Modular Traffic Shaping for Censorship Evasion

Aaron Johnson
U.S. Naval Research Laboratory
aaron.m.johnson213.civ@us.navy.mil

Rob Jansen
U.S. Naval Research Laboratory
robert.g.jansen7.civ@us.navy.mil

Abstract

Internet censorship poses a significant threat to the rights of individuals to access information and communicate without interference. Such censorship typically prohibits accessing only certain online content or platforms. As a result, it has become popular to evade censorship using encrypted connections to proxies located outside the censor's view. However, proxied traffic has been shown to exhibit characteristic packet size and timing patterns, which are observable to the censor despite encryption. Such identifying patterns may be eliminated via traffic shaping, where packet sizes and times are intentionally modified by the sender. We propose that traffic shaping be accomplished in proxy systems using a modular traffic shaper and present an API for such a module. Modularization will allow easier experimentation as well as reuse of shaping policies across different proxy systems. We implement a proof-of-concept traffic-shaping module in the Proteus system and conduct experiments showing that it can express a variety of shaping policies.

CCS Concepts

• **Networks → Network privacy and anonymity**.

Keywords

Internet censorship evasion; traffic analysis

ACM Reference Format:
Aaron Johnson and Rob Jansen. 2026. Modular Traffic Shaping for Censorship Evasion. In *25th Workshop on Privacy in the Electronic Society (WPES '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3847192.3847369

1 Introduction

Internet censorship has become a common method to control what information a population can access or what opinions it can express. In 2025, Freedom House [21] documented 45 countries blocking selected political, social, or religious content. Such blocking limits individual autonomy and freedom of expression, and on those grounds it has been opposed by institutions including the United States [19], the United Nations [20], and the Freedom Online Coalition of 41 national governments [5].

In reaction to increased censorship, techniques to evade blocks have grown in sophistication [18]. Virtual Private Networks (VPNs) are a popular method, where the VPN protocol encrypts the traffic to a proxy located outside of the censor's control. Tor provides an enhanced version of this approach, as the onion routing protocol routes traffic through multiple relays such that no relay can link the user and destination [17]. However, censors can identify and block proxies based on IP address or proxy-protocol fingerprints [28]. To evade address-based blocking, systems have been designed to provide ephemeral proxies [1, 10], which can be easily replaced if blocked, or to secretly distribute proxies via trusted partners [2]. To prevent protocol fingerprinting, *obfuscated* proxy protocols without identifiable fingerprints have been developed [3, 11, 16, 22, 23].

These protections do not hide traffic patterns that indicate proxy usage, though. The length, timing, and direction of messages sent between a client and proxy are observable despite encryption. Tunneling a protocol like TCP or TLS over a proxy connection produces characteristic traffic patterns that can be used by a censor to identify and block proxy connections [8]. Discrepancies between transport- and application-layer round-trip times can similarly reveal the use of a proxy [29]. Classifiers have been demonstrated that can identify Tor connections based only on packet timing and size and despite the use of protocol obfuscation [25]. The minimum message size has also been shown to be identifying for obfuscated protocols [3].

Some obfuscated protocols do provide message padding to address this problem, including obfs4 [11] and VMess [22]. Obfs4, a Pluggable Transport [14] built for Tor, optionally uses padding to randomize message lengths and frustrate traffic analysis [27], but its simple randomization scheme and has proven to be insufficient for preventing an adversary from recognizing its network traffic [25].

Traffic shaping schemes that modify packet lengths and timing to effectively hide proxied traffic have not yet been demonstrated. Effective shaping may require trading off performance, in the form of bandwidth overhead or latency, for protection from recognition and blocking. Experience from the similar problem of preventing website fingerprinting [9] suggests that efficient solutions may be non-trivial and may evolve as attack techniques improve.

As a result, it would be useful to have a software *framework* for shaping the traffic of proxied systems. Such a framework could allow rapid prototyping of new defenses, easier deployment of new defenses, and defense reuse across multiple different proxy systems. This approach has been suggested and preliminarily explored in recent work [12, 26]. We improve on these proposals with the following contributions: (1) we propose the first concrete interface for traffic shaping in censorship evasion systems, (2) we provide the first demonstration of a traffic-shaping module in a censorship evasion system (Proteus [24]), and (3) we perform experiments showing that our system can express several traffic-shaping policies that are used in existing censorship evasion systems. For further comparison to related work, see Appendix A, and for a discussion of future work, see Appendix B.

2 Interface

Our system is designed primarily for certain types of *obfuscated channels*, that is, communication systems designed to be difficult to detect. The channels we target have the following properties:

This paper is authored by an employee(s) of the United States Government and is in the public domain. Non-exclusive copying or redistribution is allowed, provided that the article citation is given and the authors and agency are clearly identified as its source. Request permissions from owner/author(s).
WPES '26, The Hague, Netherlands
2026. ACM ISBN 979-8-4007-3026-9/2026/11
https://doi.org/10.1145/3847192.3847369

Listing 1: Traffic shaping API

```
trait TrafficShapingPolicy {
  fn set_size_hints(&mut self, mhs: usize, mss: usize);
  fn next(&mut self, event: Event) -> Action;
}
enum Event {
  Read(usize),
  Wrote(usize),
  Received(usize, usize, Option<Bytes>),
  Sent(usize, usize, usize),
  Connected(Option<(Bytes, u32)>),
  Disconnected,
  Waited
}
enum Action {
  Send(usize, Option<Bytes>),
  Wait(Instant)
}
```

(1) like TCP, they provide an interface that is a connection-based bytestream with reliable, in-order delivery; (2) they provide confidentiality and integrity to payload bytes; and (3) their inherent features, such as their message format and their connection handshake, do not themselves need to be shaped.

Obfuscated channels meeting the requirements include uTLS [6], UPGen [23], and Snowflake [1]. Fully Encrypted Protocols like shadowsocks [16] and obfs4 [11] may not fully satisfy the last requirement, as censors may be able to detect them based on handshakes patterns or metadata size [3, 25], but our framework is still useful with such protocols in that it hides application-level traffic patterns indicating tunneling. They just might also require separate shaping below the protocol.

In the setting we target, a client relays traffic through a proxy server via an obfuscated channel. The channel is instructed by the client to open a connection to the proxy server, and it simply delivers bytes between them.

The interface. We present an Application Programming Interface (API) for traffic shaping in Listing 1. We express it here in Rust, but the same pattern can be used in any language or even cross-languages given a similar Application Binary Interface. The API is meant to be implemented by a traffic-shaping *policy* and used to instruct a traffic *shaper* that actually sends and receives data through the obfuscated channel. The policy is a software module that maintains state (e.g., an object). Shaping policies on either end of a connection can communicate to each other via *control* bytes.

The interface (given as a Rust `trait`) consists of two functions: `set_size_hints()` and `next()`. `set_size_hints()` is intended to be called before traffic shaping begins to inform the shaping policy. Its `mhs` argument is the maximum header size, which bounds the number of metadata bytes that may be added by the traffic shaper. Its `mss` argument is the maximum segment size, which is the maximum number of bytes that can be sent by the shaper after a header and before another header is needed. The `mhs` and `mss` values enable the policy to determine how many bytes it would need to allow to be sent in order to ensure the transmission of any control bytes it needs to send. Also, while the policy can assume that any number of bytes can be sent by the shaper, if the policy instructs the shaper to send fewer than `mhs` bytes, then the policy should expect the shaper to send only padding bytes. The `next()` function is called with the most recent event in the system, which the policy uses to update its state, and it returns either (1) when to query the policy again or (2) how many total bytes and any control bytes to send through the obfuscated channel. The total number of bytes to send may be any value of type `usize`, and is not limited in any way by the `mhs` or `mss` values.

The events given to the policy include both network events and proxy events. Note that the parameters attached to some values allow them to carry data. The `Read` and `Wrote` events indicate that the given number of bytes were read from or written to the proxy. The `Received` and `Sent` events both indicate in their first two parameters the total number of bytes and the number of padding bytes that were received from or sent to the obfuscated channel. The third parameter of `Received` includes any control bytes received from the other traffic shaper, and the third `Sent` parameter indicates how many control bytes were sent. The `Connected` and `Disconnected` events indicate that the connection maintained by the obfuscated channel was opened or closed. `Connected` optionally includes shared random bytes generated during connection establishment, such as an ephemeral key, as well as a lower bound on the entropy of those bytes. Shared randomness can be useful for coordinating a random policy selection on both sides without requiring extra control communication. `Waited` is used when the shaper has waited until the time previously returned by the policy and allows it to get the next action to perform.

The actions from the policy are either to send a message or to wait. `Send` indicates how many total bytes to send through the channel and any control bytes to send. `Wait` indicates that the shaper should wait until the specified time, if no other event happens first, and then ask the policy for the next action. An absolute timestamp rather than a duration to wait enables the policy to return the same output across subsequent calls if the next action has not changed.

Using the interface. To give more semantics to the interface, we describe abstractly how we expect the policy to be incorporated into a proxy system. There are several ways of realizing this abstraction, and we do not prescribe any specific programming methodology or software architecture. An illustration of how the traffic shaper is expected to fit into a proxy system is given in Figure 1. Both the client and server incorporate traffic shaping in this way so that traffic is shaped in both directions. At the top, the application, such as a web browser, communicates with the proxy, such as Tor. The proxy sends data through the traffic shaper, which may be required to buffer data to resolve the difference between the amount of data it needs to send and the amount allowed by the shaping policy. The traffic shaper sends to and receives from an obfuscated channel such as Proteus or obfs4. The specific interfaces among the application, proxy, traffic shaper, and channel are outside the scope of this paper; designs may implement them as different processes (e.g., communicating with the application through SOCKS) or as compile-time libraries (e.g., uTLS [6]). Inside the traffic shaper, we assume that there are asynchronously executing tasks (e.g., threads or processes) separately handling reading from and writing to the proxy and the channel. Each invokes `next()` on a shared traffic-shaping policy, which implements the API in Listing 1.

When Sender creates an obfuscated connection (or reconnects after a disconnection), it sends a `Connected` event and performs the action returned by `next()`. If that action is `Send`, it sends that

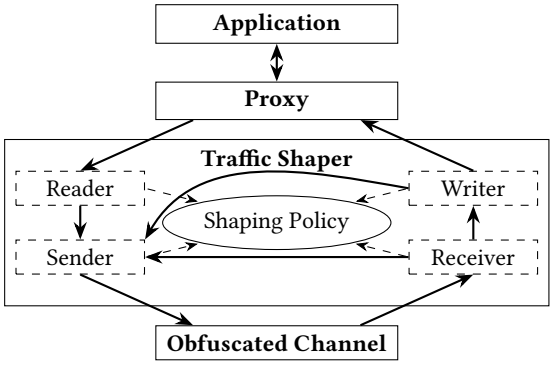


Figure 1: How a proxy uses a traffic shaper and policy, done at both client and server. Dashed squares are asynchronous tasks, the oval is a shared object, solid lines denote the communication, and dashed lines show method invocations.

number of bytes through the channel, preferring first any control bytes returned or already buffered, then taking as many of the remaining bytes from Reader as it can, creating padding bytes if necessary, and adding metadata fields (which count towards total bytes) indicating the number of padding, payload, and control bytes. Then it sends the `Sent` event. If the action is `Wait`, it sets a timer that expires at the indicated timestamp, upon the expiration of which it sends the `Waited` event. This timer is interrupted by the other three shaper tasks if they process some asynchronous event, such as receiving bytes on the channel or reading bytes from the proxy. When its waiting is interrupted, Sender performs the action passed to it by the other task. Sender always performs the action returned by `next()`, except that when a connection closes, it sends a `Disconnected` event and then ignores the returned action.

The other shaper tasks process asynchronous events and update the policy with those events by calling `next()`. Instead of performing the returned action, they alert Sender (e.g. by waking it) and pass the action to it. After Reader reads from the proxy it sends a `Read` event, and after Writer writes to the proxy it sends a `Wrote` event. After Receiver reads from the obfuscated channel, it parses the metadata added by Sender, extracts the proxy and control bytes, discards the padding, and sends a `Received` event.

This design gives the traffic-shaping policy the events relevant to determining a traffic shape. The policy is informed when bytes are sent and received through the obfuscated channel, when data is exchanged with the proxy, and when a connection is opened and closed. The policy therefore knows the apparent application behavior in both directions from the perspective of a censor observing the network. It can, for example, implement request-response patterns like a ping, insert random delays like ScrambleSuit [27], or perform constant-rate sending. The control bytes allow the policies on each end to coordinate, such as for version negotiation. The policy is also informed about how much data has been read from or written to the proxy, which is critical for good performance and enables it to implement simple patterns like padding proxy messages to a fixed size and then immediately sending them. It is important to note, however, that using this information poses a tradeoff between security and performance, as allowing the traffic shape to depend on proxy demands leaks information about them. Because of a similar risk, we do not share the values of the unencrypted proxy bytes, just their number, as providing the values would enable the policy to violate the confidentiality of the underlying obfuscated channel.

We emphasize that communicating through the obfuscated channel is not directly communicating through the network or even an OS-level transport like TCP. Therefore, the times and sizes of packets written to the network depends on the design of the channel, which could be as simple as TLS or as complicated as the multi-server Snowflake system [1]. This behavior is by design, as we assume that the obfuscation prevents protocol-level detection, and so traffic shaping is trying to create apparent *application* behavior that does not contain patterns representative of tunneling traffic through a proxy connection. Also, an obfuscated channel might reveal the size of the write calls made to it (e.g. uTLS via TLS record sizes), making it imperative in such cases to shape traffic before it is sent to the obfuscated channel. Thus a shaping policy indicates (through its `Send` action) how many bytes should be written to the obfuscated channel. Note that these bytes do include proxy control messages (such as SOCKS commands), policy control messages, and shaper headers, and so they are also shaped.

3 Implementation

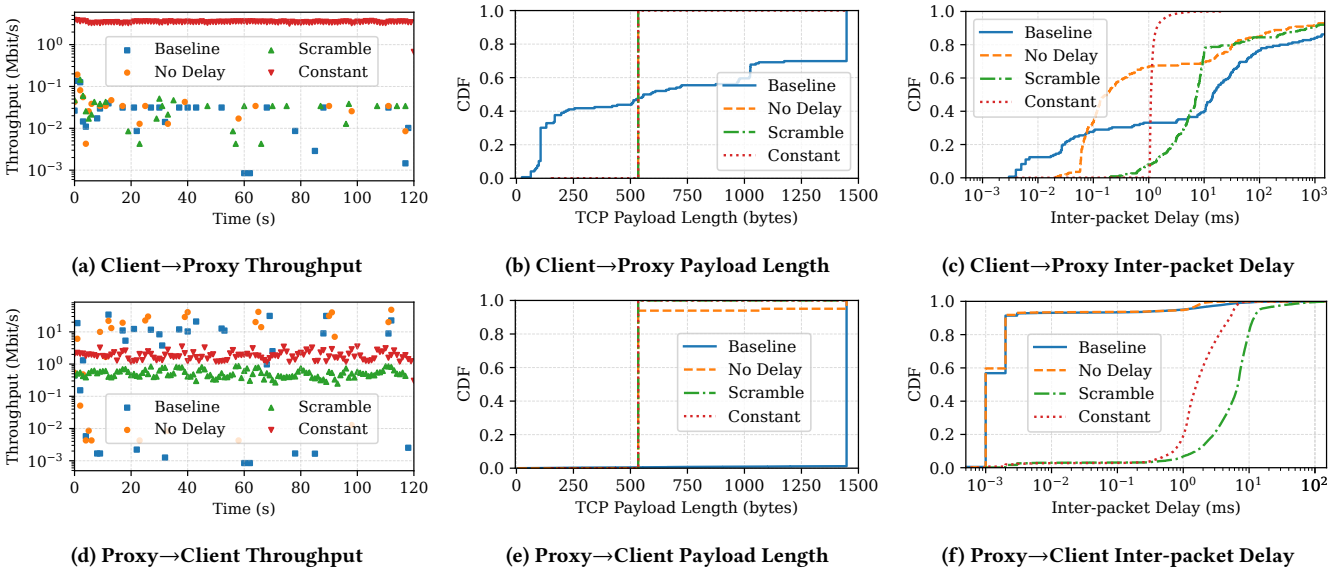
We implemented the traffic-shaping API presented in §2 in Rust as a proof of concept. We envision that it would be incorporated into a software library that could be shared and re-used across multiple tools wishing to incorporate traffic obfuscation. While the interface itself is relatively simple (see Listing 1), we expect that such a traffic-shaping library would also include implementations of several common traffic-shaping policies and could be collectively expanded by the community over time. In our proof of concept, we implemented the following traffic-shaping policies:

No Delay: Outgoing message payloads are padded to a fixed size of 500 bytes (à la Tor), messages are only sent when there exists non-padding data to send, and no artificial delays are added between the sending of consecutive messages. This mirrors obfs4's `iatmode=0` configuration.

Scramble: Like No Delay, except each message send is delayed by a distinct value δ drawn from the uniform distribution $\delta \sim U(0, \delta_{max})$. We use δ_{max}=10 ms to mirror obfs4's `iatmode=1` configuration, which itself was inspired by ScrambleSuit [27].

Constant: Outgoing message payloads are padded to a fixed size of 500 bytes. Messages are sent at a constant rate of one per millisecond, even when non-padding data is unavailable. This mode was previously evaluated in obfs4 [4].

We integrated our proof-of-concept library into Proteus [24] so that we can demonstrate the observable network effects of these traffic-shaping policies. Proteus is a tool for creating VPN-like tunnels where the overt protocol (observable to a network censor) is programmable. Proteus is written in Rust, and its adaptive nature makes it well-suited to traffic-shaping enhancements. We modified Proteus to import and use our library, adding hooks so that its Reader, Writer, Sender, and Receiver tasks (see Figure 1) inform the traffic shaper when events occur while taking the appropriate `Send` and `Wait` actions. The experiments described in the next section use this modified version of Proteus.



(a) Client→Proxy Throughput (b) Client→Proxy Payload Length (c) Client→Proxy Inter-packet Delay

(d) Proxy→Client Throughput (e) Proxy→Client Payload Length (f) Proxy→Client Inter-packet Delay

Figure 2: Network traffic analysis results comparing the unshaped "Baseline" with the three shaping policies from §3.

4 Evaluation

We evaluate our modified version of Proteus with our implemented traffic-shaping policies (see §3) to demonstrate our API's functionality. Our evaluation uses the following nodes:

Client: We run a Proteus client configured to tunnel all Internet streams through a single proxy connection. The client runs on a MacBook Pro and connects over WiFi using a standard home Internet service provider. We use Firefox 140.12.0esr to stream a 4k 60fps HDR video from YouTube through the Proteus client.

Proxy: We run a Proteus proxy on a virtual private server running Ubuntu Linux 22.04.3 LTS. The proxy listens for incoming connections from the client, and then bidirectionally forwards all tunneled client streams between the client and the Internet.

Servers: Standard Internet servers interact with the client through the proxy. The servers participate in neither the proxying protocols nor the traffic-shaping functions.

We set the Proteus Client↔Proxy connection to use a simple encrypted protocol that is distributed with Proteus. Each protocol-level message contains length, length MAC, payload, and payload MAC fields, resulting in 34 bytes of non-payload overhead (2 for the length, and 16 for each of the two MACs). Traffic shaping messages (containing app data, padding, or both) are placed in the protocol's payload field. Finally, the length and payload fields are encrypted using the ChaCha20-Poly1305 cipher with a pre-shared key.

We run a YouTube streaming measurement four times, once without traffic shaping ("Baseline"), and once with each of the policies from §3 (i.e., "No Delay", "Scramble", and "Constant"). During each streaming session, we use tshark to collect packet/payload lengths and directions for 120 seconds. During this experiment, we (1) set the `TCP_NODELAY` option on both ends of the Client↔Proxy TCP connection, and (2) disable TCP segmentation offloading (TSO) on both the client and the proxy; this reduces the chance that the kernels and NICs will coalesce data and distort the traffic patterns.

Our results are shown in Figure 2. The Client→Proxy throughput (Figure 2a) results primarily from client requests for new media segments and is low and sporadic for all but Constant, which exhibits a constant rate (of mostly padding) as expected. In the Proxy→Client direction (Figure 2d) the Baseline and No Delay modes, which do not add artificial delay, oscillate between high-throughput media transfer and then low-throughput control data (while the streaming buffer is sufficiently full). The throughput for Scramble and Constant modes appears more consistent because they add delays, which evens out the flow.

The TCP payload lengths are consistently 534 bytes as expected (34 of Proteus protocol headers and 500 of payload) for all shaping modes in both the Client→Proxy (Figure 2b) and Proxy→Client (Figure 2e) directions. The exception is the Proxy→Client No Delay mode, where media data is being sent fast enough that the absence of artificial delays resulted in multiple messages coalescing in the same TCP packet.

The Client→Proxy inter-packet delay (IPD) distribution (Figure 2b) has a wide range for Baseline and No Delay modes: segment requests are sent relatively infrequently, but when the requests are queued these modes can send them quickly. The IPD distribution for Constant mode drifts just above the configured 1 ms delay due to CPU processing overhead. For Scramble, the IPDs follow the configured uniform distribution between x=0 and x=10 ms, when segment requests are ready; the IPDs can exceed x=10 ms when waiting for more app data (Scramble only sends when data is available). In the higher-volume Proxy→Client direction, when app data is usually available, the IPDs confirm that the Baseline and No Delay modes send quickly while the Constant and Scramble modes follow their configured delays as before.

Acknowledgments

This work was supported by the NRL Base Program.

References

[1] Cecylia Bocovich, Arlo Breault, David Fifield, Xiaokang Wang, et al. 2024. Snowflake, a censorship circumvention system using temporary WebRTC proxies. In *33rd USENIX Security Symposium (USENIX Security)*.

[2] Frederick Douglas, Weiyang Pan, Matthew Caesar, et al. 2016. Salmon: robust proxy distribution for censorship circumvention. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2016, 4.

[3] Ellis Fenske and Aaron Johnson. 2024. Bytes to schlep? Use a FEP: hiding protocol metadata with Fully Encrypted Protocols. In *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)*.

[4] David Fifield. 2017. The limits of timing obfuscation in obfs4. https://archive.torproject.org/websites/lists.torproject.org/pipermail/tor-dev/2017-June/012310.html. Accessed July 12th, 2026. (2017).

[5] Freedom Online Coalition. 2025. Joint statement on protecting human rights online and preventing internet shutdowns in times of armed conflict. Accessed: July 14, 2026. (2025). https://freedomonlinecoalition.com/.

[6] Sergey Frolov and Eric Wustrow. 2019. The use of TLS in censorship circumvention. In *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*.

[7] Jiajun Gong, Wuqi Zhang, Charles Zhang, and Tao Wang. 2022. Surakav: generating realistic traces for a strong website fingerprinting defense. In *2022 IEEE Symposium on Security and Privacy (S&P)*.

[8] Michelina Hanlon, Gerry Wan, Anna Ascheman, and Zakir Durumeric. 2024. Detecting VPN traffic through encapsulated TCP behavior. In *Free and Open Communications on the Internet (FOCI)*.

[9] Nate Mathews, James K Holland, Se Eun Oh, Mohammad Saidur Rahman, Nicholas Hopper, and Matthew Wright. 2023. SoK: a critical evaluation of efficient website fingerprinting defenses. In *2023 IEEE Symposium on Security and Privacy (S&P)*.

[10] Daiyuu Nobori and Yasushi Shinjo. 2014. VPN gate: a volunteer-organized public VPN relay system with blocking resistance for bypassing government censorship firewalls. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[11] obfs4 (The obfourscator). 2023. https://github.com/Yawning/obfs4. Accessed July 14, 2026. (2023).

[12] Hugo Santos Pereira, Afonso Vilalonga, Kevin Gallagher, and Henrique Domingos. 2025. Traffic shaping for network protocols: a modular and developer-friendly framework. *Free and Open Communications on the Internet (FOCI)*.

[13] Mike Perry and George Kadianakis. 2020. Circuit padding developer documentation. https://gitlab.torproject.org/tpo/core/tor/-/blob/HEAD/doc/HACKING/CircuitPaddingDevelopment.md. Accessed July 18, 2026. (2020).

[14] Pluggable Transport Specification (Version 1). 2012. https://spec.torproject.org/pt-spec/. Accessed April 20, 2026. (2012).

[15] Tobias Pulls and Ethan Witwer. 2023. Maybenot: a framework for traffic analysis defenses. In *Proceedings of the 22nd Workshop on Privacy in the Electronic Society (WPES)*.

[16] Shadowsocks. 2022. https://shadowsocks.org/. Accessed July 14, 2026. (2022).

[17] Paul Syverson, Roger Dingledine, and Nick Mathewson. 2004. Tor: the second-generation onion router. In *Proceedings of the 13th Usenix Security Symposium (USENIX Security)*.

[18] Michael Carl Tschantz, Sadia Afroz, Vern Paxson, et al. 2016. SoK: towards grounding censorship circumvention in empiricism. In *2016 IEEE Symposium on Security and Privacy (S&P)*.

[19] U.S. Department of State. 2022. Declaration for the future of the internet. Accessed: July 14, 2026. (2022). https://www.state.gov/declaration-for-the-future-of-the-internet.

[20] United Nations General Assembly. 2023. Promotion and protection of human rights in the context of digital technologies. A/RES/78/213. (2023). https://digitallibrary.un.org/record/4000000.

[21] Kian Vesteinsson and Grant Baker. 2025. Freedom on the net 2025. Accessed: July 14, 2026. (2025). https://freedomhouse.org/report/freedom-net/2025/uncertain-future-global-internet.

[22] VMess. 2024. https://www.v2ray.com/chapter_02/protocols/vmess.html. Accessed July 14, 2022. (2024).

[23] Ryan Wails, Rob Jansen, Aaron Johnson, and Micah Sherr. 2025. Censorship evasion with unidentified protocol generation. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security)*.

[24] Ryan Wails, Rob Jansen, Aaron Johnson, and Micah Sherr. 2023. Proteus: programmable protocols for censorship circumvention. In *Free and Open Communication on the Internet (FOCI)*.

[25] Ryan Wails, George Arnold Sullivan, Micah Sherr, and Rob Jansen. 2024. On precisely detecting censorship circumvention in real-world networks. In *Network and Distributed System Security Symposium (NDSS)*.

[26] Sarah Wilson, Stella Tian, and Sina Kamali. 2025. Shaperd: easily adoptable real-time traffic shaper for fully encrypted protocols. *Free and Open Communications on the Internet (FOCI)*.

[27] Philipp Winter, Tobias Pulls, and Juergen Fuss. 2013. Scramblesuit: a polymorphic network protocol to circumvent censorship. In *Proceedings of the 12th ACM workshop on Workshop on Privacy in the Electronic Society (WPES)*.

[28] Diwen Xue, Reethika Ramesh, Arham Jain, Michaelis Kallitsis, J Alex Halderman, Jedidiah R Crandall, and Roya Ensafi. 2025. OpenVPN is open to VPN fingerprinting. *Communications of the ACM*, 68, 1.

[29] Diwen Xue, Robert Stanley, Piyush Kumar, and Roya Ensafi. 2025. The discriminative power of cross-layer RTTs in fingerprinting proxy traffic. In *Network and Distributed System Security (NDSS)*.

Appendices

A Related Work

Pereira et al. [12] propose a similar traffic-shaping module for censorship evasion. However, their work is preliminary and includes neither implementation nor experimentation. Moreover, their design only supports traffic-independent shaping policies, as the shaper takes no inputs. As a result, the design cannot support interactive patterns between the proxy client and server, such as an HTTP request-response. It also cannot take into account application demands to reduce performance impacts.

Wilson et al. [26] also propose a traffic-shaping system for censorship evasion. Their Shaperd system targets Fully Encrypted Protocols, where the shaping is applied below the encrypted protocol, in contrast to our system which applies shaping above the encrypted protocol. Shaperd uses a high-level constraint-based framework that supports limited shaping policies. In addition, it does not test any actual obfuscated channel or function as a system to correctly transfer bytes between hosts.

Maybenot [15] is another framework for traffic shaping, which evolved out of the Tor Circuit Padding Framework [13]. It has since been applied to WireGuard and QUIC. Although it supports adding extra padding messages, it does not support changing the message size (a limitation inherited from its origin in Tor where data is sent in fixed-size cells). Maybenot also expresses shaping policies using a limited computational formalism (probabilistic finite-state machines), which makes it unsuitable for computationally complex shaping policies such as generative ML models [7].

The original motivation for Maybenot was to defend against website fingerprinting (WF) [9]. Maybenot has been shown to be sufficient to implement many proposed WF defenses. However, our main goal is to prevent the detection of proxy-based censorship evasion, which may require somewhat different techniques. WF defenses need to limit information leakage about the contents of the connection, but censorship evasion requires making the connection look benign. Therefore, aspects like controlling message lengths may not be as relevant to WF defenses as they are to censorship evasion. However, our framework is situated within a larger space of defenses against traffic analysis, and our proposed traffic-shaping modules may have applications to the broader problem.

B Future Work

There remain several opportunities for future work. Our framework operates above the obfuscated channel, but adding shaping below as well may be useful for evasion protocols that themselves have identifying features. The shaper could be extended to shape when connections are opened and closed. Finally, a different interface is needed for obfuscated channels that provide unreliable delivery.