# BlindTLS: Circumventing TLS-based HTTPS censorship

Sambhav Satija
University of Wisconsin-Madison

Rahul Chatterjee
University of Wisconsin-Madison

## ABSTRACT

Governments across the globe limit which sites their citizens can visit by employing multiple kinds of censorship techniques for different types of traffic. ISPs have been able to effectively censor HTTPS traffic by inspecting the TLS handshake which leaks the domain being visited. TLS1.3 attempts to solve this with a proposed ESNI extension which encrypts the SNI (server name indication) value. Since ESNI is optional, ISPs have been known to simply drop handshakes that attempt to use it; SNI based censorship is therefore still a problem even in TLS1.3. We present BlindTLS, a technique that hides the true SNI value in TLS1.2. BlindTLS requires no server modifications and expects only minimal (existing) external infrastructure to circumvent TLS-based censorship. We evaluate and show that BlindTLS is able to successfully provide access to a majority of websites blocked by a real-world ISP with minimal performance overhead.

## CCS CONCEPTS

• **Social and professional topics** → **Censorship**; • **Security and privacy** → **Security protocols**;

## KEYWORDS

Censorship, Circumvention, SNI, TLS

## 1 INTRODUCTION

It is fairly common for governments to censor websites their citizens can visit. They achieve this in different ways. China, for instance, implements a nationwide firewall, colloquially known as the Great Firewall of China or GFW, which detects and prevents connections to blocked websites [3] and uses several techniques to prevent circumvention. In India, the government issues directives to the ISPs, which are free to implement their own means of detecting and blocking connections to websites on a deny-list [25]. In addition to preventing access to certain websites, censorship techniques can also be used to track which websites a user visits, violating their privacy.

ISPs use different filters to prevent a client from connecting to a blocked website. Broadly speaking, traffic is filtered by checking and manipulating (a) unencrypted DNS packets, (b) IP packets, (c) HTTP headers, and (d) leaky TLS handshakes. Manipulation of DNS traffic can be prevented by using DNS over HTTPS (DoH) [11] or DNS over TLS (DoT) [12]. Major browsers now support DoH and use it by default, rendering censorship using DNS packet manipulation largely ineffective (as long as the DoH or DoT servers are accessible). ISPs hesitate to employ IP filters for blocking websites to avoid overblocking [1, 2, 25, 29]. HTTP packet inspection (and manipulation) has been mostly thwarted by the adoption of HTTPS in the last decade [17]. Therefore, ISPs instead identify the domain a client is trying to access by snooping on the TLS handshake's subject name identifier (SNI) field. Although TLS1.3 optionally allows a client to hide the SNI value by sending it in an encrypted-SNI (ESNI) field, this has not been widely adopted. Due to this low adoption, ISPs simply drop all TLS handshakes which include ESNI. Thus, ISPs can still effectively censor HTTPS traffic by parsing the TLS handshake.

To evade circumvention and improve user privacy, a separate class of work broadly treats the ISP as a network adversary and builds protocols to prevent it from peeking into plaintext packets or manipulating them. For example, VPN services [20, 22], Tor [6], and decoy routing [14] enable routing through alternate paths by creating overlay networks over helper nodes. In such systems, the ISP sees the the intermediate relay node (for instance, the VPN server or the Tor relay) as the destination node. Due to their ability to hide the actual destination domain from the ISP, these systems are highly effective and hence widely used to circumvent censorship. However, all these strategies suffer from one key drawback: As the entire traffic from the client is forwarded through the helper nodes, the helper node has to share a significant portion of their network bandwidth with the client.

In contrast, we propose a censorship circumvention technique, called BlindTLS, that introduces minimal overhead by selectively transferring packets over an encrypted TCP proxy. It significantly improves the client's performance by sending most of the network traffic directly to the server, while still bypassing any ISP filters. BlindTLS achieves this by having the client setup the initial TLS session via an encrypted tunnel. This effectively hides the TLS handshake from the ISP's view. Once a TLS session has been established, the client directly reaches out to the server outside the tunnel and resumes the session using the TLS resumption protocol (§2.2). Since ISPs primarily use the leaky TLS1.2 handshake to filter for blocked domains, hiding it from their view is sufficient to evade such filters.

As BlindTLS closely follows the TLS1.2 protocol, deviating only slightly during session resumption, we evaluate and show that a client can access >50% of websites blocked by an Indian ISP without making any modifications to servers (§4.2).

## 2 BACKGROUND & RELATED WORK

### 2.1 TLS handshake

**Leaky TLS handshake:** In the TLS handshake (both TLS1.2 and TLS1.3), the client sends the domain name of the server in plaintext in the Server Name Indication (SNI) field [9]. Moreover, in TLS1.2, the server presents its certificate in plaintext which includes the domain name. Either of these two, a plaintext SNI value or a plaintext certificate, is enough to reveal to a snooping ISP that the connection is intended for a blocked website and can be dropped.

**Necessity for SNI:** While establishing the TLS session, the client expects to receive a valid certificate for the corresponding domain it reached out to. This is a bit tricky in the real world where sites are hosted by front end proxy servers or load balancers, like CloudFlare and Akamai. A given Apache/nginx process, for example, could be listening on a specific IP but be hosting multiple websites, each on different domains and therefore with separate certificates. The server uses the domain name provided in the SNI field to serve the correct certificate to the client.

**Plugging leaks in the TLS1.3 handshake:** TLS1.3 [23] changed some parts of the handshake to prevent leaking the server's domain name in plaintext:

(1) *Encrypted certificates:* By the time the server has to present a certificate to the client, they have already exchanged their corresponding Diffie Hellman shares. Thus, the server is able to encrypt the certificate with a secret key derived from the handshake (§7.3 of [23]) before sending it across. The client can derive the same key as well and is able to decrypt it. The ISP however, is unable to read the certificate to extract the domain name.

(2) *encrypted-SNI (ESNI):* The TLS1.3 handshake introduces an optional ESNI extension [9] which contains the domain name encrypted by the front-end server's public key[1]. The front-end server can decrypt the ESNI value to read the host value, thus effectively blocking it from the view of ISPs.

The key reason why ESNI has been unsuccessful in providing privacy is because it was not an in-built part of the TLS1.3 handshake, but an optional privacy extension. Chai et al. [3] talk about the importance of quick ESNI adoption before counter measures get deployed. Usage of this extension is an explicit signal to the ISP that the client probably wants to reach out to a blocked domain. They argue that since ESNI is optional, the more number of innocent clients use it, the harder it will be to deploy counter-measures against it. Unfortunately, the GFW has already started dropping packets that use the ESNI extension[15], rendering ESNI ineffective.

**Encrypted Client Hello (ECH):** There is ongoing effort [4, 8, 9] to encrypt the entire *ClientHello* message, which would be a drastic change in the handshake specification. Until ECH reaches mass adoption, the ISP can easily drop such packets without worrying about affecting the QoS for a large chunk of innocent citizens, which is precisely what happened for handshakes which included

---

[1]To solve the chicken & egg problem, the client gets the server's public key as a TXT record from DNS. Furthermore, the integrity of this DNS record is ensured using DNSSEC.

ESNI. This change is expected to be more successful if most common browsers quickly adopt this. However, the deployability of ECH is key to realize the promise of immediate mass adoption. It will need to consider the network ossification problem: can middleboxes found in the wild gracefully handle changes to the TLS protocol? Moreover, server administrators will need to ensure that they update DNS records with corresponding servers' public keys. While CDNs will probably adopt this quickly, tools will need to be developed to allow admins of smaller websites to deploy ECH with zero or minimal friction.

To recap, connections using TLS1.2 can be censored because the handshake leaks the domain name in both the plaintext certificate and the SNI value. While TLS1.3 encrypts the certificate, ESNI, being optional, has been ineffective at hiding SNI. Therefore, TLS1.3 is still susceptible to SNI based censorship.

### 2.2 TLS resumption protocol

In order to better explain BlindTLS, we first describe the session resumption protocol of TLS 1.2 [24].

To improve performance and prevent the need for handshakes for each connection, the server sends a unique ticket to the client once the first TLS handshake has been established. When the client wishes to establish another TLS session with the server after some time, it can simply send a *ClientHello* to the server along with this ticket. The server can then, based on the implementation, either decrypt the ticket or look it up from the database, thereby recreating the earlier cryptographic session. This allows the client and the server to immediately restart communication by reusing the secret key established in the previous session. TLS resumption tickets expire after a time period specified by the server, with the observed TTL typically ranging from 5 minutes to 2 days.

TLS 1.3, however, has a slightly different resumption protocol which makes it difficult to directly apply BlindTLS. In §5, we discuss how BlindTLS can work with minor changes to TLS1.3.

We emphasize here that the client is required to send the SNI value while invoking the resumption protocol as well. So the domain name is clearly visible to the ISP during a typical session resumption handshake.

### 2.3 Feasibility of IP filtering

ISPs can easily block traffic to a specific IP destination. However, as pointed out by [1, 25], they hesitate to do so. This intuitively makes sense because in the current landscape of the internet, a hefty portion of the traffic is handled by a small number of CDNs – Cloudflare for instance handles more than 15% of the entire internet traffic [28]. If an ISP employs IP filtering, it will have to drop traffic to all other (potentially harmless) websites that sit behind that particular CDN as well, resulting in overblocking.

Prior work has shown that Indian ISPs prefer to use DNS and HTTP filters [29], and more recently SNI filters [25], but do not employ IP blocking. A recent report [15] implies that GFW uses SNI based filtering to censor HTTPS traffic as well. GFW is known for employing IP level filters as long as the collateral damage is "acceptable". It is unclear how the collateral damage is weighed against the sensitivity of a blocked domain, for instance GFW immediately bans the IP of Tor bridges, while CDNs are handled more carefully.

Aceto et al. [1] demonstrate that Korea and Pakistan do not block IPs, while Bock et al. [2] show that Iran filters for blocked protocols instead of blanket banning IPs.

## 2.4 Evading SNI based censorship

Domain Fronting [10] was an SNI spoofing technique that enjoyed a lot of success in circumventing SNI censorship. Domain Fronting exploits the fact that a lot of websites are hosted behind common CDN servers and that such servers typically do not use the SNI value to determine the backend host. The SNI value is indeed used to serve the appropriate certificate, but the request is routed back to the host specified in the HTTP Host header. This mismatch allows a client to circumvent analysis of the TLS handshake by setting *allowed.com* in the SNI field, but specifying *blocked.com* in the HTTP Host header.

A shortcoming of domain fronting is that a client can only connect to a blocked domain as long as it is behind a CDN that also hosts an allowed domain. Unfortunately, major CDN vendors like Cloudflare, Azure [7] and AWS [18] have banned domain fronting. Moreover, the client never receives the certificate of *blocked.com*, but it receives one for *allowed.com*. Thus, the client has to implicitly trust the CDN that it indeed hosts the *blocked.com* backend.

Domain Hiding[13] bypasses filters which read the plaintext SNI value by sending mismatched ESNI and SNI values. However, it works only with TLS1.3 and can be filtered by checking for the presence of the ESNI extension.

MultiFlow[19] uses the TLS resumption protocol in a manner quite similar to BlindTLS. However, MultiFlow is a decoy routing technique. It assumes the presence of a trusted router in the path between the server and the client. In this lenient trust model, the router knows the private Diffie Hellman share chosen by the client. Moreover, it requires the presence of a separate bulletin board service (decoy host server) to relay the response back to the client.

**Threat Model:**  For our setting, we assume that a client wishes to access a web page on *blocked.com* over HTTPS and the ISP (the attacker) wants to prevent the client from doing so. The ISP can observe and manipulate any traffic involving a client within its service, but is unable to read or affect the traffic outside its boundaries. Thus, there are clients outside its service which can access *blocked.com*. The ISP is free to apply any technique it wishes to censor *blocked.com*, as long as it does not affect the connection to any other allowed domain (*allowed.com*). This includes, but is not limited to, tampering with DNS queries, dropping any packets with certain signatures, and reading the domain name from the SNI field or plaintext certificates.

We assume that the ISP cannot break the security guarantees provided by TLS connections. Based on how ISPs typically censor traffic (§2.3), we assume that the ISP does not employ IP-based blocking. Additionally, we assume that the client can find a TCP proxy node which is not served by the same ISP, and can establish an encrypted channel with it. This proxy in turn has unfettered access to the server hosting *blocked.com*. We do not require the proxy to be honest. While a malicious proxy can leak the identity of the client and degrade performance by not forwarding packets, it cannot break the security of the TLS connection.
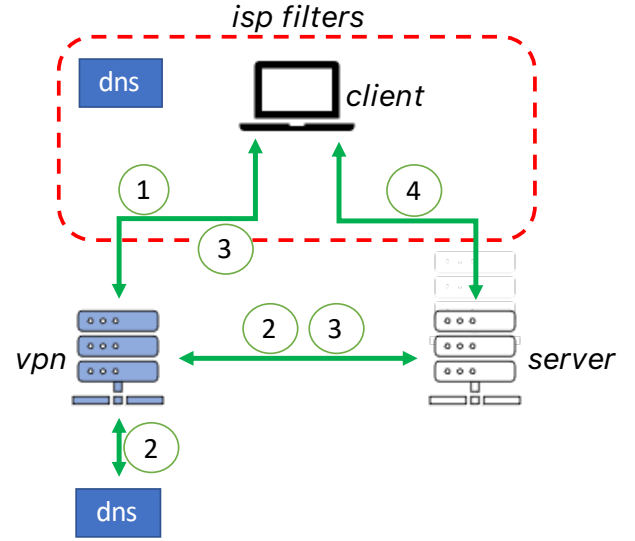


**Figure 1: BlindTLS flow: each circled step corresponds to those in §3. The *vpn* node is the proxy node.**

## 3 DESIGN OF BlindTLS

We now present how BlindTLS uses TLS1.2 to effectively hide the true SNI value from snooping ISPs, while still validating the server by receiving the certificate for the blocked domain.

At a high level, BlindTLS is quite straightforward: the client hides the initial TLS handshake from the ISP by tunneling it via the proxy, and resumes it in plain sight of the ISP. Specifically,

(1) The client finds a TCP proxy node and establishes an encrypted channel with it.
(2) The client first makes a DNS query for *blocked.com* and then initiates the TLS handshake with the supposed server IP. Both these requests are made via the proxy.
(3) On establishing the TLS session, the client waits for the server to send a resumption ticket. This can be achieved by sending normal HTTP requests to the server.
(4) Once the client possesses the *resumption ticket*, it directly reaches out to the same server IP from the previous step and initiates the session resumption protocol (details given in §2.2) using this freshly acquired *resumption ticket*. In this step however, we deviate from the prescribed TLS1.2 specification by sending a random innocuous domain (*allowed.com*) in the SNI header.

In the case of a malicious proxy node, or one with transient connectivity issues, (3) or (4) might fail. The client can then pick a different proxy and restart the protocol. After a successful resumption, a server might provide a fresh resumption ticket to the client. The client can use this new ticket to directly establish a new connection (step 4) the next time. In case a server does not provide a fresh ticket, or if the previous ticket expires, the client reruns BlindTLS. We now further discuss the three techniques BlindTLS uses to function effectively:

**DNS queries:** The client routes the DNS queries via the encrypted proxy tunnel. This allows it to circumvent any DNS censorship mechanisms of its ISP. The client can gain further trust in the DNS response by making a DoH or DoT query via the proxy. Regardless, the client will finally trust the result because it does the entire TLS handshake with the resolved IP address; this step includes receiving and validating the certificate for *blocked.com*.

**Same IP re-connection:** While resuming the TLS session, the client attempts to connect to the same physical server by directing its traffic to the same IP from step 2. This is because if a TLS resumption request somehow reaches a server which is unable to handle the ticket, it will be treated as a fresh TLS handshake request. While some CDNs [5] have mechanisms in place which allow different edge servers to resume each other's sessions, it is not widespread.

**Spoofing SNI during resumption:** As mentioned earlier, the TLS1.2 resumption request includes the SNI value in plaintext. In step 4, the client sends an arbitrary innocuous value (*allowed.com*) in the SNI field, instead of *blocked.com*. Moreover, the server does not present any certificates during the resumption protocol. Thus, the ISP is effectively blinded from ever knowing the true value of the domain name. The key insight of BlindTLS is that a server need not look at the SNI value if it finds the resumption ticket in its cache.

BlindTLS is different from past work like Domain Fronting. In Domain Fronting, the client spoofs the SNI value to one of the co-resident domains hosted by the same server. Therefore, the client receives a valid certificate for *allowed.com* instead of the intended *blocked.com* domain and has to trust that the server indeed hosts *blocked.com* as well. This constraint arises because the server needs to present some certificate that the client can validate; this is required to establish trust in the TLS session. Thus, Domain Fronting primarily works for websites which are hosted behind CDNs. Moreover, it assumes an oracle which informs the clients about active domains hosted by that CDN. BlindTLS does away with all three constraints: A client can reach out to websites that could either be independently hosted or behind a CDN, it does not require any oracle and finally, it removes the trust requirement on a CDN by receiving and validating the actual blocked domain's certificate.

## 4 EVALUATION

We have built a prototype of BlindTLS using OpenSSL [26]. We use openssl s_client to establish a TLS1.2 connection with a server, save the session (including the resumption ticket), and later resume the connection from the saved session information. As and when needed, we tunnel packets via a TCP proxy node in the US (Spectrum ISP). We establish this TCP proxy by utilizing the remote port forwarding capabilities of SSH. We also tested the setup with a small number of websites on ProtonVPN [22] and found no difference in the responses. In our evaluation, the client attempts to resume the session immediately after receiving the ticket to prevent it from expiring. This is representative of a real world setup.

The goal of the evaluation is to check: (a) Is it feasible to use BlindTLS to connect to an unmodified HTTPS server? and (b) Can BlindTLS effectively circumvent censorship by a real-world ISP? To check that the client sees the correct website after resumption, we save the homepage of the website after the initial TLS handshake

and after we resume the session. In all instances where BlindTLS was successful in resuming the session, we found no difference in the web pages.

### 4.1 Feasibility of using BlindTLS for unmodified servers

**Alexa top100:** We first test whether BlindTLS can be used against servers deployed in the wild. For this, we tried accessing the Alexa top100 websites using BlindTLS and record how many of these were accessed successfully. For this setup, the client's ISP was AT&T (west USA), while the proxy's ISP was Spectrum (east USA).

We found that BlindTLS could connect to 47 websites successfully. Among the 53 that it could not connect to, further investigation revealed that 16 websites did not support TLS session resumption, or rather, openssl s_client did not receive a TLS resumption ticket. Servers of the remaining 37 websites refused to resume the session with a spoofed SNI value and instead attempted a fresh handshake. Different edge servers evidently differ in how they handle SNI field modification. Further investigation could reveal how different web server implementations react to SNI spoofing in the resumption handshake. Disproportionate use of such webservers in the Alexa top100 websites could dramatically affect the results.

**Latency:** BlindTLS introduces latency by establishing an initial TCP+TLS session with the server over the encrypted tunnel. While this is an overhead of 3 RTTs, we expect this initial cost to get amortized since the rest of the communication happens directly between the client and the server.

Due to DNS load balancing, a resolved IP is typically geographically close to the node which initiated the query. Thus, in BlindTLS, if the proxy being used is far away from the client, the client might have to communicate with a server that is not necessarily the closest. However, this should still be more performant than routing the entire traffic via the proxy, a VPN node or Tor.

### 4.2 Efficacy of BlindTLS Against a Real ISP

As we show, nearly half of the websites already support session resumption and allow SNI modifications. However, these are Alexa top100 websites; although a few of them are blocked in certain countries, none are blocked in the US, which is where we did our first experiment. We therefore now turn to validate the efficacy of BlindTLS against an ISP that does block a number of websites: Reliance Jio (a major ISP operating in India). As of Feb. 2021, Jio serves more than 35% of wireless users in India [21], the largest share of users compared to any other ISP in India. Moreover, Jio employs the most aggressive censorship techniques amongst all Indian ISPs [25]. For instance, it is the only Indian ISP which actively uses SNI filtering to detect and block traffic. Therefore in this experiment we setup a client that is served by Jio ISP.

Singh et al. [25] provide a corpus of 5,798 domains that the Indian ISPs have been ordered to block. Out of these domains, we randomly sample 500 domains to test if we can access them using BlindTLS. Of these 500 domains, we could only retrieve A records for 332 of them using Cloudflare's DoH server. This is expected because blocked websites frequently change their domain names and discard their previous ones in an effort to stay ahead of ISP DNS blocking.

We now verify the techniques Jio uses to block these domains. We do not test for DNS filters or attacks as BlindTLS (step 2 of §3) explicitly sidesteps any DNS-based censorship techniques employed by the client's ISP. We first check whether Jio employs IP filtering. We do that by attempting to establish a TCP session with the domain's resolved IP on port 443. We find that all connections succeed, except 4. For these 4, we reattempt the TCP connection to port 443 from 2 different ISPs in USA and India each and the connection fails for each case. Hence we safely assume that there is no longer any active server on these IPs. Therefore, Jio clearly does not use IP-based blocking. This is critical for BlindTLS because the client will directly reach out to the server IP post resumption.

To test for SNI filtering, we set up a server which accepts TLS connections with any SNI value and have the client establish TLS1.2 sessions using the remaining 332 domain names. Surprisingly, we found that *all* sessions succeeded; Jio never really checked the SNI value during the handshake[2]. We then modified our server such that it replies with a certificate which specifies a blocked domain as the bound domain name. TLS sessions fail in this case; thus Jio censors traffic by looking at the domain name in the certificate presented by the server during the TLS1.2 handshake.

Interestingly, this is different during a TLS1.2 resumption handshake. Jio drops connections based on the SNI value specified in the resumption protocol. These 2 findings motivate the need for (a) receiving the certificate via an encrypted tunnel — where all the TLS handshake traffic is encrypted and (b) spoofing the SNI during TLS session resumption.

Knowing that Jio does not do IP filtering, but does check the leaky TLS1.2 handshake, we run BlindTLS to see if it is able to effectively circumvent these filters. The client is behind Jio in India and the proxy node is in the USA. Of the 332 domains, openssl s_client was unable to establish and save the session for 120 domains. In these cases, the server did not serve any TLS certificate – this can again be explained by website admins discarding the domain names and forgetting to remove the DNS entries. Thus, we are finally left with 212 live domains that actively host content and Jio is actively preventing connections to them.

Of these 212 active domains, the client is able to use BlindTLS to successfully access 116 domains, achieving a censorship circumvention rate of 54%. The remaining 46% servers did not resume the session (step 4) and instead restart the TLS handshake protocol by presenting their certificate, effectively getting blocked by Jio. We manually validate that the client sees the same website both during the initial handshake and post resumption. This breakup is illustrated in Fig. 2.

## 5 DISCUSSION & FUTURE DIRECTIONS

We now discuss some limitations of BlindTLS and the potential ways an ISP might able to detect and stop its use. We list some possible ways to avoid its detection and argue why attempting to block BlindTLS traffic will lead to over-blocking. We also outline how BlindTLS could be deployed and how it would manage proxy peers.
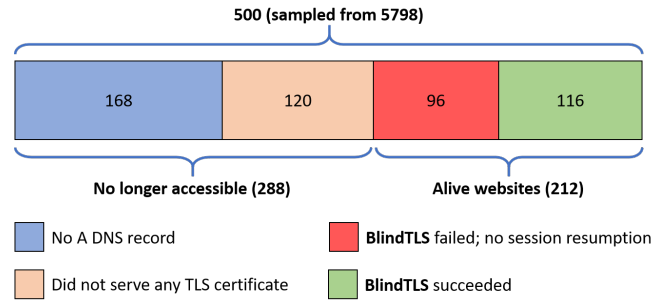


**Figure 2: Out of the sampled 500 domains (all actively censored by Jio), 168+120 are no longer accessible. Of the 212 alive websites, BlindTLS is successful for 116 (54%).**

**Investigating BlindTLS's unsuccessful attempts:** While evaluation against a real world ISP shows that BlindTLS is able to circumvent the filters for a majority of blocked domains, it is not immediately clear why it fails for the rest.

One broad reason for this could be that the server's TLS setup is different from what BlindTLS expects. Due to the various TLS libraries and server implementations being used, there could be inconsistencies in how they react to unexpected input. Specifically, some server implementations might trip when BlindTLS sends a modified SNI value during resumption. It could also be the case that some implementations do not allow TLS resumption when a client's IP has changed. While less probable, it is also possible that some servers have simply not enabled the TLS resumption protocol. A systematic study of common server setups (configuration and TLS libraries) and attempting resumption with changing client IPs would be a reasonable next step in understanding such inconsistencies.

Another possibility is that the client ends up talking with different hosts. As discussed earlier, if a TLS resumption request reaches a physical server which is unable to handle it (decrypt it, or find it in its cache), it is treated as a simple *ClientHello* and a new session handshake is initiated. While BlindTLS attempts to communicate with the same physical host by using the original IP address (step 4 of §3), this might be insufficient in CDNs which employ anycast to distribute load. In anycast, two different hosts share the same physical IP so that the client is automatically routed to the nearest one. Thus, a client might end up communicating with one host when it sets up the initial TLS handshake via the proxy node, but be inadvertently routed to a different host while attempting to resume the session. This phenomena could have been further amplified in our evaluation setup given the geographical difference between our client and proxy node. A further study which places the client and the proxy close by, while still in different countries, could yield more insight. We leave this for future work.

**Censoring BlindTLS:** ISPs could identify the use of BlindTLS in multiple ways. For example, an ISP could *actively probe*[3] the server to check if it indeed hosts the domain passed in the SNI value of the TLS resumption request. This requires the ISP to attempt to establish a TLS connection with the server using the observed SNI value. If the server does not host the domain, it will respond with

---

[2]We validated that all handshakes fail when TLS1.3 is used – i.e. Jio looks at the SNI value in TLS1.3.

[3]As of now, only GFW uses active probing to discover censorship evasion attempts.

either a wrong certificate for a different domain, or an error message stating that there is no valid certificate for that domain. In either case, the ISP will learn that this was probably an evasion attempt. Of course, if the server hosts the domain then this approach will not work, and the ISP will be unable to detect SNI spoofing. Thus if available, BlindTLS should use a domain that is allowed by the ISP and is served by the same IP address. This is similar to the approach used by Domain Fronting.

Servers can make such active probing attempts by ISPs even less effective by simply responding with a self-signed certificate for any queried domain that they do not have a certificate for. Such a certificate would be valid for establishing TLS connections, and ISPs cannot filter on this without blocking the use of all self-signed certificates, a common usecase amongst developers. ISPs could try to create maps of SNIs to server IPs, and look for anomalies in usage. But we believe such a mapping will produce too many false alarms due to IP addresses being constantly reused, making it ineffective.

Another way an ISP can attempt to prevent circumvention by BlindTLS is by blocking communication to proxy nodes. We note that BlindTLS only needs an encrypted tunnel. This simple primitive can be achieved in numerous ways, for instance, by exploiting the remote port forwarding capabilities of SSH, VPN gateways, or even tunneling this traffic within an HTTPS connection (similar to DoH). It would be infeasible for an ISP to disable all such protocols.

Finally, ISPs can disable BlindTLS by simply dropping all TLS session resumption packets. This would result in effectively blocking BlindTLS, while innocent clients would still be able to reconnect to allowed websites by always re-initiating the TLS session. We note that such a blanket ban would however degrade performance for all users and also significantly impact the performance of all servers, blocked or otherwise. Session resumption was added in the TLS specification precisely because a majority of web requests are short lived and the cryptographic cost of repeated TLS handshakes becomes the primary overhead for such connections. TLS session resumptions amortize the cost of the full TLS handshake if the client and the server have had a recent TLS connection.

More broadly, the security of BlindTLS does not depend on a large number of clients quickly adopting it, a property that ESNI required. ESNI was built as an optional header in the TLS1.3 *ClientHello* message. The success of ESNI relied on quick adoption: If many honest clients start using ESNI without falling back to standard SNI, the ISP would be forced to disrupt the communication of honest clients in an effort to stop ESNI. Since less than 0.01% of client traffic [3] used the ESNI field, GFW was able to simply drop all such packets [15] without much outcry. BlindTLS on the other hand, exploits the behaviour of already deployed webservers in how they handle spoofed SNI values during resumption. While newer server versions could check that the SNI value during resumption matches the original value, we do not believe an ISP can block all servers which simply do not upgrade their implementations. Failure to upgrade to a recent version cannot be construed as the server's intent to aid in circumvention.

**Managing proxy nodes:** In order to find the list of available proxy nodes, solutions for finding Tor relays directly apply [27]. Even if a client picks a malicious node, the security of the TLS connection will not be affected, and a client can simply choose another one.

While this leaks the client's IP to the adversary, to the best of our knowledge, no ISP/firewall has the capacity to actively maintain a block-list of client IPs. Furthermore, a direct link between the client and the requested domain name can be circumvented using anonymous messaging systems [16].

**Benefits on selectively tunnelling packets:** BlindTLS differs from past work in that it uses a covert channel only for establishing the initial TLS handshake. The client then switches to using its original ISP for all subsequent traffic. This significantly cuts the compute and bandwidth requirements of existing proxy nodes (for instance, VPN gateways). In such a setting, the cost to spin up peer nodes can be drastically lowered, enabling quicker adoption. We believe that this should also enable volunteers[20] across the globe to willingly participate as proxy nodes. Additionally, since BlindTLS allows the client to directly communicate with the server, its performance is not adversely affected by a sluggish intermediary proxy.

**Adopting BlindTLS for TLS1.3:** TLS1.3 requires that the SNI sent during resumption matches the one sent in the initial handshake. There is no obvious inherent benefit to this design other than for the purpose of making the resumption handshake similar to the original one. Specifically, we leave it to future work to investigate if the original TLS1.2 functionality can be backported to TLS1.3. This would allow BlindTLS to cleanly work for TLS1.3 without further modifications.

**Deploying BlindTLS:** We envision integrating BlindTLS into a browser or a similar application that can (a) filter and route packets to different interfaces when needed, (b) setup the TLS handshake and (c) later modify the SNI field during session resumption. The user could specify a custom proxy peer, along with the mechanism for setting up an encrypted tunnel, or the browser could simply pick one at random from the list it has. Such a browser could behave as usual most of time until connections to a specific site get constantly dropped. Then, the browser could use BlindTLS to transparently gain access to the website.

**Ethical considerations:** This paper did not study any human subjects and raise any ethical issues. While investigating the TLS session resumption behaviour of different servers, we made less than 2 requests per minute, which is negligible compared to the traffic served by such public servers.

## 6 CONCLUSION

We presented BlindTLS, a method to circumvent TLS-based HTTPS censorship by spoofing the SNI value during TLS resumption, and discuss some initial results of its potential to be used in practice without any modification to the web servers. Preliminary evaluations show that BlindTLS can successfully connect to 54% of sites blocked by a major Indian ISP.

# REFERENCES

[1] Giuseppe Aceto, Alessio Botta, Antonio Pescapè, Nick Feamster, M Faheem Awan, Tahir Ahmad, and Saad Qaisar. 2015. Monitoring Internet censorship with UBICA. In *International Workshop on Traffic Monitoring and Analysis*. Springer, 143–157.

[2] Kevin Bock, Yair Fax, Kyle Reese, Jasraj Singh, and Dave Levin. 2020. Detecting and Evading Censorship-in-Depth: A Case Study of Iran's Protocol Whitelister. In *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20)*. USENIX Association. https://www.usenix.org/conference/foci20/presentation/bock

[3] Zimo Chai, Amirhossein Ghafari, and Amir Houmansadr. 2019. On the importance of encrypted-SNI ({ESNI}) to censorship circumvention. In *9th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 19)*.

[4] CloudFlare Christopher Patton. 2020. Good-bye ESNI, hello ECH! https://blog.cloudflare.com/encrypted-client-hello/. (2020).

[5] Cloudflare. 2021. TLS Session Resumption: Full-speed and Secure. https://blog.cloudflare.com/tls-session-resumption-full-speed-and-secure/. (2021).

[6] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, USA, 21.

[7] Eric Doerr. 2021. Securing our approach to domain fronting within Azure . https://www.microsoft.com/security/blog/2021/03/26/securing-our-approach-to-domain-fronting-within-azure/. (2021).

[8] N. Sullivan E. Rescorla, K. Oku. 2020. *TLS Encrypted Client Hello*. RFC. https://tools.ietf.org/html/draft-ietf-tls-esni-08

[9] D. Eastlake. 2011. *Transport Layer Security (TLS) Extensions: Extension Definitions*. RFC 6066.

[10] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. 2015. Blocking-resistant communication through domain fronting. *Proceedings on Privacy Enhancing Technologies* 2015, 2 (2015), 46–64.

[11] P. Hoffman and P. McManus. 2018. *DNS Queries over HTTPS (DoH)*. RFC 8484.

[12] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman. 2016. *Specification for DNS over Transport Layer Security (TLS)*. RFC 7858.

[13] Erik Hunstad. 2020. New tool brings back 'domain fronting' as 'domain hiding'. https://www.zdnet.com/article/def-con-new-tool-brings-back-domain-fronting-as-domain-hiding/. (2020).

[14] Josh Karlin, Daniel Ellard, Alden W Jackson, Christine E Jones, Greg Lauer, David Mankins, and W Timothy Strayer. 2011. Decoy Routing: Toward Unblockable Internet Communication.. In *FOCI*.

[15] Dave Levin Kevin Bock. 2020. Exposing and Circumventing China's Censorship of ESNI. https://gfw.report/blog/gfw_esni_blocking/en/. (2020).

[16] David Lazar, Yossi Gilad, and Nickolai Zeldovich. 2018. Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 711–725. https://www.usenix.org/conference/osdi18/presentation/lazar

[17] LetsEncrypt. 2021. HTTPS stats. https://letsencrypt.org/stats. (2021).

[18] Colm MacCarthaigh. 2018. Enhanced Domain Protections for Amazon CloudFront Requests. https://aws.amazon.com/blogs/security/enhanced-domain-protections-for-amazon-cloudfront-requests/. (2018).

[19] Victoria Manfredi and Pi Songkuntham. 2018. MultiFlow: Cross-Connection Decoy Routing using {TLS} 1.3 Session Resumption. In *8th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 18)*.

[20] 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 14)*. USENIX Association, Seattle, WA, 229–241. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/nobori

[21] Telecom Regulatory Authority of India. 2021. Telecom Subscription Data as on 28th February, 2021. https://www.trai.gov.in/sites/default/files/PR_No.27of2021_0.pdf. (2021).

[22] Proton. 2020. Proton VPN. https://protonvpn.com/. (2020).

[23] E. Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446.

[24] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. 2008. *Transport Layer Security (TLS) Session Resumption without Server-Side State*. RFC 5077.

[25] Kushagra Singh, Gurshabad Grover, and Varun Bansal. 2020. How India Censors the Web. In *12th ACM Conference on Web Science (WebSci '20)*. Association for Computing Machinery, New York, NY, USA, 21–28. https://doi.org/10.1145/3394231.3397891

[26] The OpenSSL Project. 2003. OpenSSL: The Open Source toolkit for SSL/TLS. (April 2003). www.openssl.org.

[27] Tor. 2021. Tor BridgeDB. https://bridges.torproject.org/bridges. (2021).

[28] W3Techs. 2020. Usage statistics and market share of Cloudflare. https://w3techs.com/technologies/details/cn-cloudflare. (2020).

[29] Tarun Kumar Yadav, Akshat Sinha, Devashish Gosain, Piyush Kumar Sharma, and Sambuddho Chakravarty. 2018. Where The Light Gets In: Analyzing Web Censorship Mechanisms in India. In *Proceedings of the Internet Measurement Conference 2018 (IMC '18)*. Association for Computing Machinery, New York, NY, USA, 252–264. https://doi.org/10.1145/3278532.3278555