inner," closer to the user.

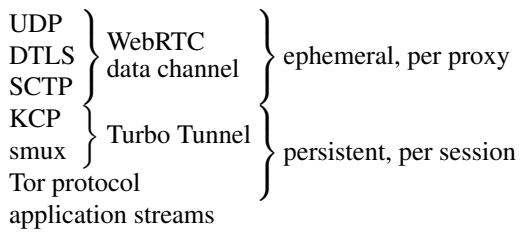

Figure 4: The protocol stack for the client–proxy link.

Each layer serves a different purpose. The layers marked "ephemeral" are replaced at every proxy switchover. These outer layers form the "carrier" over which the more stateful inner layers are transported. The layers marked "persistent" are instantiated just once per circumvention session and outlive any single proxy. They maintain end-to-end state and present an abstraction of a single long-lived tunnel. This virtual tunnel then transports user application traffic such as web browsing and messaging.

The point-to-point link between a client and its proxy is a WebRTC data channel [18]. Data channels let two WebRTC peers exchange arbitrary binary messages. A data channel is itself a composition of three protocol layers: 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. The peers authenticate each other at the DTLS layer using certificate fingerprints that were exchanged during rendezvous [17 §5.1].

Data channels are well-suited to the circumvention use case. But data channels are not the only option in WebRTC: there are also *media streams*, meant for real-time audio and video. Which of these options is used is an externally observable feature, and may therefore become a fingerprinting vector. We will consider this topic further in Section 3.

A proxy maintains two connections for each client it is currently serving: a WebRTC connection to the client and a WebSocket connection to the bridge. Data received from the client over WebRTC is copied to the bridge over WebSocket, and vice versa. Although this pair of connections effectively connects the client to the bridge, it alone is not enough for usable circumvention, because Snowflake proxies do not last forever. When a proxy goes away, its WebRTC and WebSocket connections go with it. Without additional consideration, the loss of a proxy would mean the end of a client's circumvention session. What is needed is a separate notion of session state, independent of the current proxy and its data channel.

We adopt the Turbo Tunnel design pattern [8] and insert a userspace session and reliability protocol between the ephemeral "carrier" protocols and the client's application streams.[4] This inner protocol attaches sequence and acknowledgement numbers to the pieces of data that pass through the tunnel. After a temporary break in proxy connectivity, the client and bridge first retransmit whatever data has not been acknowledged by the other side, then carry on as normal with new data, with no duplication or gaps. How this works, concretely, is the client chooses a random session identifier string at the beginning of its session, and sends it as a preamble every time it connects through a new proxy. When the bridge accepts a WebSocket connection, it inspects the preamble to find the client's session identifier string, which it uses to index a table of session state information. If the session does not yet exist, the bridge creates one; otherwise it resumes an existing session (starting by retransmitting any unacknowledged data). If a client's proxy disappears in the middle of a long download, for example, there may be a pause while Snowflake does another rendezvous to acquire a fresh proxy, but after that, the download continues uninterrupted.

For the inner session layer we use a combination of KCP [33] and smux [43]. KCP provides reliability and retransmission, and smux detects and terminates timed-out streams. Other userspace protocols, for example QUIC or TCP, could be used more or less equivalently for this purpose. We prototyped successfully with both QUIC and KCP/smux before settling on the latter. The main considerations that influenced our decision were familiarity and ease of maintenance.

One more protocol layer is needed before sending user application data through the tunnel: an end-to-end secure channel between the client and the bridge, using keys unknown to the proxy. The inner session layer does not itself provide security, and the DTLS of the data channel is only hop-by-hop, not end-to-end. The purpose of this additional secure channel is to prevent proxies from inspecting or tampering with the traffic they carry. Nothing special is needed here; for example, TLS, or any VPN protocol, would work fine. Our deployment uses the Tor protocol as this secure channel. After removing the WebSocket and Turbo Tunnel layers, the bridge feeds the data it receives into a local Tor bridge, which routes the stream into the Tor network and eventually to its destination. Tor, of course, has privacy advantages that are not, strictly speaking, necessary for circumvention, but are nevertheless nice to have, such as that not even the Snowflake bridge is trusted to know the contents or destinations of client streams. But Tor also has drawbacks, which we will comment on in Section 4.4 and Section 6.

[4] https://lists.torproject.org/pipermail/anti-censorship-team/2020-February/000059.html

Snowflake may be seen as an instance of the "untrusted messengers" model of Feamster et al. [7 §3]. Their *messengers* correspond to our *proxies*; their *portal* is our *bridge*. Proxies are trusted to deliver the client's traffic to the bridge, but do not themselves connect to the destination, or even know what it is. An inner layer of cryptography protects the client's traffic from observation and manipulation by malicious proxies. The protection goes in the other direction as well: because proxies are programmed to connect only to a Snowflake bridge, and they never process anything but ciphertext, a malicious client cannot cause a proxy to misbehave or have its actions attributed to the proxy. Without this mutual guarantee of safety, it would be too risky to associate a client and proxy who have no preexisting trust relationship.

3 Protocol fingerprinting

Snowflake's main focus is the "address blocking" side of circumvention, but the "content blocking" side matters too. The goal, as always, is to make circumvention traffic hard to distinguish from traffic the censor cares not to block. Design decisions in Snowflake—the use of WebRTC, and the requirement to run proxies in browsers—mean that Snowflake can, at best, only blend in with other WebRTC traffic. But even within that scope, there are variations in *how* WebRTC is implemented and used, which, if not carefully considered, might enable a censor to selectively block only Snowflake, leaving other uses of WebRTC undisturbed. Unfortunately for the circumvention developer, the richness of WebRTC protocols creates a large attack surface for fingerprinting. And then, there are fingerprinting considerations beyond WebRTC.

The most prevalent implementations of WebRTC are in web browsers. Snowflake originally used a WebRTC library extracted from Chromium[5], but that proved to be difficult to maintain and build for multiple platforms. Since 2019, Snowflake uses the Pion [29] implementation of WebRTC.[6] Pion is not tied to any browser, which is both good and bad. The good is less development friction, better memory safety (Pion is written in Go, Chromium WebRTC in C++), and a working relationship with upstream developers to have fingerprinting changes made when needed. The bad is that Pion's protocol fingerprints do not automatically match those of the mostly browser-originated WebRTC that Snowflake aims to blend in with.

The following is a list of the main fingerprinting concerns in Snowflake and what we have done to address them. A fingerprinting vulnerability does not automatically disqualify a circumvention system: it depends on whether the vulnerability can be fixed without changing the basic nature of the system. The important thing is to have solid fundamentals: minor flaws may be patched up as needed.

[5] https://github.com/keroserene/go-webrtc

[6] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/28942

Selection of STUN servers The use of STUN with WebRTC is common, but the choice of what STUN servers to use is up to the application. Running dedicated STUN servers for Snowflake would not work, because a censor would experience no collateral harm in blocking them. Our deployment uses a pool of public STUN servers that are used for purposes other than circumvention. The client chooses a random subset for each new session.

Format of STUN messages STUN is most often deployed over plaintext UDP, which leaves the contents of messages open to inspection. STUN messages consist of a fixed header followed by a list of attributes [28 §5]. What attributes appear, and their order, depends on the STUN implementation and how the application uses it.

We have not done anything particular to disguise STUN messages. Though UDP is the most common, STUN specifies other transports, including encrypted ones like DTLS. The alternative transports may be options for Snowflake in the future—of course, only if they are commonly used enough not to stick out in themselves.

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. Each method must also be implemented in a way that does not expose accidental distinguishers. For example, the domain fronting, AMP cache, and SQS rendezvous methods use HTTPS, which means TLS fingerprinting is a concern [11 §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. Section 5.2 has an account of when domain fronting rendezvous was briefly blocked in Iran, because we were slow in activating uTLS.

DTLS The outermost layer of a WebRTC data connection, directly exposed to a censor, is DTLS over UDP. DTLS is an adaptation of TLS [32 §1] to the datagram setting, and therefore shares the fingerprinting concerns of TLS [14].

Owing 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 for DTLS. The present way of altering DTLS fingerprints in Snowflake is to submit a patch to Pion when a feature used for fingerprinting is identified. Section 5.1 documents how this has happened twice, in response to blocking in Russia.

Data channel or media stream Along with data channels, WebRTC offers *media streams*, which transmit encoded audio and video. Though both data channels and media streams are encrypted, they are externally distinguishable because they use different encryption containers. Data channels use DTLS, while media streams use

DTLS-SRTP; that is, the Secure Real-Time Transport Protocol with a DTLS key exchange [31 §4.3].

Data channels are a closer match to Snowflake's communication model: media streams are meant for audio and video, not arbitrary binary messages. But the use of data channels could become a fingerprinting feature if other WebRTC applications mainly use media streams. Should it become necessary, it would likely be possible to adapt Snowflake to use media streams rather than data channels, either by modulating data into an audio or video signal in the manner of, say, Stegozoa [12 §3.3], or by replacing the encoded data inside SRTP packets, as in Protozoa [2 §4.4] or TorKameleon [40 §III-D].

Most research on detecting Snowflake to date has focused on protocol fingerprinting. Fifield and Gil Epner [9] studied the network traffic of WebRTC applications, with the goal of finding pitfalls that could affect Snowflake. Frolov et al. [14 §V-C] observed that the undisguised TLS fingerprint of Snowflake's domain fronting rendezvous was distinctive, and introduced the uTLS package now used to protect it.

MacMillan et al. [22] focused on the DTLS handshake, comparing Snowflake to three other WebRTC applications. They correctly anticipated features of the Pion DTLS handshake that would later be used to block Snowflake in Russia; see details in Section 5.1. Holland et al. [16 §5.3], using the bits of UDP datagrams directly as features, demonstrated approximately equal performance on the same DTLS handshake data set. Their automatically derived classifier assigned high feature importance to packet length fields, in fact doing well even without DTLS payload features.

Chen et al. [4] combined features of rendezvous and DTLS in order to reduce false positives. Their classifier begins by looking for DNS queries for STUN servers and front domains commonly used by Snowflake clients. They then apply a machine learning classifier to features of a subsequent DTLS handshake. The authors acknowledge that DTLS fingerprinting is fragile, as the DTLS features are, 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.

Xie et al. [42] trained a decision tree to distinguish domain fronting rendezvous from certain other HTTPS exchanges, using packet size, direction, latency, and bandwidth features. Wails et al. [41] criticize past research on detecting circumvention systems, saying that accuracy claims do not hold up in light of the low base rates of circumvention traffic in practice. They developed classifiers for Snowflake and other circumvention protocols that improved on the state of the art, but found them still to be prohibitively imprecise at realistic base rates. They propose reducing false positives by combining multiple observations per IP address—classifying hosts, not flows—and suggest that Snowflake's lack of fixed proxies mitigates against this enhancement.

Related to protocol fingerprinting is *traffic analysis*: classifying connections based on features like packet lengths and transmission times, which may differ, in a circumvention system, from other uses of the cover protocol. The best classifiers of Xie et al. and Wails et al. were of this type. While traffic analysis attacks are worth thinking about, we caution that academic audiences have historically overestimated their importance. Tschantz et al. [36 §VII] observed that censors are sensitive to costs and—particularly—false positives. They claim (and our experience bears it out) that censors prefer traffic classification rules that are simple, precise, and deterministic, and avoid ones that require managing state, are computationally expensive, or have non-negligible false positive rates, like those based on traffic analysis. Nevertheless, we have tried to future-proof Snowflake in this respect: the protocol inside the WebRTC data channel supports shaping of transmission sizes and timing, which ought to be sufficient to imitate the traffic fingerprint of other WebRTC applications. But the feature is currently unused.

4 Experience

Snowflake has now been in operation for a few years. In lieu of a forward-looking evaluation, here we take a look back at the history of its deployment and reflect on the experience. Fielding a new circumvention system offers an opportunity to study experimentally its uptake and performance. In this section we discuss the number of clients over time and the bandwidth they use, the size and composition of the proxy pool, the variability of proxy IP addresses, and engineering considerations related to scaling. We take up the topic of reactions by censors in Section 5.

4.1 Client counts and bandwidth

As a consequence of our use of a Tor bridge as the backend for the Snowflake bridge, we can estimate client counts and bandwidth using the privacy-preserving techniques of Tor Metrics [35]. Tor bridges publish daily summaries of statistics, including client counts and bandwidth, as well as an aggregate distribution of client countries inferred from a local geolocation database. Figure 5 shows the number of clients and amount of bandwidth consumed since 2021. The upper graph depicts not a count of unique clients, but rather the *average number of concurrent clients* per day. For example, when the graph passes through 10,000 on 2022-02-04, it means there were, on average, 10,000 clients using Snowflake at a given time on that day. The number of users is estimated by counting the "directory requests" that Tor clients make periodically. The contribution of a client session depends on its duration, not on how many temporary proxies it uses; a client connected for two hours counts half as much as one connected for four hours. Bandwidth is computed as the average of incoming and outgoing bytes (which are approximately equal anyway),

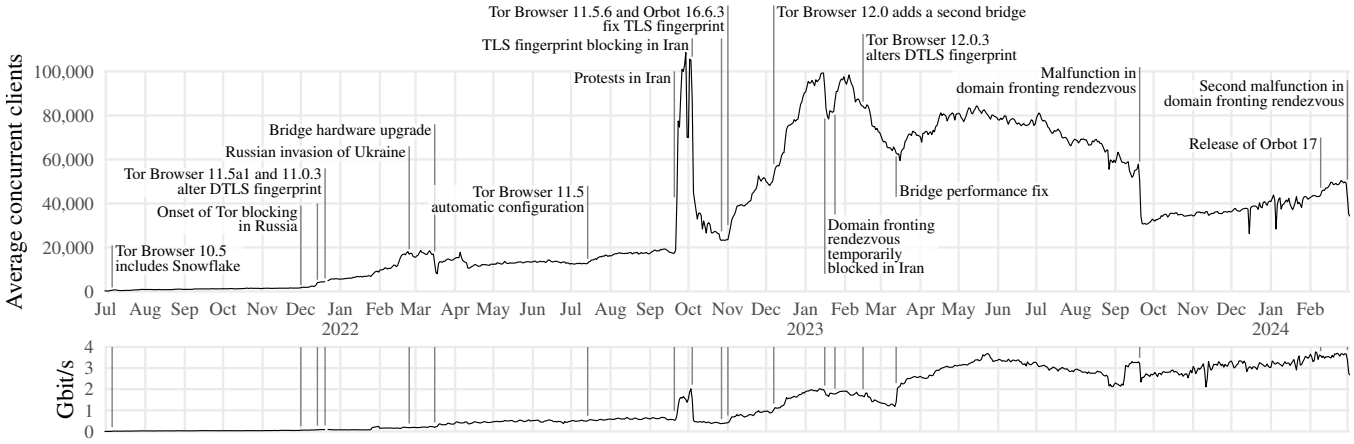

Figure 5: Estimated average concurrent Snowflake clients and consumed bandwidth by day. The values at the far left end of the graph, in early July 2021, are about 200 users and 2.7 Mbit/s.

after subtracting bytes spent processing directory requests (which are less than 1% of total bandwidth).

Snowflake shipped in the alpha release series of Tor Browser for years before entering the stable series. The first releases were for GNU/Linux in Tor Browser 7.0a1 on 2017-01-24 [7] and for macOS in Tor Browser 7.5a4 on 2017-08-08 [8]. Here we encountered difficulties with the Chromium-derived WebRTC library we had used up to that point that prevented us from making releases for other platforms. We were able to resume making progress after switching to Pion WebRTC [29] in 2019. Snowflake for Windows was released in Tor Browser 9.0a7 on 2019-10-01 [9], and for Android in Tor Browser 10.0a1 on 2020-06-02 [10].

Snowflake was then available on every platform supported by Tor Browser, but was not yet comfortably usable. Two important parts were missing: a lack of NAT type matching (Section 2.2) meant that a client could not always connect to its assigned proxy; and no persistent session state (Section 2.3) meant that even if a proxy connection were successful, the client's session would end once its first proxy disappeared. For these reasons, by early 2020, the 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.[11] The client part of NAT behavior detection was released with Tor Browser 10.0a5 on 2020-08-19 [12], and proxy support was added on 2020-11-17 [13]. With these changes, Snowflake became practical to use for daily browsing, and the number of clients began to grow into 2021.

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.[14] Snowflake graduated to Tor Browser's stable series in Tor Browser 10.5 on 2021-07-06 [15], becoming a third built-in circumvention option alongside meek and obfs4. This is the first event marked in Figure 5. Being in the stable release series meant Snowflake was easily available to all Tor users, not only a self-selected group of alpha testers. The number of concurrent clients increased steadily over the next five months, reaching almost 2,000 by December 2021.

Censorship events may have the apparently contrary effects of either decreasing or increasing usage of a circumvention system. Usage decreases when the system itself succumbs to blocking; but increases when other, less robust systems are blocked, and users become concentrated onto the reduced number of systems that remain working. Two censorship events, one in Russia and one in Iran, resulted in large increases in the number of Snowflake users.

On 2021-12-01, some ISPs in Russia instituted measures to block most ways of accessing Tor, including Snowflake [44]. The measures varied in effectiveness; in the case of Snowflake, blocking was triggered by a feature of the DTLS handshake, which we were able to mitigate in new releases within a few weeks.[16] Over the next two months, the number of clients quadrupled; by May 2022, about 70% of Snowflake users were in Russia. The client count in Russia got an additional small boost starting on 2022-07-14, when Tor Browser 11.5 added a feature to automatically enable circumvention options when needed.[17] We will look at Russia in more detail in Section 5.1.

In reaction to protests that began on 2022-09-16 after the death of Mahsa Amini, the government of Iran imposed net-

[7] https://bugs.torproject.org/tpo/applications/tor-browser/20735
[8] https://bugs.torproject.org/tpo/applications/tor-browser/22831
[9] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/25483
[10] https://bugs.torproject.org/tpo/applications/tor-browser/30318
[11] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/33745
[12] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/34129
[13] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/40013
[14] https://github.com/guardianproject/orbot/releases/tag/16.4.0-RC-1-tor-0.4.4.62021-01-12
[15] https://blog.torproject.org/new-release-tor-browser-105
[16] https://bugs.torproject.org/tpo/applications/tor-browser-build/40393
[17] https://blog.torproject.org/new-release-tor-browser-115/

work shutdowns and additional blocking, severe even by the standards of a country already notorious for censorship [3]. Users turned to the circumvention systems that continued working despite the new restrictions, Snowflake among them. Adoption was rapid: on 2022-09-20, Iran accounted for 1% of Snowflake users; by 2022-09-24 it was 67%. The sudden influx of users had us scrambling for a few days to implement performance improvements. Two weeks later, on 2022-10-04, usage dropped almost as quickly as it had risen—the cause was the blocking, in Iran, of a TLS fingerprint used by the Snowflake client.[18] After we released fixes for the TLS fingerprinting issue, the client count began to recover. But in our haste to deploy performance optimizations in September, we inadvertently introduced a bug that actually harmed performance, becoming more severe at higher client counts.[19] This dragged the count down again, until we fixed the bug in mid-March 2023. (Umayya et al. happened to be doing performance tests of Snowflake and other circumvention systems at this time [37 §4.6]—their results independently confirm the reduced reliability of Snowflake before the performance bug was fixed.[20]) We will say more about Iran in Section 5.2.

For most of this history, the Snowflake bridge was a single server, which we upgraded and optimized as needed. As the bridge reached its hardware capacity, and performance improvements got harder to achieve, we deployed a second bridge to share the load. The challenges and design considerations of doing so are discussed in Section 4.4. The new bridge was made available in Tor Browser 12.0 on 2022-12-07 and Orbot 17 on 2024-02-09. In early 2024, the second bridge handled about 15% of users and 25% of bandwidth.

The drop in users on 2023-09-20 was not caused by any censor action; rather, it was an unexpected change in the cloud infrastructure we used for domain fronting rendezvous. The front domain we had been using changed its hosting to a different CDN, which caused client rendezvous messages to fail to reach the broker.[21] We made releases with alternative front domains[22], but it took time for users to adjust. Something similar happened on 2024-03-01, when the CDN we had been using in the default configuration of domain fronting rendezvous stopped supporting domain fronting.[23] It happened just before the deadline for this paper, so we cannot yet comment on any long-term effects, but Figure 5 shows an immediate decline in users and bandwidth of about 30%.

As of 2024-03-01, Snowflake had transferred a lifetime total of 15 PB of circumvention data. Here we mean goodput: Tor TLS traffic inside the tunnel, ignoring WebRTC, WebSocket, and KCP/smux overhead. At that time, around 1.2% of Tor users (25% of bridge users) were using Snowflake.

[18] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/40207
[19] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/40260
[20] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/40262
[21] https://forum.torproject.org/t/9346
[22] https://bugs.torproject.org/tpo/applications/tor-browser/42120
[23] https://bugs.torproject.org/tpo/anti-censorship/team/135

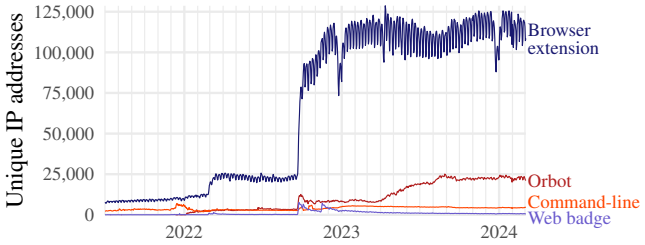

Figure 6: Unique proxy IP addresses per day, by proxy type. The two steps in the graph correspond to the invasion of Ukraine by Russia in February 2022, and network restrictions in Iran beginning September 2022, at which times there were campaigns to encourage running Snowflake proxies. Unknown proxy types (fewer than 50 instances) are not shown.

4.2 Number and type of proxies

Snowflake's effectiveness depends on its pool of proxies, of which there are several types. The primary type is the web browser extension, which runs in the background in volunteers' browsers and quietly serves clients. There is 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, rather than installing a browser extension. There is also a command-line implementation of the proxy that does not require a browser. This version is convenient to install on a rented VPS, for example. Long-term proxies running at fixed IP addresses are somewhat at odds with Snowflake's goal of proxy address diversity, but these standalone, command-line proxies are valuable because they tend to have less restrictive NATs, which makes them compatible with more clients. Finally, Orbot, as well as being able to *use* Snowflake for circumvention, can also *provide* Snowflake proxy service to others, a feature called "kindness mode."

We worked with the Tor Project's network health team to collect privacy-preserving metrics of client and proxy rendezvous interactions at the broker, and publish them in the same way as the bridge metrics of Section 4.1.[24] The metrics represent counts of proxy types, unique proxy IP addresses and geolocated country codes, NAT types (Section 2.2), and success rates of matching clients with proxies. Aggregate metrics are published at 24-hour intervals.[25] The broker does not record nor publish the IP addresses of clients or proxies. Metrics concerning client polls are rounded to multiples of 8, in accordance with established Tor Metrics practice.

Figure 6 shows the number of proxies by type. Web browser extension proxies predominate, representing about 80% of 140,000 daily IP addresses. For comparison, there were about 1,900 of the more traditional style of Tor bridge at this time. The difference in number is attributable to the relative ease of

[24] https://bugs.torproject.org/tpo/network-health/metrics/collector/29461
[25] https://metrics.torproject.org/collector.html#snowflake-stats

running a Snowflake proxy versus a Tor bridge—though the comparison is not quite direct, because Tor bridges have better defenses against enumeration than do Snowflake proxies.

It was not clear, at the outset, that it would even be possible to attract enough proxies to support a reasonable number of users with meaningful blocking resistance. Lowering the technical barriers to running a proxy was only part of it; it also took intentional advocacy and outreach. In the early days, circa 2017, the only consistent proxy support was a few command-line proxies, run by us for the benefit of alpha testers. The browser extension became available in mid-2019.[26, 27] Later in 2019, additional proxy capacity came when Cupcake, a browser extension for flash proxy with an existing user base, was repurposed for Snowflake.[28] Orbot's proxy feature was added in version 16.4.1 in February 2021.[29] (In Figure 6, Orbot is counted with command-line proxies until January 2022, when it got its own proxy type designation.)

It is worth reflecting on the popularity of the browser extension as compared to the web badge. A web badge had been envisioned as the main source of proxies in flash proxy, the idea being that people's browsers would automatically become proxies while on 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 opt-in. To run a proxy, a person must 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 limit 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 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.[30] Our technique was to record the set of proxy IP addresses seen by the broker over an interval of time, then compute the size of the intersection with other sets in nearby intervals, up to 40 hours later. Every hour, the broker recorded a snapshot of proxy IP addresses it had seen in the past hour. To avoid the risk of storing real proxy IP addresses, each snapshot was not a transparent list, but a HyperLogLog++ sketch [15], a probabilistic data structure for estimating the number of distinct elements in a multiset. Proxy IP addresses were hashed with a secret string (discarded at the end of the experiment) before being added to a sketch, to prevent their being recovered 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, namely the size of the intersection of two sketches, is computed using the formula $|X| + |Y| - |X \cup Y|$. This computation estimates how many proxy IP addresses are shared between two samples.

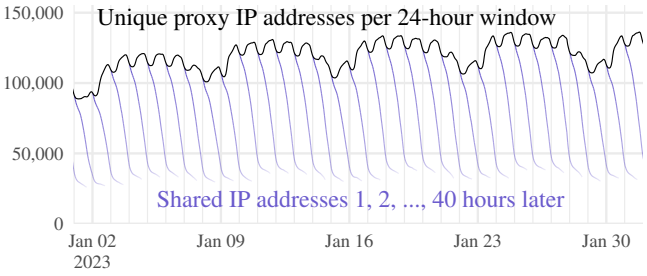

Figure 7: Proxy churn in January 2023. The dark upper line shows the number of unique proxy IP addresses in overlapping 24-hour windows. The lighter, descending lines indicate how many of the same IP addresses were in later 24-hour windows, at 1-hour increments up to 40 hours later. It takes about 20 hours for 50% of the proxy pool to turn over.

The broker recorded churn logs between 2022-07-22 and 2023-10-16. Figure 7 shows results from January 2023, which are representative. Proxy activity has a daily cycle, so we merged consecutive sketches into 24-hour windows. Starting from a reference window, we computed the size of its intersection with other 24-hour windows, offset by $+1, +2, \ldots, +40$ hours relative to the reference. After 1 hour, the shifted window had, on average, 97.3% of addresses in common with the reference; after 12 hours, the fraction had fallen to 68.8%; after 24 hours, 38.2%; and after 40 hours, 27.6%.

4.4 Multiple bridges

In the conceptual 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 reasons of scaling and performance, though, it can be useful for "the" bridge to be realized as multiple servers. Even with hardware upgrades and software optimizations, our single bridge began to hit performance limits in late 2022 (particularly after the events in Iran of Section 4.1), and we had to find ways of distributing bridge capacity to permit continued scaling. Our deployment now uses two bridges.

[26] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/30931#note_2593598
[27] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/30999#note_2593718
[28] https://github.com/glamrock/cupcake/issues/24
[29] https://github.com/guardianproject/orbot/releases/tag/16.4.1-BETA-2-tor.0.4.4.6
[30] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/34075

A multi-bridge system might be designed in many ways. In ours, the client tells the broker what bridge it wants to use in its rendezvous message. The broker conveys the choice to the proxy, and the proxy connects to the client's chosen bridge.[31] This may be compared to alternative designs in which the decision of what bridge to use is made by the broker or the proxy. Our design was largely dictated by technical constraints related to interfacing with Tor. Different designs may make sense with other deployments.

One minor consideration is the Turbo Tunnel layer. Recall from Section 2.3 that Snowflake maintains a virtual end-to-end session between the client and the bridge, independent of the temporary proxies. This is made possible by state stored at the bridge: a table of clients, reassembly buffers, transmission queues, timers, and so on. The state variables are not synchronized between bridges, which means a client session begun on one bridge must remain with that bridge, because no other has the context to make the packets of the session meaningful. When it is the client that chooses the bridge, this is easy to deal with: the client keeps its choice consistent in all rendezvous messages throughout a session. If the choice were instead made by the broker or proxy, it might be dealt with by hashing the client's session identifier string to an index, as long as the set of bridges does not change too frequently.

Another difficulty, harder to work around, has to do with Tor bridge authentication. 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 terminates the connection [5]. The Tor client can configure at most one identity per bridge; there is nothing like a certificate, for example, to indicate 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 chooses the bridge (and therefore knows what bridge identity to expect). While it would be possible to synchronize Tor identity keys across bridges, we preferred to keep keys independent, so that the effect of a security compromise at a bridge would be limited to that bridge only.

A client-chooses design risks misuse, if not handled carefully. Clients must not be able to cause proxies to connect to arbitrary destinations—otherwise proxies might be used in denial-of-service attacks, for example. To enforce this restriction, the client indicates its choice of bridge not by an IP address or hostname, but by an identity public key. The broker maps the identity to a WebSocket URL using its own database of known bridges, and rejects rendezvous messages that ask for 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 has a known suffix reserved for Snowflake bridges. So there are two independent safeguards against misuse.

[31] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/28651#note_2786323

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 always been a success, though, and here we also reflect on missteps and persistent challenges. The examples are taken from Russia, Iran, China, and Turkmenistan, and are selected for being significant and instructive. Common lessons are that communication with affected users is invaluable in quickly understanding and reacting to blocking; and that blocking resistance needs to be understood relative to a censor, because every censor's cost calculus is different.

Client count estimates come from the Tor Metrics descriptors described in Section 4.1. Country code assignments are drawn from the same IP geolocation database used by ordinary Tor relays and bridges—and the caveats of IP geolocation apply to the per-country estimates in this section. In particular, we have noticed that a fraction of clients that geolocate to the United States are likely actually to be in Iran, based on correlations of their dynamics with those of other clients that geolocate to Iran, and the timing of known political events.[32] The graphs of this section use raw geolocation results, with no attempt to adjust for errors.

Snowflake is blockable by any censor that is willing to block WebRTC. We would not try to argue otherwise. Indeed, we believe that the way to present a circumvention system is not to argue for its absolute unblockability, but to lay out what actions by a censor would be necessary to block it—or more to the point, *what sacrifices a censor would have to make* in order to block it. Advancing the state of the art of censorship circumvention consists in pushing blocking out of reach 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 [44]. The event was evidently coordinated and targeted, as it happened suddenly and affected many Tor-related protocols at once. 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. The blocking campaign was less than totally successful—one of its effects was to substantially increase the number of users accessing Tor via circumvention transports, Snowflake among them.

We benefited from established relationships with developers and users in Russia, one of whom, through manual testing,

[32] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/40207#note_2844116

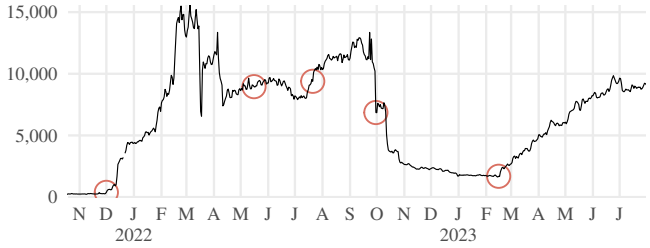

Figure 8: Snowflake users in Russia (average concurrent). Events discussed in the text are marked. The attempted blocking of Tor-related transports in December 2021 led to Snowflake's first surge in usage. The decrease in September–October 2022 coincided with an even larger influx from Iran.

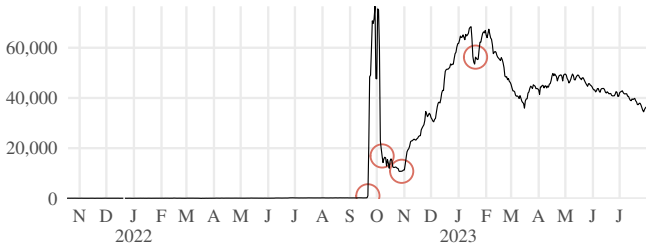

Figure 9: Snowflake users in Iran. Heightened censorship beginning in September 2022 caused Iran to become the single biggest source of Snowflake users. The drop in October 2022 was the result of TLS fingerprint blocking, which interfered with rendezvous and took some time to mitigate.

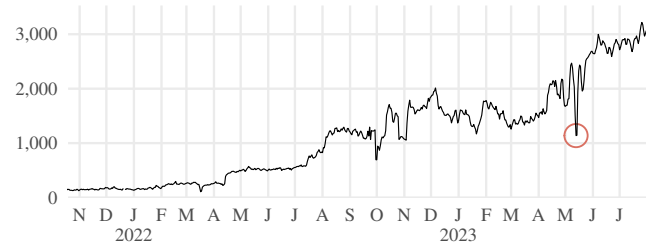

Figure 10: Snowflake users in China. Though no sustained blocking is evident, disruption of domain fronting rendezvous for three days in May 2023 briefly depressed user numbers.

Figure 11: Snowflake users in Turkmenistan. This graph shows an earlier range of dates than the other three. Though there have never been many Snowflake users in Turkmenistan, blocking events are evident on 2021-10-24 and 2022-08-03.

found the traffic feature that was being used to distinguish Snowflake. It was DTLS fingerprinting[33], 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's presence in Server Hello was in fact a bug[34]—but in any case, it afforded the censor a feature to distinguish DTLS connections with a Pion implementation in the server role from other forms of DTLS. The process of finding the flaw, fixing it, and shipping new releases of Tor Browser took a few weeks[35], after which the user count rose rapidly. From the beginning to the end of December 2021, the number of users in Russia grew from about 400 to over 4,000 (Figure 8). 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 had been discovered and documented by MacMillan et al. [22 §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, and 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 ISPs where the blocking rule was present, it did not block 100% of Snowflake connections, because of the way it targeted a quirk in Pion, and only in Server Hello. When the DTLS server role in the WebRTC data channel was played by a non-Pion peer, such as a web browser proxy, the feature was not present.

In May 2022 we got a report of a new detection rule, this time keying on not just the presence, but the *contents* of the supported_groups extension, at a byte offset suggesting that it now targeted the Client Hello message, not Server Hello.[36] The presence of a supported_groups extension in Client Hello is not unusual, but the specific groups offered by Pion's implementation differed from those of common browsers. Though we confirmed the existence of the 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 prepared a testing release in July 2022, the new rule had apparently been removed and replaced by another. We can only speculate as to why, but it may be that the old rule had too many false positives, or simply was not effective enough.

The detection rule that replaced supported_groups in Client Hello looked for the server sending a Hello Verify Request

[33] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/40014#note_2765074
[34] https://github.com/pion/dtls/issues/409
[35] https://gitlab.torproject.org/tpo/applications/tor-browser-build/-/merge_requests/375
[36] https://bugs.torproject.org/tpo/anti-censorship/censorship-analysis/40030

message.[37] 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 time containing a copy of the cookie [32 §5.1]. It is not an error to send Hello Verify Request (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. (At least, those in which a Snowflake client or command-line proxy took the DTLS server role.) This distinguisher, too, had been anticipated by MacMillan et al. in 2020 [22 §3]. The first reports of this blocking rule arrived in July 2022; but as you can see in Figure 8, 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 enormous increase 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 [38], after which the number of users in Russia began to recover.

The case of Snowflake in Russia illustrates some of the complexity of censorship measurement. The answer to a question like "Does Snowflake work in Russia?" is not always a simple yes or no. It may depend on the date, the ISP, and even details such 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 9. 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. The crypto/tls package of the Go programming language (in which the Snowflake client is written) may produce several slightly different TLS fingerprints, depending on hardware capabilities and how it is compiled.[39] It was one of these fingerprints that was blocked. Because the blocking rule was specific to one fingerprint, only some users were affected. Why would a censor block only one (even if the most common) of several TLS fingerprints? It may have been a simple oversight. On the other hand, it is not certain that this instance of TLS fingerprinting in Iran was meant for 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.

The fact that simple TLS fingerprinting worked to block Snowflake rendezvous was carelessness on our part. Having been aware of the possibility, we previously implemented TLS camouflage in the Snowflake client using uTLS—but failed to turn it on by default. Activating the feature required only a small configuration change[40], 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 9.

After we repaired 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 rendezvous front domain was blocked (by TLS SNI) in some ISPs between 2023-01-16 and 2023-01-24 [41], which we confirmed using data from the censorship measurement platform OONI. A reduction in users is visible at this time. AMP cache rendezvous continued to work. OONI measurements in the weeks after the block was lifted showed additional sporadic failures to connect to the front domain. If these were further attempts at blocking, they did not have much of an effect.

5.3 Blocking in China

The graph of users from China, Figure 10, does not show any drastic changes like those we have seen so far. There is a modest but respectable number of Snowflake users in China. Though there has been no sustained interference, we have seen some evidence of short-term or tentative blocking attempts.

In May 2019, when Snowflake was still in alpha release, a user in China reported a failure to connect. Investigation revealed that the cause was blocking of the IP address of the few proxies that existed at the time.[42] The STUN exchange worked, and rendezvous completed successfully, but the client and proxy could not establish a connection. We experimented with running a proxy at a previously unused IP address: clients in China could connect when they were assigned that proxy by the broker. This was before the web browser extension proxy existed, when the only consistent proxy support was a few standalone proxies running at a static IP address. It stopped being a problem as the proxy pool grew in size.

That same month, we noticed blocking of the default STUN server, of which there was only one at the time.[43] The solution was to add more STUN servers[44], and select a subset of them on each rendezvous attempt[45]. Curiously, it seems that when the STUN server was blocked, the standalone proxies that had been blocked earlier in the month became unblocked.[46]

The next incidents we are aware of did not occur until 2023. On May 12, 13, and 14, a few users reported problems with domain fronting rendezvous.[47] We could not get systematic

[37] https://bugs.torproject.org/tpo/anti-censorship/censorship-analysis/40030#note_2823140
[38] https://gitlab.torproject.org/tpo/applications/tor-browser-build/-/merge_requests/637
[39] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/40207#note_2844163
[40] https://gitlab.torproject.org/tpo/applications/tor-browser-build/-/merge_requests/540
[41] https://bugs.torproject.org/tpo/anti-censorship/team/115#note_2873040
[42] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/30350#note_2593274
[43] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/30368#note_2593357
[44] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/30579
[45] https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/merge_requests/7
[46] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/30368#note_2593360
[47] https://bugs.torproject.org/tpo/anti-censorship/censorship-analysis/40038

measurements, but it appeared that censorship was triggered by observing 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 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 rendezvous with multiple bridges (Section 4.4), though such a policy would certainly also affect a large number of non-Snowflake connections. The user count from China was about halved during those three days, an effect that is visible in Figure 10. On 2023-05-15, the blocking went away and the user count returned to what it had been.

Also in May 2023, one user reported apparent throttling (artificial reduction in speed by packet dropping) of TLS-in-DTLS connections, based on packet size and timing features.[48] Such a policy would affect Snowflake, because of the fact that it transports Tor TLS inside DTLS data channels. Reportedly, padding the first few packets would prevent throttling (a possible counterexample to our claims about traffic analysis in Section 3). Our own speed tests run at the time did not show evidence of throttling, with or without added padding.[49] 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 shown in Figure 11. We found a variety of causes: domain name blocking by DNS and TCP RST injection, and blocking of certain UDP port numbers commonly used for STUN.

Turkmenistan is a particularly challenging environment for circumvention. Though relatively unsophisticated, the censorship there is more severe and indiscriminate than in the other places we have discussed. Only a small fraction of the population has access to the Internet at all, 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 partially, and after protracted effort.

The drop on 2021-10-24 was caused by blocking of the default broker front domain.[50] We determined this by taking advantage of the bidirectionality of the Turkmenistan firewall. Nourin et al. [25 §2] provide more details; here we will state just the essentials. 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 of users from May to August 2022 was caused by a partial unblocking of the broker front domain on 2023-05-03. We realized this only in retrospect, after examining data from Censored Planet [34], a censorship measurement platform that, at that time, had continuous measurements of the domain from one autonomous system in Turkmenistan. On that date, there was a shift from RST responses to successful TLS connections. DNS measurements were not available at the moment of the shift, but they, too, showed no signs of blocking after that date. Evidently, some users were able to reach the broker, in those days. But the unblocking must not have been everywhere, because as late as 2022-08-18, users reported that RST injection was still in place for them (though DNS injection had stopped).

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. Testing revealed blocking of the default STUN port, UDP 3478. A client that cannot communicate with a STUN server cannot find its ICE candidate addresses (Section 2.2), without which most WebRTC proxy connections will fail. (The exceptions are proxies without NAT or ingress filtering. While there are some such proxies, censorship in Turkmenistan also blocks large parts of IP address space, including the data center address ranges where those kinds of proxies tend to run.) As chance would have it, the NAT behavior discovery feature we rely on for testing client NATs requires STUN servers to open a second, functionally equivalent listening socket on a different port [21 §6], commonly 3479. Changing to those alternative port numbers enabled some users to 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 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 is what precipitated the drop seen on that date in Figure 11.

The blocking techniques described in this section are crude, surely resulting in significant overblocking—but they nevertheless offer greater challenges to circumvention than the more considered blocking of, say, Russia and Iran. We remark on this to make the point that blocking resistance cannot be defined in absolute terms, but only relative to a particular censor. 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 priori-

[48] https://github.com/net4people/bbs/issues/255

[49] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/40251#note_2906723

[50] https://bugs.torproject.org/tpo/anti-censorship/censorship-analysis/40024

tize political control over a functional network, to an extreme degree. To paraphrase one of our collaborators: "What they have in Turkmenistan can hardly be called an Internet."[51] In a network already damaged by oppressive policy, the additional harm caused by the clumsy blocking of this or that circumvention system is comparatively small. This shows the sense in which a resource-poor censor can "afford" certain blocking actions that a richer, more capable censor cannot.

6 Future work

We designed Snowflake to be useful today, as well as resilient to anticipated future censorship attacks. Here we list some research questions that affect Snowflake and circumvention more broadly. The uniqueness and scale of Snowflake's deployment provide a platform to help explore these questions.

What could alternative Snowflake deployments look like? A natural extension of Snowflake would be to base it on a system other than Tor, such as an ordinary VPN. Tor has attractive privacy benefits and a convenient framework for developing circumvention modules, but it also has relatively low speed and lacks support for non-TCP applications. The prospect of multiple Snowflake deployments raises the question of how the proxy pool should be managed. Building Snowflake's population of proxies has been an undertaking in itself—it would be a regrettable duplication of effort if every project had to repeat the process from scratch. Rather than retrofit our existing Tor-based proxies, a next-generation proxy pool might be designed from the ground up with multiple cooperating projects in mind. There is also the question of incentives: while some proxy operators may be happy to donate bandwidth to a free-to-use project like Tor, they may need more motivation to help a commercial VPN, for example.

How might proxy enumeration attacks be inhibited? The size (Section 4.2) and churn (Section 4.3) of the Snowflake proxy pool are obstacles to comprehensive blocking. However, it is worth considering the maximum impact of enumeration attacks, and ways to mitigate them while preserving usability.

What is a good (family of) traffic shapes? Snowflake does not yet attempt to shape its traffic analysis features (Section 3); but if it were to, what shape should it use? This consideration goes beyond just Snowflake. Frameworks for proposing and evaluating traffic shaping techniques are under-explored.

Can traffic splitting benefit performance or resistance to blocking? The Turbo Tunnel reliability layer of Section 2.3 lets us join a sequence of proxies into a session. In principle, it also makes it possible to split traffic over many proxies, not just sequentially, but simultaneously—something like multipath TCP. Sequence numbers, retransmissions, and reassembly would ensure a reliable stream, even when proxies have different lifetimes and performance characteristics. Traffic splitting could reduce the performance impact of a slow proxy, and eliminate the brief pause for re-rendezvous that now occurs between consecutive proxies. Our initial experiments did not show enough benefit to justify the change, though it may be a matter of tuning.[52] And of course, even if there are performance and usability benefits, 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. Code and data to reproduce this paper and its figures are at https://archive.org/details/snowflake-paper.

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, Diogo Barradas, Griffin Boyce, Anthony Chang, 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, Michael Pu, Kieran Quan, Sukhbir Singh, Aaron Swartz, ValdikSS, Vort, Andrew Wang, Philipp Winter, WofWca, Yi Wei Zhou, Censored Planet, the Counter-Power Lab at UC Berkeley, Greenhost, Guardian Project, Mullvad VPN, the Net4People BBS and NTC forums, OONI, the Open Technology Fund, Pion, the Tor Project, financial donors, and the volunteers who run Snowflake proxies. This work was supported in part by DARPA under Contract No. FA8750-19-C-0500.

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.

[51] https://bugs.torproject.org/tpo/anti-censorship/censorship-analysis/40024#note_2889792

[52] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/25723#note_2718643

[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. Negotiating and initializing channels. In *Tor Protocol Specification*. February 2024. https://gitlab.torproject.org/tpo/core/torspec/-/blob/29e445bd6e9efe82367b8a2b09a6c6aa0bc92b7b/spec/tor-spec/negotiating-channels.md.

[6] Donald E. Eastlake 3rd. Transport Layer Security (TLS) extensions: Extension definitions. RFC 6066, January 2011. https://www.rfc-editor.org/info/rfc6066.

[7] 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.

[8] 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/.

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

[10] 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.

[11] 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/.

[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] 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/.

[16] Jordan Holland, Paul Schmitt, Nick Feamster, and Prateek Mittal. New directions in automated traffic analysis. In *Computer and Communications Security*. ACM, 2021. https://dl.acm.org/doi/10.1145/3460120.3484758.

[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] 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.

[22] 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.

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

[24] 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/.

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

[26] 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 2024-03-05].

[27] 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.

[28] 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.

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

[30] 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.

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

[32] 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.

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

[34] 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.

[35] Tor Metrics. Reproducible metrics. https://metrics.torproject.org/reproducible-metrics.html [cited 2024-03-05].

[36] Michael Carl Tschantz, Sadia Afroz, Anonymous, and Vern Paxson. SoK: Towards grounding censorship circumvention in empiricism. In *Symposium on Security & Privacy*. IEEE, 2016. https://internet-freedom-science.org/circumvention-survey/.

[37] Zeya Umayya, Dhruv Malik, Devashish Gosain, and Piyush Kumar Sharma. PTPerf: On the performance evaluation of Tor pluggable transports. In *Internet Measurement Conference*. ACM, 2023. https://ptperf.github.io/.

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

[39] uProxy v1.2.5 - design doc. https://docs.google.com/document/d/1t_30vX7RcrEGuWwcg0Jub-HiNI0Ko3kBOyqXgrQN3Kw [cited 2024-03-05]. Archived at https://archive.org/details/uProxy-Design-Doc-v1.2.5.

[40] Afonso Vilalonga, João S. Resende, and Henrique Domingos. TorKameleon: Improving Tor's censorship resistance with K-anonymization and media-based covert channels. In *Trust, Security and Privacy in Computing and Communications*. IEEE, 2023. https://arxiv.org/abs/2303.17544.

[41] Ryan Wails, George Arnold Sullivan, Micah Sherr, and Rob Jansen. On precisely detecting censorship circumvention in real-world networks. In *Network and Distributed System Security Symposium*. The Internet Society, 2024. https://www.robgjansen.com/publications/precisedetect-ndss2024.html.

[42] Yibo Xie, Gaopeng Gou, Gang Xiong, Zhen Li, and Mingxin Cui. Covertness analysis of Snowflake proxy request. In *Computer Supported Cooperative Work in Design*. IEEE, 2023. https://ieeexplore.ieee.org/document/10152736.

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

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