Fingerprint-resistant DTLS for usage in Snowflake

Theodor Signebøen Midtlien

David Palma
Norwegian University of Science and Technology
Trondheim, Norway

Abstract

Internet censorship circumvention requires continuous effort and attention in order to achieve its goals. This paper aims at making the Pion DTLS library used in Snowflake less prone to fingerprinting and blocking. We developed a tool for analyzing and passively identifying field-based fingerprints of DTLS, validated using a dataset containing known fingerprints. Our findings revealed that the extensions field is particularly vulnerable to identification. To address this, we propose and implement *covertDTLS* which extends the Pion DTLS Go library with handshake hooking to offer mimicry and randomization features inspired by uTLS. In addition, we created a continuous delivery pipeline to generate fresh DTLS-WebRTC handshakes based on popular browsers, allowing researchers to monitor changes and ensuring that mimicking remains up-to-date. Our results indicate that mimicking and randomization are effective countermeasures against allowlisting and blocklisting respectively. We further analyze the evolution of DTLS-WebRTC handshakes collected from browsers over a year and their impact on Snowflake's distinguishability. To fully understand the impact of our proposed solution, we also deployed standard Snowflake proxies and improved ones using our fingerprint-resistant DTLS library, and report our findings. Our observations suggest that the prompt adoption of DTLS 1.3 is necessary to keep pace with browser updates, and our fingerprint-resistant library demonstrated stability when mimicking DTLS 1.2 handshakes, but less so with the randomization approach. These obtained results suggest that our modifications to Snowflake effectively reduce the fingerprintability of DTLS traffic, enhancing its capability to bypass censorship. However, we also argue that continuous monitoring and prompt adaptation to evolving Internet protocols and applications is essential for the anti-censorship community.

Keywords

Censorship Circumvention, DTLS, Network Protocol Fingerprinting, Snowflake

1 Introduction

The Internet facilitates global sharing of information and ideas, such freedom of opinion and expression are protected by Article 19 of the United Nations Universal Declaration of Human Rights (UDHR) [20]. However, there are diverse attempts by censors (e.g. governments, institutions, and service providers) to violate these rights by regulating, monitoring, or, in some cases, by entirely stifling access to the open Internet [22, 23, 30]. This phenomenon,

known broadly as Internet censorship, represents both a technical challenge and a significant global societal concern, impacting free speech and human rights at large. Internet access is even being censored in regions often considered "*free*", such as the European Union (EU) [29, 32].

One Internet censorship circumvention system that is commonly used in the Tor Browser and Orbot is Snowflake[1] [7]. Operating on the principle of volunteerism and decentralization, Snowflake employs ephemeral proxies run by volunteers using Web Real-Time Communication (WebRTC) [5] peer-to-peer connections. So far, censors have not shown willingness to block WebRTC as a protocol [7], which allows Snowflake to blend in with the long tail of other WebRTC traffic.

No censorship circumvention system is perfect, and Snowflake has been successfully blocked at multiple occasions [7]. An example of this is Russia blocking Snowflake in May of 2022 [1]. This was accomplished by fingerprinting unique *ClientHello* messages associated to Snowflake using the Datagram Transport Layer Security (DTLS) protocol, which is used by WebRTC. This method has previously been discussed in literature and was a known attack vector [18]. Reactive measures have been deployed by the Tor project to remove the distinguishing *ClientHello* fingerprint in the DTLS implementation by Pion[2], but these were not integrated in Snowflake and other weaknesses may still exist [2].

Transport Layer Security (TLS), being a similar protocol to DTLS, can provide some insights into fingerprint resistance, as it has been more widely studied for use in censorship circumvention. Frolov and Wustrow [14] found multiple ways of fingerprinting TLS, including the *ClientHello* method used to block DTLS in Snowflake. To handle this problem, the researchers developed a library called *uTLS*[3] that aims to protect against fingerprinting. However, no such library exists for DTLS, which concerns the team behind and actively developing Snowflake.

This paper extends our previous work (Midtlien's Master's thesis at the Norwegian University of Science and Technology) on 'Reducing distinguishability of DTLS for usage in Snowflake' [19], which is presented in Section 3. We developed *dfind*[4], a tool for analyzing and passively discovering field-based fingerprints of DTLS. This tool was validated using a dataset with known fingerprints, and found that the extensions field was especially vulnerable for identification. To combat such fingerprints, we propose and implement a Go library inspired by uTLS that we named *covertDTLS*[5]. The module extends the Pion DTLS library with handshake hooking to offer mimicry and randomization features. To ensure that mimicking remains up-to-date, we developed a novel continuous delivery workflow for generating fresh DTLS-WebRTC handshakes based

---

[1]https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake
[2]https://github.com/pion
[3]https://github.com/refraction-networking/utls
[4]https://github.com/theodorsm/dfind
[5]https://github.com/theodorsm/covert-dtls

on popular browsers. We concluded that mimicry and randomization are effective countermeasures against passive, stateless, and field-based fingerprinting.

We further explore how DTLS fingerprints from popular browsers (Firefox and Chrome) changed over the course of a year and how this affects the distinguishability of Snowflake. We deployed standard Snowflake proxies and improved ones using our fingerprint-resistant DTLS library. Traffic from these deployments is analyzed and we present lessons learned that may contribute to finding new areas of research on Internet anti-censorship.

2 Background and related work
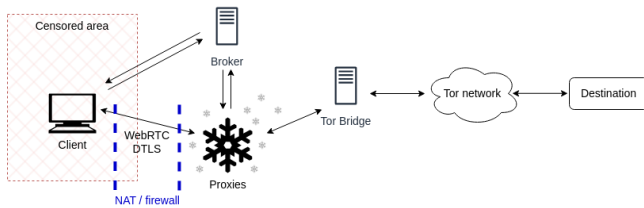
2.1 Snowflake

Figure 1: Architecture of Snowflake

The Snowflake circumvention system comprises three primary participants: the client, proxies, and a broker. Its architecture, illustrated by Figure 1, shows the essential components and communication pathways. The client is an individual utilizing Snowflake within a region where a censor blocks traffic to specific destinations (IP addresses). To circumvent these IP blockages, the client engages a broker to identify an available proxy in a process known as rendezvous. The broker matches the client with an idle proxy, which allows routing the client's traffic through an encrypted WebRTC data channel. Proxies are managed by volunteers with unblocked IPs, granting access to the open internet.

There are three main types of proxies: webextensions, badge and standalone. The webextension is a browser extension that volunteers can run that opens a WebRTC connection in the background using the browser networking stack. The badge can be embedded in websites and visitors can run a Snowflake proxy using the browser networking stack as long as the tab is open. Finally, the standalone version is a Go binary that can be run on a desktop or server from the command line. This type uses the same Pion WebRTC/DTLS networking stack as the Snowflake client.

To initiate a connection with the broker, the client must use an indirect, unblockable channel to bootstrap into Snowflake. Three methods are supported for rendezvous: domain fronting[6], Accelerated Mobile Pages cache and Simple Queue Service [7]. Once the indirect channel is established, the client communicates with the broker, which pairs the client with an idle proxy from its pool, based on self-reported Network Address Translation (NAT) types. The broker then facilitates the exchange of Session Description

---

[6]It is worth noting that this approach requires anti-censorship teams to constantly update the content providers they use for domain fronting, as many of the providers are stopping their support of the service

Protocol (SDP) [6] offers and answers between the client and proxy, as specified by WebRTC.

Following rendezvous, the client and proxy must navigate NAT traversal during the connection establishment phase. Devices behind NATs and firewalls typically only allow outgoing connections initiated by the client. To address this challenge, WebRTC employs the Interactive Connectivity Establishment (ICE) [17] procedure, which enables direct communication channels through NATs and firewalls. The ability to establish a connection between a client and proxy during the ICE procedure depends on using Session Traversal Utilities for NAT (STUN) [21] and servers supporting Traversal Using Relays around NAT (TURN) [24]. Upon successfully establishing a connection, the client and proxy can exchange traffic.

The final phase of Snowflake involves data transfer, which includes a persistent session layer and an ephemeral data channel. A persistent session is maintained using Turbo Tunnel [11], which adds sequence numbers and acknowledgments to the data exchanged between the client and a bridge. This ensures that if the current proxy becomes unavailable, the data will be retransmitted through a new proxy. For the ephemeral channel, WebRTC data channels are used, facilitating the transmission of encrypted and integrity-protected data via DTLS.

2.2 DTLS

DTLS is a protocol designed by the Internet Engineering Task Force (IETF) to provide secure communication for datagram-based applications, similar to how Transport Layer Security (TLS) secures applications over TCP. DTLS comprises two primary components: the handshake and the record layer. The DTLS handshake is responsible for negotiating cryptographic algorithms and keys between the client and server. Once these parameters are established, the DTLS record layer takes over, encapsulating the data from the upper layers into encrypted records that are transmitted over UDP.
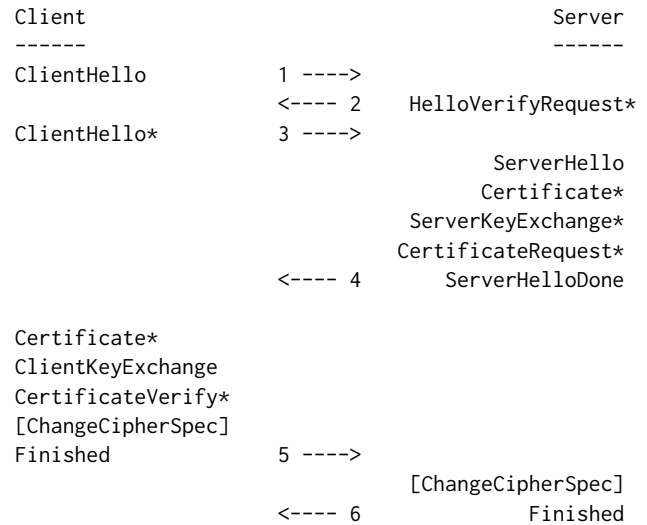
```
Client                                        Server
------                                        ------
ClientHello          1 ---->
                     <---- 2      HelloVerifyRequest*
ClientHello*         3 ---->
                                              ServerHello
                                             Certificate*
                                       ServerKeyExchange*
                                       CertificateRequest*
                     <---- 4        ServerHelloDone

Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished             5 ---->
                                        [ChangeCipherSpec]
                     <---- 6                    Finished
```

Figure 2: Messages for the full DTLS 1.2 handshake. Optional messages are indicated with an asterisk.

DTLS has undergone several iterations to enhance security and performance. DTLS 1.2 built upon TLS 1.2 was introduced in RFC 6347 [25] in 2012 and is still the most common version deployed on the internet [10]. The most recent version is DTLS 1.3, based on TLS 1.3. It features a streamlined handshake that reduces the number of round-trips, lowering latency. This version also provides default forward secrecy and elimination of outdated cryptographic algorithms. DTLS 1.3 became a standard in 2022 with RFC 9147 [27], but has seen little adoption.

Figure 2 shows the full DTLS 1.2 handshake between the two parties to negotiate security parameters, authenticate each other, and establish shared secrets for encrypted communication.

The handshake begins with the client sending a *ClientHello* message to the server, containing the protocol version, a randomly generated number (*ClientRandom*), *SessionID*, supported cipher suites, compression methods, and any relevant extensions such as Application-Layer Protocol Negotiation (ALPN) and *Supported-Groups*.

To prevent denial-of-service attacks from spoofed IP addresses, the server may respond with a *HelloVerifyRequest* message. It includes a stateless cookie created as a HMAC of a secret, the client parameters and IP. The client must echo back cookie in a subsequent *ClientHello* message. This step is optional but recommended to verify the client's reachability and mitigate resource exhaustion risks.

Upon receiving the *ClientHello*, the server responds with a *ServerHello* and optional messages such as *Certificate*, *ServerKeyExchange* and *CertificateRequest*. The *ServerHello* message includes the server's chosen protocol version, a randomly generated number (*ServerRandom*), *SessionID*, chosen cipher suite, compression method, and any relevant extensions. The *ServerHelloDone* message indicates the end of the server's initial handshake messages.

The handshake concludes with sending the *ChangeCipherSpec* and *Finished* messages. The *ChangeCipherSpec* message signifies a switch to the newly negotiated cipher suite and keys. The *Finished* message is a hash of the entire handshake encrypted with the new session keys by the server. The client and server can now send encrypted records to each other.

To provide extra flexibility, DTLS utilizes extensions in the *ClientHello* and *ServerHello* messages. The extension field can be of variable size, with a maximum size of 2 bytes, allowing different amounts of extensions to be in any order. This is a way to negotiate additional features without altering the core protocol. These extensions are specified in various RFCs, often for both TLS and DTLS. The Encrypted Client Hello (ECH) is an extension currently in draft for TLS/DTLS 1.3 that aims to enhance privacy and security by encrypting parts of the *ClientHello* message [26].

2.3 Fingerprinting

Network protocol fingerprinting is the process of identifying and classifying network protocols based on their unique characteristics or patterns, similar to how fingerprints uniquely identify individuals. These protocol-specific patterns enable the detection and analysis of the communication protocols used within a network.

Guoqiang Shu and David Lee introduced a formal methodology for network protocol fingerprinting, outlining a taxonomy that addresses the challenges of fingerprinting through three main components: active and passive experiments, fingerprint discovery, and fingerprint matching [28]. Active fingerprinting involves engaging with the target system by sending specific probes or queries and analyzing the responses to gather information about the protocol and its implementation. While this method can be intrusive and may cause some disruption to the target system, it is highly effective in extracting detailed protocol information. Passive fingerprinting relies on observing and analyzing network traffic patterns without direct interaction with the target system. This approach is less intrusive and does not risk disrupting the target system. However, it may be less accurate than active fingerprinting. Passive fingerprinting utilizes deep packet inspection (DPI) to examine protocol fields or analyze statistical traffic patterns.

Fingerprint discovery involves systematically uncovering a fingerprint for an unknown implementation. This process gathers comprehensive information to create a unique identifier for the protocol. Fingerprint matching is the process of comparing collected fingerprints to determine if they originate from the same protocol implementation. This can be done through exact one-to-one mapping or probabilistically, assessing the likelihood that two fingerprints correspond to the same implementation.

Fifield and Epner [12] are the first publicly to explore ways of fingerprinting parts of the Snowflake system. The authors conducted a manual analysis of different WebRTC applications to identify features that could be used to fingerprint them. Their findings revealed significant fingerprinting potentials in the DTLS and STUN/TURN protocols used by WebRTC, including differences in cipher suites, extensions, and certificate details.

The work of MacMillan et al. [18] is the most prominent work on detecting Snowflake traffic by fingerprinting DTLS handshakes. They collected the largest dataset to date with 6,500 handshakes of different WebRTC based applications. They extracted features from DTLS fields and performed classification with the random forest machine learning algorithm. Although their dataset is publicly available, their classification software is not.

Chen et al. [8] and Holland et al. [15] also take a machine learning approach to discovering Snowflake. Both use the MacMillan et al. dataset of DTLS handshakes and claim high accuracy of detection. To our knowledge these approaches has not been adopted in the real world.

For fingerprinting, we consider only the DTLS handshake to be in scope for this paper, not the encrypted record layer. We will also not explore traffic pattern (flow) analysis (e.g. timings, packet size, speed) such as Wang et al. [34] and Xie et al. [37]. Even though they claim promising results, we believe it is difficult to know if the statistical properties are of the DTLS implementation or the network itself. Such approaches require highly-controlled environments to not fingerprint the underlying network. Bocovich et al. [7] also warn against this, as traffic analysis attacks have historically been overestimated due to un-realistic base rates [33].

2.4 Fingerprint resistance

There are two main obfuscation techniques used in practice to combat fingerprinting: mimicking and randomization [13].

Mimicking, also known as mimicry or steganography, aims to replicate the behavior of a protocol. The goal is to make it challenging to distinguish between the genuine protocol and the obfuscating protocol. Houmansar et al. [16] argue that mimicking application layer protocols is particularly challenging and fundamentally flawed, a criticism summarized by the phrase "The parrot is dead".

Randomization, often referred to as polymorphism, involves implementing random protocol features to make the traffic appear dissimilar to any protocol or pattern that a censor might block. The objective is to eliminate all statistical characteristics, causing the traffic to resemble "junk" data. However, this approach can be ineffective if the censor employs allowlist blocking, as the traffic would not match any approved protocols.

While Snowflake uses DTLS, the similarities with TLS make it worth exploring the realm of TLS fingerprinting, and existing mitigation techniques. For example, Sergey Frolov and Eric Wustrow developed the fingerprint-resistant uTLS library [14]. The library employs multiple techniques for obfuscating traffic: low-level access to the handshake, randomized *ClientHello* fingerprint, mimicking *ClientHello* messages of other implementations and use of multiple fingerprints. For their mimicked fingerprints, they collected fingerprints from real browsers and rely on volunteers to update future fingerprints in the library.

2.5 Capabilities of censors

Tschantz et al. [30] did a study in 2016 to ground the evaluation of circumvention approaches in empirical observations of real censors. They found that censors prefer simple cost-effective solutions, with mostly passive monitoring (e.g. DPI) and some active probing. They suggest that censorship circumventors should concern themselves more with low-cost exploits. We assume that a censor will prefer simple, stateless and deterministic solutions to perform detection and blocking. Our focus is to prevent such low-hanging fruit.

We further assume that a censor prefers passive fingerprinting over active probing. The Great Firewall of China have been deploying active probing for *Shadowsocks* [4, 36] and *obfs3* [9]. This is probably due to it not being possible to perform passive field based fingerprinting, because of the randomized nature of the protocols. Active fingerprints are hard to discover, because you have to find a bug or side-channel of the protocol. A censor would also have to deploy infrastructure to send the fingerprint payloads. There have not been any active attacks on Snowflake, as far as we know.

3 Fingerprinting-resistant Snowflake

Our primary objective is to reduce the fingerprintability of DTLS when utilized within Snowflake. Several approaches can be considered to achieve this objective, such as protocol obfuscation or traffic shaping to change data packet flows to mimic regular, noncensored traffic. Traffic shaping is already supported in Snowflake, but it has not been put to use yet [7]. Thus we looked towards the DTLS handshake process that has been blocked by fingerprints before. Apart from encrypted *ClientHello* (ECH) messages, which is only available in version 1.3 of DTLS, other techniques must be developed to improve Snowflake's resistance to fingerprinting.

To better understand the problem at hand, a DTLS fingerprint discovery tool was created to simulate a censor discovering unique fingerprints of Snowflake. The tool, *dfind*, parses DTLS handshakes and extracts features such as length, cipher suites, and extensions bytes within the hello messages. To discover fingerprints, the database is queried as part of an automatic analysis step with two routines. One routine finds unique values of fields which are identifying for a certain implementation type, the other finds similar hex-strings in the extensions of each type, so that they can be further manually analyzed. This artifact was validated against the dataset by MacMillan et al. [18] and found that extensions was the most prominent feature for fingerprinting [19], as multiple researchers have pointed out before [12, 18].

Considering the MacMillan et al. [18] dataset to be outdated, we developed a continuous deployment (CD) setup with GitHub Actions workflows for generating fresh DTLS-WebRTC handshakes with the most recent version of popular browsers[7]. The workflow pulls the newest stable version of Firefox and Chrome on Ubuntu every day. We use Selenium to automate the browsers to create a simple WebRTC datatunnel simultaneously as we run a packet capture. The captures are filtered for DTLS handshakes and committed to the repository. This can be used for mimicking DTLS handshake messages and as a dataset for further research (available in the *covertDTLS* GitHub repository). The pipeline can keep the mimicking functionality up-to-date with popular browsers (that usually silently update "themselves"). Validation included comparing the handshakes to manually generated ones, testing for consistency, and using the fingerprint discovery tool to compare the automatically generated handshakes. This public corpus of DTLS handshakes will grow over time and facilitates studying the evolution of different implementation as we do in section 4.1. A censor would have access and the ability to do large scale collection of DTLS traffic, so a publicly available dataset is valuable for researchers to keep up with their adversaries.

The final and most central of our contributions is *covertDTLS*, a DTLS Go module that extends the Pion DTLS library to offer fingerprint-resistance features inspired by *uTLS*. To support this, we contributed new capabilities to the Pion DTLS codebase, including hookable handshake logic. These hooks allow external modules to intercept and modify *ClientHello*, *ServerHello* and *CertificateRequest* messages immediately before transmission. This preserves the Pion's API while enabling external modules like ours to extend handshake behavior without maintaining a forked version of upstream, like *uTLS* does.

covertDTLS implements two main fingerprint-resistance techniques for *ClientHello* messages (with other messages planned):

- Mimicry: This mode loads a target fingerprint as a hexstring (e.g., from our generated browser handshakes) and reconstructs the *ClientHello* message to match its fields, including the cipher suite list, compression methods and extension bytes. It aims to make the traffic indistinguishable from the browser WebRTC implementations using passive field based fingerprinting. We also expose the method to load arbitary fingerpints to let users of our library bring their own DTLS fingerprints they would like to mimic.

---

[7]https://github.com/theodorsm/covert-dtls/tree/main/.github/workflows

- Randomization: In this mode, we modify the original *ClientHello* with a combination of order permutation and picking random values for cipher list and extensions, effectively generating unique messages that evade deterministic matching.

Our fingerprint discovery tool was also employed to validate that the library did not produce any distinguishable fingerprints when mimicking. We used *snowbox*[8] to capture traffic from regular Snowflake and a modified version using *covertDTLS*, comparing the traffic to the DTLS-WebRTC corpus from MacMillan et al. combined with captures of newer browser versions. Using *dfind*, we found distinguishing fingerprints for regular snowflake traffic, but not for Snowflake using mimicking [19]. This suggest that our mimicking technique is effective against a censor deploying allowlisting for the DTLS stack of browsers.

We wanted to calculate the theoretical number of fingerprints our randomization offered. Equation 1 is used for finding the amount of possible permutations when randomizing the length of a list with n elements.

$$P(n) = \sum_{r=1}^{n} \frac{n!}{(n-r)!} \tag{1}$$

Pion supports 8 ciphers which we pick a random amount of and shuffle. The number of possible values for our randomized ciphers list are therefore:

$$P_{ciphers} = P(8) = 109,600 \tag{2}$$

Our randomization mode only shuffles the order of extensions, which we observed usually contained 7 extensions. We shuffle and pick random values for the *supported_groups*, *use_srtp* and *signature_algorithms* extensions with 3,4 and 7 possible values. Additionally, we randomly pick one of 15 common values for the *APLN* extension. The number of possible unique fingerprints for randomized extensions are:

$$\begin{aligned} P_{extensions} &= 7! \times P(3) \times P(4) \times P(7) \times 15 \\ &= 994,218,624,000 \end{aligned} \tag{3}$$

We believe both $P_{ciphers}$ and $P_{extensions}$ are sufficiently large to resist blocklist fingerprinting.

4 Results

We analyse the key findings of our approach in this section. It includes a review of the DTLS handshakes that were collected over the period of one year for Chrome and Firefox releases. We also provide an evaluation of our DTLS implementation in deployed Snowflake proxies, using both mimicking and randomization, and compare them against standard proxies.

4.1 DTLS fingerprint evolution of browsers

In this section we present the results of capturing DTLS handshakes from a WebRTC application in Chrome and Firefox over a year (April 2024 to April 2025). As discussed in Section 3, these handshakes were generated daily based on the latest available versions of the browsers.

The dataset contained 44 handshakes from Chrome version 124.0.6367.60 to 135.0.7049.84 and 30 handshakes from Firefox version 125.0.1 to 137.0.1. Unfortunately, we lost the captures of about two months in October and November of 2024, thus not having

captures for Chrome version 130 and Firefox version 132. With this dataset, we were able to use *dfind* on the browser captures in a chronological order to determine if there unique fingerprints introduced between versions.

For the Firefox handshakes we found that until May 2024 there was a constant fingerprint that identified the browser. However, from version 127 released in May of 2024, DTLS 1.3 started being used for WebRTC by default. As we did not notice this before analysis, we only captured DTLS 1.3 handshakes for most of the year, although gathering handshakes from the DTLS 1.2 implementation in the same time period would have been useful. In the analyzed data, the extension bytes are unique each time as they include the *key_share* extension containing unique *key_exchanges* (containing public cryptographic keys). Despite the use of DTLS 1.3, we did not identify any use of ECH in Firefox, which would further increase privacy, security and fingerprint-resistance. In the few DTLS 1.2 handshakes we generated, we discovered that the *record_size_limit* extension was present in Firefox, which is not supported by Pion and is a certain way of differentiating DTLS implementations.

DTLS 1.3 has been implemented in the *boringSSL* library, which is used by Chrome. However, in this browser, it has not yet been enabled by default.

Analyzing the Chrome handshakes we found that from version 129.0.6668.58 the order of the extension list is randomized and thus produces a unique fingerprint each time. This makes DTLS fingerprinting harder to do in Chrome by default. Before the randomization of the extension list in mid-September 2024, we saw no change in fingerprint for Chrome for a period of 5 months.

4.2 Snowflake measurements

This section presents measurements of Snowflake traffic. We begin by introducing the metrics for the different types of proxies, as well as the number of daily Snowflake users. Then we look at the traffic captured while running distinct proxy configurations, each over the span of 24 hours.

We used the statistics of Snowflake proxy types collected by the brokers that is being used in the Tor Project[9]. These metrics are according to the broker specification a count of the total number of unique IP addresses of Snowflake proxies types that have been polled[10]. We assume that the count does not necessarily indicate that they were used by a Snowflake client. This means that these proxies were available for usage, but may not have been used to send traffic to a Tor relay. We use the scripts by David Fifield for generating graphs related to daily users of Snowflake in addition to our own script for plotting unique IPs by type[11].

There are four different types of proxies available in the statistics: badge, standalone, webextension and iptproxy. The first three are types we introduced in the background section above. The iptproxy[12] is a pluggable transport (including Snowflake) proxy for iOS and Android. The application seems to be a *gomobile* wrapper for pluggable transports that are implemented in Go. Since *gomobile*

---

[8]https://github.com/cohosh/snowbox

[9]https://metrics.torproject.org/collector/archive/snowflakes/
[10]https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/blob/main/doc/broker-spec.txt
[11]https://gitlab.torproject.org/dcf/snowflake-graphs/
[12]https://github.com/tladesignz/IPtProxy

is used, we assume the Pion library is used for DTLS, not the native implementation of the mobile platforms.

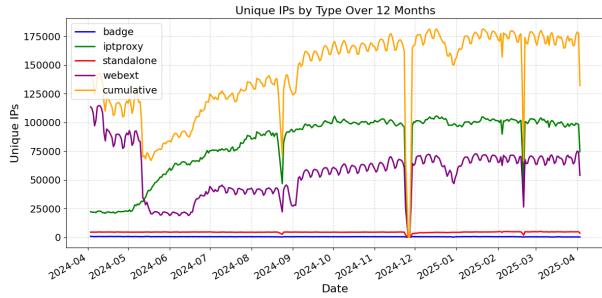**Figure 3: Daily Snowflake users (April 2024 to April 2025)**

Figure 4: Unique proxies by type (April 2024 to April 2025)

Figure 4 show the total number of unique IP addresses per proxy type. Assuming an average of around 175,000 proxies (cumulative) over the past few months, with 100,000 attributed to iptproxy and 60,000 to webextensions, we can say that the distribution is 57% for iptproxy and 35% for webextensions. This reveals that there is only a small percentage (<8%) of active badge and standalone proxies. It is important to note that the high observed number of iptproxy proxies is partly due to the dynamic nature of mobile IP addresses, which can artificially inflate the apparent availability of these proxies.

In Figure 5 we see a drop of Snowflake users starting in November 2024. This has been discussed in the community as blocking in Russia[13] and is suspected to be due to blocking domain fronting to phpmyadmin.net. Additionally, we see a drop and a subsequent steady decline of users from Iran in late December 2024. This might be due to Iran lifting their ban on WhatsApp and Google Play [3], and therefore reducing the interest in Snowflake.

During testing of our DTLS library we observed that the Snowflake proxy tends to become the DTLS client during handshakes. To measure the stability and impact of *covertDTLS* manipulating *ClientHello* messages, we deployed four different configurations of Snowflake proxies: a standalone proxy baseline[14], standalone proxy using *covertDTLS* with mimicking (picking a new fingerpint for each session), standalone proxy using *covertDTLS*

---

[13]https://github.com/net4people/bbs/issues/422
[14]Commit ae5bd528211f07ca4be8582571b5a33f11fcf853

---

with randomization and Chrome[15] with the Snowflake webexten-sion[16]. Each of the proxies were run continuously for 24 hours on the same machine[17] with a static IP (IPv4 and IPv6) and no NAT. We expect restricted NATing to be more realistic for a proxy, but we chose unrestricted NAT to more easily be able to be matched with any client.

For all the deployments, we saw that only between 6 and 8 DTLS *ClientHello* messages were not sent from the proxies. Most of those originated from known Internet scanners. In total, only 5 hand-shakes were completed that were not initiated by a *ClientHello* from the proxy, but instead by the other peer. All of these hand-shakes came from the same data center in the Netherlands. For the webextension, handshakes were not completed when the proxy received a *ClientHello*.

Type	# CH	# SH	# HV	# CCS*	Failed
Baseline	3136	1466	1456	1458	12.5%
Mimicking	2055	961	921	922	18.2%
Randomization	1678	709	702	708	27.0%
Chrome webext.	216	74	113	72	25.8%

Table 1: Number of DTLS handshake messages and failure rate of Snowflake proxies running for 24h. Asterisk indicates message being sent from the proxy.

Table 1 shows the number of handshake messages and the esti-mated failure rate of handshakes for each deployment. We assume that a *ChangeCipherSpec* (CCS) message indicates a successful hand-shake by starting an encrypted channel, and a *HelloVerifyRequest* (HV) message triggers duplicate *ClientHello* (CH). For calculating an estimated failure rate we used the following formula:

$$\text{Failure Rate} = 1 - \frac{\#\text{CCS}}{\#\text{CH} - \#\text{HV} - \#\text{CH from scanners}}$$

The baseline proxy configuration exhibits the highest traffic and is the most stable among the tested setups. However, the amount of traffic may vary over time due do varying demand. The mimick-ing option also demonstrates considerable stability, regarding the amount of successful handshakes. On the other hand, randomiza-tion appears to be error-prone, likely due to the cipher list offering several ciphers that the server rejects. Interestingly, the Chrome webextension generates significantly fewer DTLS sessions. This may be attributed to the standalone proxy's ability to utilize con-currency in Go, allowing multiple sessions to run in parallel. The reason behind the Chrome's proxy instability remains unknown to us.

5 Lessons learned

This section provides the key takeaways from combining the knowl-edge gained during the implementation of our artifacts together with the results discussed in the previous section.

The evolution of browsers' fingerprints had no noticeable effect on Snowflake's number of daily users. Despite having observed two significant changes in the DTLS fingerprints from Chrome and Firefox during the last year, this was not reflected in

---

[15]Version 135.0.7049.95
[16]Version 0.9.3
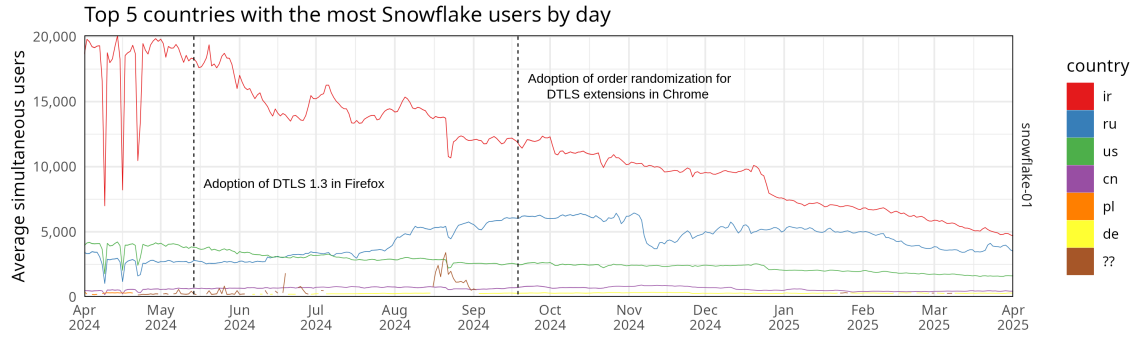[17]Ubuntu 24.04.2 LTS (GNU/Linux 6.8.0-57-generic x86_64) with 4 virtual cores.

Figure 5: Average simultaneous users of Snowflake by country from April 2024 to April 2025

the collected statistics. If new browser fingerprints had a strong effect on Snowflake usage, we would presumably see drops in the number of users when these changes occurred. However, in the per country metrics in included in Figure 5, this is not the case.

A prompt adoption of DTLS 1.3 is needed to keep up with browsers. Firefox's roll-out of DTLS 1.3 for WebRTC introduced some challenges in mimicking current browser behavior. Chrome seems to soon be ready to have DTLS 1.3 by default too. In addition, the ECH feature in DTLS 1.3 would prevent fingerprinting of extensions lists by encryption. However, censors might be willing to block DTLS 1.3 ECH[18]. Except if browsers adopt ECH by default, then the collateral damage from blocking ECH might be too high, making it too costly to be blocked by a censor. Unfortunately, the Pion WebRTC/DTLS stack focuses on DTLS 1.2 with no concrete progress on the DTLS 1.3 implementation yet. We urge the community to prioritize such an implementation, but understand it will take time in a open-source community with no financial support for maintainers to do the work.

We initially did not notice that DTLS 1.3 was adopted in Firefox, and we replayed extensions data byte-for-byte. This naive replaying would produce unique fingerprints for our library as session values in the new extensions for DTLS 1.3 would be the same value in every handshake. After this was discovered, we developed a proof-of-concept of mimicking DTLS 1.3 features in *covertDTLS*. We substituted the public keys in the *key_share* extension with new keys, but had limited success in completing handshakes. This highlights the need for a proper implementation of the DTLS 1.3 extensions rather than modifications such as updating public keys without integrating them into the handshake logic. While waiting for Pion to support DTLS 1.3, we revised our fingerprint generation workflow to cap the maximum DTLS version at 1.2 for WebRTC. This was achieved using the *media.peerconnection.dtls.version.max* setting in Firefox.

Even with a sharp drop in the amount of proxies, it does not seem to affect the number of Snowflake users. There was a large drop of webextension proxies in May 2024 (see Figure 4) due new opt-in consent required by the Mozilla's Add-on store[19]. Even with the large decline of proxies, we see no drop in users in the

same period in Figure 3. Thus we believe there is plentiful amount of proxies to fulfill the needs of users.

Snowflake proxies produce the *ClientHello* messages during the DTLS handshakes. In previous datasets (MacMillan et al.[18]) and earlier fingerprinting attempts, it was implied that the CH was consistently generated using the Pion library by the Snowflake client. However, during testing we observed that it is actually the proxy that assumes the client role in the DTLS handshake, and the reason for us deploying the fingerprint-resistant library at proxies rather than at clients. This behavior is dictated by the SDP protocol. According to RFC 5763, the SDP offer must include the *setup:actpass* attribute, and it is recommended that the SDP answer uses the *setup:active* attribute [31]. The peer designated as active is responsible for initiating the DTLS handshake by sending the CH. Since the Snowflake client is implemented to always send the SDP offer, the proxy—in responding with the SDP answer—becomes the active peer and thus initiates the handshake. If *covertDTLS* is adopted by Snowflake, we should encourage operators of standalone proxies, or iptproxy, to configure their deployments with appropriate randomization or fingerprint-mimicking techniques.

Browser extensions make Snowflake resistant to *ClientHello* fingerprinting. As shown in Figure 4, standalone proxies make up only a small fraction of the total proxy pool with iptproxy and webextensions being abundant. Maintaining a large pool of webextension proxies enhances Snowflake's resilience against fingerprinting as the browser DTLS stack is used for CH messages, although the failure rate is surprisingly high for Chrome.

Standalone proxies can serve more Snowflake clients per volunteer than webextensions. Assuming each CCS message from the proxy indicates a successful DTLS handshake for a new client, we observe that a standalone proxy can serve at least 10 times more clients compared to those based on webextensions. This extra capacity of the standalone proxies to serve more clients is most likely due to the concurrency offered by goroutines. Over the past six months, there have been approximately 4,500 standalone proxies available daily. If each of these proxies can serve 10 times the number of clients, they could collectively handle nearly the same volume as webextensions, which average around 60,000 clients. If we further assume that iptproxy can serve only as many clients as the Chrome extension—an intentionally conservative estimate, since *gomobile* may leverage goroutines similarly to standalone

---

[18]https://gitlab.torproject.org/tpo/anti-censorship/censorship-analysis/-/issues/40057#note_3184292

[19]https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake-webext/-/issues/97

proxies—we estimate that approximately 28% of the traffic could be handled by webextensions, with the remaining 72% handled through the Pion DTLS stack.

We need metrics on which types of proxies are actually being matched and successfully used by clients. Such metrics would enable more effective troubleshooting of potential fingerprinting issues. For example, if we observe a decline in successful connections for both iptproxy and standalone proxies, while webextension proxies remain unaffected, we could reasonably infer that fingerprinting of the Pion WebRTC stack may have occurred.

covertDTLS **is stable when mimicking DTLS 1.2 handshakes, while randomization and Chrome webextensions tends to be less stable**. As shown in Table 1, the failure rate for mimicking is within the same order of magnitude as the baseline, while still supporting a high number of clients. This suggests that mimicking DTLS 1.2 has similar stability as baseline and is not currently subject to blocking. We were even successful in mimicking the *record_size_limit* extension added by Firefox and not used by Pion.

However, mimicking Chrome has some caveats. Chrome's fingerprint is evolving at every capture of a new version due to extension order randomization. While this randomization improves Chrome's resistance to fingerprinting by forcing censors to rely on more advanced DPI, it also makes accurate mimicking more difficult. The collected data shows that Chrome typically includes six extensions, leading to 6! = 720 possible permutations — each representing a unique CH fingerprint. Just mimicking a single capture with one of the possible permutations would be a fingerprint that could be blocked with little collateral damage. To reduce the possibility of being consistently blocked by such a fingerprint, we also support picking a random fingerprint to mimic from all the browser versions. In a future version of *covertDTLS* we are considering to implement parsing and shuffling of extensions to mimic this Chrome behavior.

Our library also incorporates randomization, not only of the order of extensions like Chrome, but also in the number of options selected in certain extensions, the cipher suites advertised, and the ALPN values. We estimate that this results in approximately 994, 218, 624, 000 unique permutations vs. 720 in Chrome, significantly lowering the likelihood of generating the same fingerprint repeatedly and thus being targeted by fingerprint blocklisting.

Randomization also seems to correlate with a higher failure rate for both our library and Chrome webextensions. We expect our library to announce short cipher lists and extensions that could be rejected by a server, explaining the failures. That said, we still do not fully understand why Chrome webextensions exhibit a higher failure rate than the baseline standalone proxy, as Chrome do not remove options like our library. Even the failure rate for the standalone proxy is unexpectedly high, and we suggest further investigation into this and webextension stability. We argue that the failure rate of randomization is still acceptable as it is on par with the stability of Chrome webextensions already widely deployed, while offering high resistance against blocklisting. It seems unlikely that the high failure rates across all proxies tested are due to fingerprint blocking, as the CH message would likely be dropped and we would not receive the corresponding SH like we do. Keeping the handshake alive until the final CCS and then possibly modifying it to fail the handshake is more expensive to do, as it requires handshake state management by a censor.

6 Ethics

The pursuit of anonymity and censorship circumvention through technologies such as Snowflake raises significant ethical considerations. While these tools are designed and intended to promote free access to information and protect users from oppressive tactics, they also have the potential to be put to wrong use. The anonymity provided by such technologies can be exploited to commit crimes, including cyber-attacks and other illicit activities, which we do not endorse. Researchers and developers must ensure that their work is aligned with ethical standards and legal frameworks.

During this project, traffic was collected from deployed Snowflake proxies over multiple days. The collected traffic was filtered to isolate DTLS handshakes, while discarding the data traffic. This paper presents an analysis based on the filtered captures. To protect the privacy of the users who connected to our proxies, the dataset will not be released and subsequently deleted with the release of this paper. This decision underscores our commitment to ethical research practices and the safeguarding the privacy of individuals.

7 Conclusion

This work presented a set of artifacts aimed at reducing the fingerprintability of DTLS traffic in Snowflake. We introduced *covertDTLS*, a fingerprint-resistant Go library extending the Pion DTLS library, enabling both mimicking browser implementations and randomizing DTLS handshake messages. Supporting tools include a fingerprint discovery tool and a pipeline for collecting up-to-date DTLS-WebRTC handshakes from Firefox and Chrome.

Our evaluation showed that mimicking the *ClientHello* messages of browsers yields stable performance compared to baseline Snowflake proxies, while randomization of *ClientHello* messages significantly expands the space of potential fingerprints, though at the cost of increased handshake failures. We observed that proxies, not clients, typically initiate the DTLS handshake in Snowflake, clarifying where fingerprint-resistance mechanisms should be applied. Despite changes in browsers DTLS fingerprints and proxy availability, Snowflake usage remained stable, suggesting resilience to evolving protocol behavior. Those changes included Firefox adopting DTLS 1.3 by default and Chrome randomizing its list of extensions list in *ClientHello* messages.

We also emphasize the importance of preparing for broader adoption of DTLS 1.3. As browsers increasingly migrate to this version by default, Snowflake must adapt accordingly to maintain effectiveness. DTLS 1.3 introduces new extension behaviors that challenge our mimicry approach and features like ECH, which would make fingerprinting of extensions obsolete. Meanwhile we wait for Pion to support DTLS 1.3, the current version of *covertDTLS* is useful library to provide fingerprint-resistance in the environment of DTLS 1.2.

Our obtained results demonstrate that the proposed modifications to Snowflake reduced the distinguishability of DTLS traffic, thereby enhancing its capability to bypass censorship. This approach represents a step forward in the ongoing effort to provide access to a free and open internet in regions with heavy censorship. With our modifications, censors might look towards more sophisticated DPI, traffic shape analysis or other parts of the WebRTC stack to find fingerprints of Snowflake.

Acknowledgments

References

[1] 2022. IRC Tip about Signature used to block Snowflake in Russia, 2022-May-16 (#40030) · Issues · The Tor Project / Anti -censorship / censorship-analysis · GitLab. https://gitlab.torproject.org/tpo/anti-censorship/censorship-analysis/-/issues/40030

[2] 2023. Apply Snowflake Remove HelloVerify Countermeasure. https://gitlab.torproject.org/tpo/applications/tor-browser-build/-/merge_requests/637

[3] 2024. Iran lifts ban on WhatsApp and Google Play, state media says. *Reuters* (Dec. 2024). https://www.reuters.com/technology/iran-lifts-ban-whatsapp-google-play-state-media-says-2024-12-24/

[4] Alice, Bob, Carol, Jan Beznazwy, and Amir Houmansadr. 2020. How China Detects and Blocks Shadowsocks. In *Proceedings of the ACM Internet Measurement Conference (IMC '20)*. Association for Computing Machinery, New York, NY, USA, 111–124. https://doi.org/10.1145/3419394.3423644

[5] Harald T. Alvestrand. 2021. *Overview: Real-Time Protocols for Browser-Based Applications*. Request for Comments RFC 8825. Internet Engineering Task Force. https://doi.org/10.17487/RFC8825 Num Pages: 17.

[6] Ali C. Begen, Paul Kyzivat, Colin Perkins, and Mark J. Handley. 2021. *SDP: Session Description Protocol*. Request for Comments RFC 8866. Internet Engineering Task Force. https://doi.org/10.17487/RFC8866 Num Pages: 57.

[7] Cecylia Bocovich, Arlo Breault, David Fifield, Serene, and Xiaokang Wang. 2024. Snowflake, a censorship circumvention system using temporary WebRTC proxies. (2024). https://www.usenix.org/conference/usenixsecurity24/presentation/bocovich

[8] Junqiang Chen, Guang Cheng, and Hantao Mei. 2023. F-ACCUMUL: A Protocol Fingerprint and Accumulative Payload Length Sample-Based Tor-Snowflake Traffic-Identifying Framework. *Applied Sciences* 13, 1 (Jan. 2023), 622. https://doi.org/10.3390/app13010622 Number: 1 Publisher: Multidisciplinary Digital Publishing Institute.

[9] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. 2015. Examining How the Great Firewall Discovers Hidden Circumvention Servers. In *Proceedings of the 2015 Internet Measurement Conference*. 445–458. https://doi.org/10.1145/2815675.2815690

[10] Nurullah Erinola, Marcel Maehren, Robert Merget, Juraj Somorovsky, and Jörg Schwenk. 2023. Exploring the Unknown DTLS Universe: Analysis of the DTLS Server Ecosystem on the Internet. *32nd USENIX Security Symposium (USENIX Security 23)* (2023). https://www.usenix.org/conference/usenixsecurity23/presentation/erinola

[11] David Fifield. 2020. Turbo Tunnel, a good way to design censorship circumvention protocols. https://www.usenix.org/conference/foci20/presentation/fifield

[12] David Fifield and Mia Gil Epner. 2016. Fingerprintability of WebRTC. https://doi.org/10.48550/arXiv.1605.08805 arXiv:1605.08805 [cs].

[13] David Fifield. 2017. *Threat modeling and circumvention of Internet censorship*. PhD dissertation. University of California, Berkeley, Berkeley, CA.

[14] Sergey Frolov and Eric Wustrow. 2019. The use of TLS in Censorship Circumvention. In *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2019.23511

[15] Jordan Holland, Paul Schmitt, Prateek Mittal, and Nick Feamster. 2022. Towards Reproducible Network Traffic Analysis. http://arxiv.org/abs/2203.12410 arXiv:2203.12410 [cs].

[16] A. Houmansadr, C. Brubaker, and V. Shmatikov. 2013. The Parrot Is Dead: Observing Unobservable Network Communications. In *2013 IEEE Symposium on Security and Privacy*. IEEE, Berkeley, CA, 65–79. https://doi.org/10.1109/SP.2013.14

[17] Ari Keränen, Christer Holmberg, and Jonathan Rosenberg. 2018. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*. Request for Comments RFC 8445. Internet Engineering Task Force. https://doi.org/10.17487/RFC8445 Num Pages: 100.

[18] Kyle MacMillan, Jordan Holland, and Prateek Mittal. 2020. Evaluating Snowflake as an Indistinguishable Censorship Circumvention Tool. http://arxiv.org/abs/2008.03254 arXiv:2008.03254 [cs].

[19] Theodor Signebøen Midtlien. 2024. *Reducing distinguishability of DTLS for usage in Snowflake*. Master's thesis. NTNU. https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3159369

[20] United Nations. 1948. Universal Declaration of Human Rights. https://www.un.org/en/about-us/universal-declaration-of-human-rights Publisher: United Nations.

[21] Marc Petit-Huguenin, Gonzalo Salgueiro, Jonathan Rosenberg, Dan Wing, Rohan Mahy, and Philip Matthews. 2020. *Session Traversal Utilities for NAT (STUN)*. Request for Comments RFC 8489. Internet Engineering Task Force. https://doi.org/10.17487/RFC8489 Num Pages: 67.

[22] Ram Sundara Raman, Adrian Stoll, Jakub Dalek, Reethika Ramesh, Will Scott, and Roya Ensafi. 2020. Measuring the Deployment of Network Censorship Filters at Global Scale. In *Proceedings 2020 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2020.23099

[23] Reethika Ramesh, Ram Sundara Raman, Apurva Virkud, Alexandra Dirksen, Armin Huremagic, David Fifield, Dirk Rodenburg, Rod Hynes, Doug Madory, and Roya Ensafi. 2023. Network Responses to Russia's Invasion of Ukraine in 2022: A Cautionary Tale for Internet Freedom. *32nd USENIX Security Symposium (USENIX Security 23)* (2023). https://www.usenix.org/conference/usenixsecurity23/presentation/ramesh-network-responses

[24] Tirumaleswar Reddy.K, Alan Johnston, Philip Matthews, and Jonathan Rosenberg. 2020. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. Request for Comments RFC 8656. Internet Engineering Task Force. https://doi.org/10.17487/RFC8656 Num Pages: 79.

[25] Eric Rescorla and Nagena Modadugu. 2012. *Datagram Transport Layer Security Version 1.2*. Request for Comments RFC 6347. Internet Engineering Task Force. https://doi.org/10.17487/RFC6347 Num Pages: 32.

[26] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. 2025. *TLS Encrypted Client Hello*. Internet Draft draft-ietf-tls-esni-25. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-tls-esni-25

[27] Eric Rescorla, Hannes Tschofenig, and Nagena Modadugu. 2022. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. Request for Comments RFC 9147. Internet Engineering Task Force. https://doi.org/10.17487/RFC9147 Num Pages: 61.

[28] Guoqiang Shu and David Lee. 2011. A Formal Methodology for Network Protocol Fingerprinting. *Parallel and Distributed Systems, IEEE Transactions on* 22 (Dec. 2011), 1813–1825. https://doi.org/10.1109/TPDS.2011.26

[29] Ram Sundara Raman, Prerana Shenoy, Katharina Kohls, and Roya Ensafi. 2020. Censored Planet: An Internet-wide, Longitudinal Censorship Observatory. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Virtual Event USA, 49–66. https://doi.org/10.1145/3372297.3417883

[30] Michael Carl Tschantz, Sadia Afroz, Name Withheld On Request, and Vern Paxson. 2016. SoK: Towards Grounding Censorship Circumvention in Empiricism. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA, 914–933. https://doi.org/10.1109/SP.2016.59

[31] Hannes Tschofenig, Eric Rescorla, and Jason Fischl. 2010. *Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)*. Request for Comments RFC 5763. Internet Engineering Task Force. https://doi.org/10.17487/RFC5763 Num Pages: 37.

[32] Vasilis Ververis, Lucas Lasota, Tatiana Ermakova, and Benjamin Fabian. 2023. Website blocking in the European Union: Network interference from the perspective of Open Internet. *Policy & Internet* 16 (Sept. 2023). https://doi.org/10.1002/poi3.367

[33] Ryan Wails, George Arnold Sullivan, Micah Sherr, and Rob Jansen. 2024. On Precisely Detecting Censorship Circumvention in Real-World Networks. *Network and Distributed System Security Symposium (NDSS)* (2024).

[34] Hongxin Wang, Baojiang Cui, Wenchuan Yang, Jia Cui, Li Su, and Lingling Sun. 2022. An Automated Vulnerability Detection Method for the 5G RRC Protocol Based on Fuzzing. In *2022 4th International Conference on Advances in Computer Technology, Information Science and Communications (CTISC)*. 1–7. https://doi.org/10.1109/CTISC54888.2022.9849690

[35] Yuying Wang, Guilong Yang, Dawei Xu, Cheng Dai, Tianxin Chen, and Yunfan Yang. 2024. Snowflake Anonymous Network Traffic Identification. In *Proceedings of the 13th International Conference on Computer Engineering and Networks*, Yonghong Zhang, Lianyong Qi, Qi Liu, Guangqiang Yin, and Xiaodong Liu (Eds.). Vol. 1127. Springer Nature Singapore, Singapore, 402–412. https://doi.org/10.1007/978-981-99-9247-8_40 Series Title: Lecture Notes in Electrical Engineering.

[36] Mingshi Wu, Jackson Sippe, and Danesh Sivakumar. 2023. How the Great Firewall of China Detects and Blocks Fully Encrypted Trafc. (2023).

[37] Yibo Xie, Gaopeng Gou, Gang Xiong, Zhen Li, and Mingxin Cui. 2023. Covertness Analysis of Snowflake Proxy Request. In *2023 26th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE, Rio de Janeiro, Brazil, 1802–1807. https://doi.org/10.1109/CSCWD57460.2023.10152736