NetShuffle: Circumventing Censorship with Shuffle Proxies at the Edge

Patrick Tser Jern Kon   Aniket Gattani   Dhiraj Saharia⋆   Tianyu Cao
Diogo Barradas‡   Ang Chen†   Micah Sherr⋆   Benjamin E. Ujcich⋆
Rice University   †University of Michigan   ⋆Georgetown University   ‡University of Waterloo

***Abstract*—NetShuffle is a censorship resistance system that offers "shuffle proxies," where regular proxy services (e.g., HTTPS proxies, Tor bridges) are decoupled from their addresses via continuous in-network change. This makes shuffle proxies significantly more difficult to block compared to their traditional counterparts, because the network locations are now in constant flux. NetShuffle is also designed to engage a new class of support base—edge networks—which have received scant attention from existing work. NetShuffle uses emerging programmable switches to provide the shuffle, while staying otherwise transparent to services and clients, enabling it to be applied as a drop-in network appliance to help promote Internet freedom. We have prototyped NetShuffle in testbed environments and operated it seamlessly on a slice of a live campus network for more than a month, showing that it provides network shuffles in a way that is transparent and incurs negligible overheads.**

1. Introduction

Circumvention technologies are essential for Internet freedom, as more than half the world's population lives in countries ruled by repressive regimes [1]–[3]. Censorship presents significant challenges due to the power imbalance between the censor and the censored. Nation states are capable of massive surveillance, but censored users have very little leverage. Fortunately, censored users are not left to fend for themselves. Users and networks from uncensored countries contribute circumvention services [4]–[10] that relay censored requests originating from censored regions to blocked content located in uncensored regions. Censorship circumvention is therefore, in nature, a collective enterprise. Online freedom requires mobilizing any form of help that uncensored regions are willing to provide. This has further led to an expanding arsenal of circumvention services [4], [11]–[13] and to a perennial call for stronger participation in circumvention services [14]–[16].

Two approaches have gained popularity, operating at end users and core networks, respectively. End users from uncensored regions provide assistance by setting up circumvention proxies, (e.g., Tor [4] or Lantern [11]). Such proxies require few resources to operate, and almost any end user can opt in, encouraging a potentially large support base. However, nation-state censors strive to identify and block user proxies [17]. For instance, a censor can easily obtain Tor relay nodes from nodes' publicly advertised IP addresses [17]. Even for private Tor bridges, whose addresses are distributed out of band (e.g., via email or moat [18]), censors have been adept in identifying and blocking them nevertheless [19]–[22]. From the censor's perspective, blocking individual proxy addresses incurs little collateral damage in terms of economical or social impact, as proxy addresses rarely host services important to the censor.

Core network circumvention techniques are at the opposite end of the spectrum, operating in critical infrastructures. In ISP networks, decoy routing (DR) and its variants [5]–[10] secretly divert traffic from the network core to covert destinations by tapping and sifting through network traffic to identify packets with embedded steganographic signals. In CDN networks, domain fronting [23] and domain shadowing [13] leverage content distribution mechanisms to hide covert domains in network requests and redirect them within the CDN infrastructure. These techniques rely on the reluctance of the censor to block key infrastructures due to high collateral damage, since blocking core networks can lead to severe performance penalty or service unavailability [23]. However, given that only a small number of dominant players operate key infrastructures, there is a high barrier to participation [24], and opt-outs have a critical effect [25]–[27]. Even one opt-out can disable circumvention services for a significant portion of the Internet. To date, Amazon, Azure, and Google have disabled domain fronting [28]–[30], and Akamai selectively suppresses the distribution of content in specific countries [31].

Both classes of techniques enhance the circumvention arsenal and are crucial for countering censorship. However, we believe that more is needed, at an operating point that is currently mostly untapped by censorship circumvention approaches. This paper explores novel circumvention techniques to mobilize a third support base—*edge networks*—which are notably missing from the picture. As a result of their unique resources and features, edge networks offer a new front in the censorship arms race.

Edge networks, such as campus networks, enterprises, and private data centers, provide salient properties as they lie in the middle of the spectrum. In many aspects, edge networks exhibit similar properties as the network core, as they are an integral part of the Internet infrastructure. They offer a variety of services, such as education materials, software downloads, and web content. They are also endowed with considerable resources, including publicly-routed IP

address blocks and hardware equipment such as high-speed networking and compute infrastructure. At the same time, edge networks share certain characteristics with end users as well. They are greater in number than core networks, and may be less constrained by social and economic ties with censors, thus providing a robust support base.

NetShuffle. We present NetShuffle, a censorship circumvention system targeted towards edge networks. NetShuffle offers a new class of "shuffle proxies." NetShuffle works by raising regular proxy services (e.g., HTTPS proxies) to a new degree of unblockability by *decoupling services from their public identifiers via shuffling*. NetShuffle scrambles the mapping between a participating network's domains and its IP addresses, and breaks away from the prevailing method in which proxies are located by fixed identifiers (e.g., fixed IP addresses of Tor bridges). The status quo in the current censorship-resistance landscape is that the tight coupling between proxies and their fixed network locations render them easily blockable once their addresses are identified [19]–[22]. Moreover, the impact of this blocking is both *precise* (i.e., only the targeted proxy gets blocked, with little collateral damage) and *enduring* (i.e., no other service instances can be offered at the same address). Shuffle proxies aim to confuse the censor and drastically increase its blocking cost by moving away from static identifiers, making precise blocking impossible.

For ease of deployment, NetShuffle uses existing edge network resources (e.g., network equipment and IP space) with minimal changes. Operators need only upgrade an edge network's border router to a programmable device (e.g., an Intel Tofino P4 switch [32]) to perform programmable packet processing at hardware speeds, and allow NetShuffle to interface with its authoritative name server to ensure synchronization between domain names and shuffled IPs. Programmable switches are available off-the-shelf, comparable with non-programmable counterparts in cost, and have been deployed in several production data centers [33], [34]. Executing inside the switch, NetShuffle presents a shuffled view to external users regarding the proxy service/identifier mapping, while keeping the internal edge network structure unchanged and staying transparent to internal users and services. Censored clients use NetShuffle by obtaining a proxy identifier, which is an innocuous-looking domain name (e.g., `abc.university.edu`). When a client queries this domain name, NetShuffle's DNS resolver reveals a temporary, client-facing IP address for the proxy that is not the actual internal IP address and may differ across clients even for the same proxy. Inbound and outbound connections to this client-facing address are translated by the border router.

The main technical challenge that NetShuffle addresses is to perform a *transparent shuffle* in which several key security and operational properties must be met simultaneously: (1) external network views must obfuscate the internal network configuration, (2) shuffling must not degrade or disrupt other services' performance, (3) shuffling must impose only a minimal resource footprint (e.g., usage of the IPv4 address space), and (4) any in-network modifications (e.g., TCP or IP header rewriting) must not break existing services. We achieve these properties by a hardware/software codesign at the programmable switch, and build a prototype running in both a testbed and a small-scale live deployment. It took us one person-day to deploy NetShuffle on a campus network slice that has been in use for five years, and NetShuffle has been operating seamlessly with existing services and clients for over a month. We present a comprehensive evaluation to demonstrate its practicality.

Ethics. The IPv4 subnet used for live experiments is operated by the authors, and all additional users of this slice of the university network were made aware of this project before we performed any experimentation. We inspected only our own traffic and did not add additional monitoring of traffic not produced by the authors. Our live experiments were designed to minimize harm—the primary risk was degraded performance, which did not occur.

2. A Case for Circumvention at the Edge

We make a case for edge networks' untapped potential by contrasting them against existing support bases for censorship circumvention. Anti-censorship is a constant arms race and a community endeavor. This points to two ways to tilt the balance in favor of online freedom: (1) enhance the anti-censorship ecosystem with new evasion mechanisms, thus raising the censor's cost; and (2) make it easier for different network entities to volunteer in evasion efforts, thus enlarging the support base. Operating at the edge, NetShuffle contributes to both dimensions. Figure 1 depicts the network landscape.

2.1. Strengths/weaknesses of end user proxies

Individual users have played an instrumental role in anticensorship efforts, with notable services including Tor [4] and Lantern [11] proxies. User proxies are easy to operate and lightweight, and have led to volunteer-driven communities with many supporters.

However, end user proxies are susceptible to enumeration attacks as they are easy to block with little to no collateral damage. To counter this, Tor assigns proxies (i.e., *bridges*) to users using out-of-band mechanisms (e.g., e-mail, CAPTCHA) to reduce enumeration speed. Researchers have also studied distribution methods grounded in game theory [21], social network trust [35], [36], and reputation systems [37]–[39]. Flash proxies [40] and Snowflake [41] allow common Internet users to conveniently deploy ephemeral proxies in their web browsers, which further increases proxy counts at the cost of limited communication performance [42] and possible detection [43]. Despite these efforts, censors often block such proxies [19]–[22], as end users typically do not provide critical services.

Tradeoffs: (+) Very large support base, (-) Very little collateral damage.

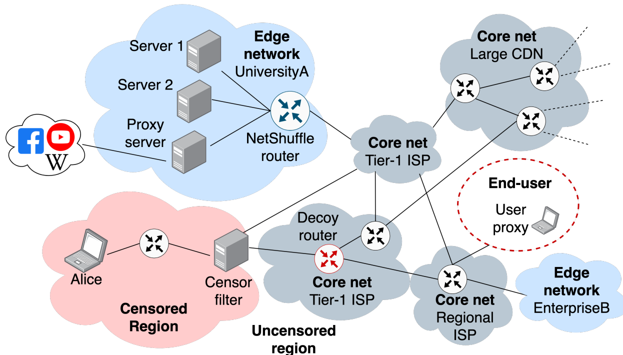


Figure 1: Simplified networking landscape. Edge networks are small ASes, or entities that obtain IP address blocks from an upstream provider. They are mostly customers, rather than providers of Internet access/transit.

2.2. Strengths/weaknesses of core net services

Core network infrastructure (e.g., ISPs, CDNs) can also provide circumvention solutions since such infrastructure controls a vast set of network resources essential to the economic or social well-being of a censoring state [41]. The collateral damage of blocking core networks is substantial, as blocking them effectively takes down a sizeable portion of the Internet for the censored region.

Decoy routing redirects Internet traffic transiting through ISP networks to a covert destination [24], [44], with recent approaches providing reduced overhead with switch-based solutions [7], [45]. For CDN networks, domain fronting [23] makes creative use of the plaintext SNI (Server Name Indication) field that is part of HTTPS requests. A client requests an innocuous domain name in the SNI field and includes an encrypted "Host" header in the payload requesting a censored domain; CDNBrowsing [31], [46] and domain shadowing [13] introduced recent improvements to this idea.

A key limitation, however, is the small number of core networks that can assist with anti-censorship, and consequently the significance that an opt-out has on the underlying circumvention service. Decoy routing remains far from being widely implemented by ISPs, while major CDNs have disabled support for domain fronting [25], [26], [47], [48] in the interest of maintaining business relationships with censoring states. In addition, the proliferation of CDNs around the world (and within censored regions) has resulted in the majority of traffic destined to these core infrastructures to never actually exit the region [49], subjecting it to a region's censorship policies [31] or the provider's own censorship policies [50]–[53].

Tradeoffs: (+) Very large collateral damage, (-) Very small support base.

2.3. Distinct tradeoffs/opportunites at the edge

We argue that *edge networks*, such as campus, enterprise, and other organization networks, are uniquely positioned and hold great potential to act as a new support base.

First, given the large number and distributed nature of edge networks, these networks are strategically located to exploit a large support base whose characteristics resemble those of end users. A recent study [54] conservatively estimates that there are 48k autonomous systems (ASes) at the edge, even without counting edge networks that do not possess their own AS numbers. These edge networks substantially outnumber core networks, thus having the potential to provide a very large support base.

Furthermore, just like core networks, edge networks are an integral part of the Internet infrastructure. They possess a substantial amount of resources, both in terms of the web services and content that they serve, and also their network resources (e.g., domain names and IP spaces) and equipment (e.g., network switches) that end users typically do not possess. In fact, edge networks are collectively the de-facto and decentralized source of collateral damage for blocking adversaries (at the core ISP level), since the negative impact of blocking a core ISP network eventually stems from the inability to access the content served by individual edge networks. Further, maintaining block lists at the address block granularity can be error-prone, and mistakes are compounded when over-blocking IP prefixes at a wrong granularity, creating further difficulty for the censor.

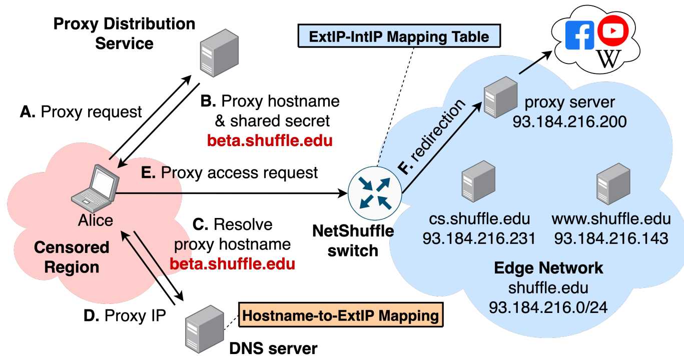
New operating point at the edge: (♦) Substantial collateral damage, (♦) Substantial support base.

However, we lack an effective design that leverages the edge resources for anti-censorship. Today, edge networks can only contribute by setting up regular user proxies (e.g., Tor bridges)—which they often do—but this does not adequately put their substantial edge resources to use. We believe that edge-native evasion techniques will be a strong supplement to the existing anti-censorship apparatuses.

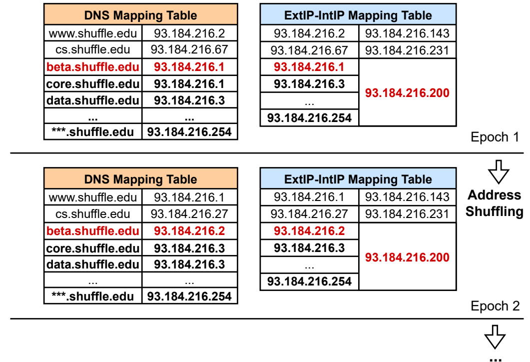
3. NetShuffle Overview

NetShuffle works by raising regular proxies to a new degree of unblockability, leveraging the edge network's support. An edge network participates by (1) setting up one or more existing proxies out of the box, and then (2) deploying NetShuffle to protect these proxies via shuffling. We call the resulting service "shuffle proxies." NetShuffle is agnostic to the specific choice of proxies, so it inherits any proxy-native defenses against active probing and fingerprinting attacks [55] (or the lack thereof). Like many existing circumvention services [24], [56], [57], NetShuffle relies on the altruism of its operators for deployment. Thus, beyond fulfilling its main mission of censorship resistance, we also aim at lowering its deployment barrier as key design goals.

- **Unblockability.** It should be difficult for a censor to identify and block proxies hosted by NetShuffle without incurring high collateral damage.



(a) NetShuffle's workflow from a client's perspective.



(b) Address shuffling between two epochs (**Bold:** proxies).

Figure 2: NetShuffle's workflow across two epochs. In the basic version (§4.1), ExtIP-to-IntIP mapping is a full permutation.

- **Drop-in deployment.** NetShuffle allows for a modular deployment. The biggest change is to replace a border device with a programmable switch[1].
- **Transparent shuffle.** NetShuffle shuffles addresses in a transparent manner without impacting normal edge services and their clients, and does not require configuration changes on edge services or clients.
- **Low resource footprint.** IPv4 addresses are a scarce resource. NetShuffle does not require (but is compatible with) IPv6 since it would create a high barrier to entry due to low (single-digit in many nations) [59]–[61] and stalling [62] adoption at the edge.

Non-goals. There are other protections against proxy enumeration attacks which are orthogonal to our key innovation. Recent work has developed advanced proxy distribution schemes [21], [35], [38], obfuscation techniques to protect against statistical traffic analysis [55], [63], [64], and probe-resistant proxies (that can, for example, send legitimate-looking responses to unauthenticated clients) [63], [65]. We do not advance the state of the art in these aspects, but note that they can be directly applied to shuffle proxies.

Threat model. NetShuffle assumes a standard threat model in censorship circumvention, and we state it for completeness. NetShuffle clients are located within geographical regions under the control of a state-level adversary. The censor is able to monitor and tamper with clients' traffic within its jurisdiction, operate legitimate NetShuffle clients to obtain proxy identifiers, and perform active probing. However, we assume a rational adversary that is sensitive to the collateral damage of blunt coarse-grained tools (e.g., outright blocking of entire address spaces). The adversary is also computationally bounded and cannot break standard cryptographic assumptions. Traffic analysis and active probing attacks are realistic but widely studied concerns [55], [63]–[65], so we assume that some existing countermeasures are in place.

1. NetShuffle currently only supports single-switch execution. We leave the incorporation of multi-switch support using existing work on distributed shared-state for programmable switches [58] as interesting future work.

4. The NetShuffle Defense

We now present a basic version of NetShuffle to illustrate its workflow, and then describe several enhancements.

4.1. The basic network shuffle

Figure 2(a) depicts NetShuffle's workflow. In this scenario, Alice is a client located within a censored region who wishes to communicate with the free Internet. Alice will use NetShuffle to help her reach the unfiltered Internet via Shadowsocks [66]—which is deployed as a shuffle proxy—hosted within an edge network, e.g., a university that owns `shuffle.edu`. (NetShuffle is agnostic to the proxy used.)

Registration and authentication. To access a shuffle proxy, Alice starts (A) by accessing an out-of-band proxy distribution system (e.g., email, moat [18], or those mentioned in §3) The distribution system replies back to Alice with the information for a NetShuffle proxy, which includes a proxy hostname (e.g., `beta.shuffle.edu`) and a shared secret[2] that Alice will later use to establish a secure channel with the proxy (B); the distribution system shares the same information with the proxy. Alice then issues a DNS query for the proxy hostname (C), and waits for a DNS response that contains the proxy IP (D). Alice then establishes a connection with her proxy by contacting this IP (E). The NetShuffle switch converts this IP to the proxy's true IP (see below), and forwards Alice's traffic to the proxy (F). Finally, the proxy authenticates Alice using the previously shared secret. Clients that access a shuffle proxy domain name without a valid shared secret, on the other hand, would receive seemingly "genuine" responses— e.g., existing work (§3) uses timeout or `4xx` HTTP error codes [63], [65].

Per-epoch shuffle. To prevent a censor from blocking Alice's connections towards a proxy, NetShuffle presents an external view of a continuously shuffling set of IP addresses for the hostnames served by the NetShuffle-enabled edge

2. This is analogous to Tor's "bridge lines" which contain key material for accessing a bridge. Bridge lines are communicated via Tor's bridge distribution mechanism. Similar to Tor, we also assume that a censor can access the proxy distribution system to learn about NetShuffle proxies.

network. This external view is updated at a set time interval (i.e., an *epoch*). This shuffle decouples proxy nodes from their static IP addresses by (1) identifying proxies using a large space of subdomain names under a domain belonging to the edge network, but that are not allocated to legitimate edge services, and (2) resolving them to changing IP addresses over time. This procedure is shown in Algorithm 1. (In §7, we discuss why censors cannot block uncommon subdomains without incurring sizable collateral damage.)

Let ALLIPS denote all IP addresses for the network,[3] then a shuffle is a random permutation Π_{IP}: ALLIPS$\rightarrow$ALLIPS. To distinguish between external and internal views of the IP space, we use EXTIPS$\subseteq$ALLIPS to denote the external-facing IP addresses that are visible to clients, and INTIPS$\subseteq$ALLIPS to denote the true IP addresses for edge services, which are never affected by the shuffle. Indexing Π_{IP} with a particular EXTIP will yield its true INTIP. This mapping resides in the border switch, which translates incoming packets' destination IP addresses from EXTIPS to INTIPS. Its reverse mapping Π_{IP}^{-1} is also installed on the switch to translate the source IP addresses of outgoing packets from INTIPS to EXTIPS, so that internal addresses are never revealed to the clients.

Each epoch generates a random shuffle Π_{IP} and its reverse Π_{IP}^{-1}, as well as a third DNS mapping Π_{DNS}, which maps all advertised subdomain names to their current EXTIPS and is installed to the authoritative DNS server to answer resolution requests.

Algorithm 1 The basic network shuffle

```
1: function BASICSHUFFLE(ALLIPS)
2:     Π_ip ← RANDPERM(ALLIPS×ALLIPS)
3:     for each edge service ⟨IntIP, subdomain⟩ do
4:         ExtIP ← Π_ip[IntIP]
5:         Π_dns ← Π_dns ∪ ⟨ ExtIP, subdomain ⟩
6:     Π_ip^-1 ← INV(Π_ip);                              ▷ Inverse
7:     return ⟨ Π_ip, Π_ip^-1, Π_dns ⟩
```

Confusing the censor. We now describe how address shuffling enables circumvention with our running example in Figure 2(b). Consider the DNS mapping for the first epoch, which shows two subdomains for legitimate services (`www.shuffle.edu` and `cs.shuffle.edu`), and many other subdomains that are resolved to EXTIPS, whose corresponding INTIPS map to a proxy. Since Alice's assigned proxy is `beta.shuffle.edu`, she establishes a connection to `93.184.216.1` as the EXTIP and NetShuffle's switch would redirect Alice's connection to the proxy node at `93.184.216.200` as the INTIP after address translation.

Once the first epoch elapses, NetShuffle's mappings will be randomly shuffled, so that the censor cannot easily enumerate and permanently block EXTIPS associated with the hosted proxies. This is because an EXTIP that was used to access a proxy can now point to a legitimate service—see that the EXTIP `92.184.216.1`, which was originally used by hostnames `beta.shuffle.edu` and `core.shuffle.edu`, now maps to the legitimate `www.shuffle.edu` service. Thus, if a censor observes Alice's DNS resolution response for `beta.shuffle.edu` on steps C and D and permanently blocks the resulting EXTIP—or equivalently, Alice acts on behalf of the censor to obtain the subdomain name and EXTIP—it will likely block access to some legitimate service over subsequent epochs, causing collateral damage. In a similar vein, an EXTIP that pointed to a legitimate service can now be used to access a proxy—see that the EXTIP `92.184.216.2` that was originally attributed to `www.shuffle.edu` is now used by `beta.shuffle.edu`. Thus, creating an allowlist for EXTIPS associated to legitimate services is also a futile exercise for the censor. The censor can, however, resort to blocking the subdomain name `beta.shuffle.edu` at the DNS level, but new subdomain names can be constantly supplied to clients as they are abundant[4] and easy to create (we explore this facet of NetShuffle in §7). Another brute-force strategy is to block the entire ALLIPS address space, but this may incur significant collateral damage (see §7).

In most cases, the client's DNS resolver will reside in the censored region. However, regardless of the location of the resolver, requests to `*.shuffle.edu` will ultimately be resolved using the edge network's authoritative name server. (We discuss DNS caching in the next section.) The censor can block responses from this name server, but it incurs a collateral damage that is equivalent to blocking the edge network entirely.

4.2. Performing an asynchronous shuffle

Despite its usefulness in helping to grasp the general idea of our solution, the basic shuffle described above does not consider an important practical aspect of NetShuffle's deployment: dealing with network asynchrony. First, DNS updates experience a propagation delay from the authoritative servers to the clients. This delay is due to the combined effect of the DNS TTL (time-to-live) caching mechanism, which holds on to previously resolved DNS records until TTL timeout [69], and potentially other forms of latency (e.g., DNS caching in resolvers, network delays). Thus, NetShuffle cannot assume that all clients have instantaneous access to the new Π_{DNS} mappings. Second, long-lived connections may span multiple epochs, whereas modifications to Π_{IP} and Π_{IP}^{-1} may change an EXTIP's current mapping and disrupt such connections. Below, we propose two techniques to tackle these issues and accomplish asynchronous address shuffling. This shuffle is detailed in Algorithm 2.

Handling DNS propagation delay. NetShuffle handles the DNS propagation delay using a technique called *IP address space segregation*. We divide ALLIPS into non-overlapping address partitions, $\{\text{IPPART}_1, \text{IPPART}_2, \cdots, \text{IPPART}_R\}$, and cycle through these partitions in a round robin manner

3. Note the operator can specify a list of STATICIPS to be excluded from the shuffle, and hence excluded from ALLIPS. This is useful to enable services directly accessed via IP without first performing DNS resolutions.

4. Proxies use legitimate domains belonging to the hosting edge network. Subdomains could be resupplied by, for instance, sampling from a corpus of hundreds of millions of existing subdomains [67], [68].

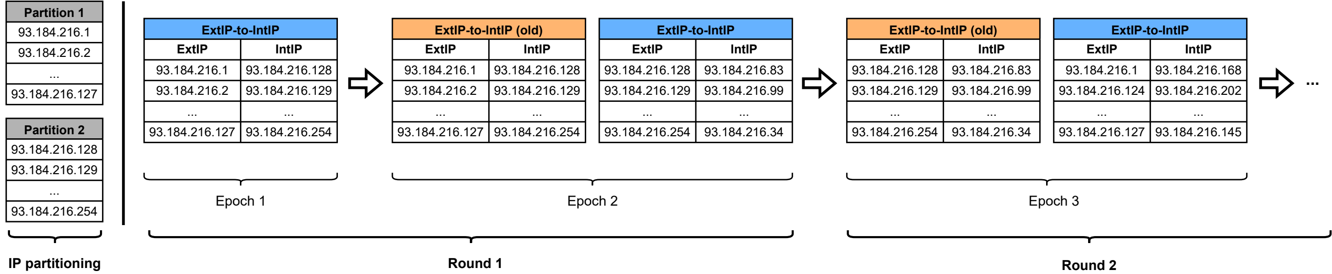


Figure 3: IP address space segregation for R=2. NetShuffle splits all IP addresses into two random partitions. (IP addresses shown in ascending order for clarity of presentation.) It cycles through the partitions across epochs to generate active ExtIPs.

Algorithm 2 The asynchronous shuffle

1: **function** ASYNCSHUFFLE(ALLIPS)
2: $\langle$ IPPart$_1$, $\cdots$, IPPart$_R$ $\rangle \leftarrow$ PARTITION(ALLIPS)
3: **for each** epoch i++ **do**
4: i $\leftarrow$ i mod R
5: $\langle$ Π_{ip}, ... $\rangle \leftarrow$ BASICSHUFFLE(IPPart$_i$)
6: WAITUPONNEWCONN(timeout)
7: **function** NEWCONN(SrcIP, SrcPort, DstIP, Π_{ip})
8: key $\leftarrow$ $\langle$ SrcIP, SrcPort, DstIP $\rangle$
9: val $\leftarrow$ Π_{ip}[DstIP]
10: Conn $\leftarrow$ Conn $\cup$ $\langle$ key, val $\rangle$
11: Conn^{-1} $\leftarrow$ INV(Conn)

across epochs. For epoch i, we use IPPART$_i$ to construct the active mappings Π_{IP_i}, $\Pi^{-1}_{\text{IP}_i}$, and Π_{DNS_i}, but in the switch we remember all mappings used in the most recent R epochs. Assuming for now $|$IPPART$_i| \geq$ $|$INTIPS$|$—i.e., a single partition of addresses is sufficient to support all active edge services—then the mappings are constructed as follows. Instead of the full shuffle Π_{IP}: ALLIPS$\rightarrow$INTIPS, we construct Π^{i}_{IP}: IPPART$_i$$\rightarrow$INTIPS, where we randomly assign an EXTIP chosen from IPPART$_i$ to support the service running at each INTIP. Since partitions do not overlap and we keep state for the R most recent epochs, clients with expired DNS records will only match exactly one (expired) partition, say IPPART$_{i'}$ where $i' < i$, and packet destinations will be translated into the correct INTIPS used by the i'-th epoch. The $\Pi^{-1}_{\text{IP}_i}$ and Π_{DNS_i} mappings are constructed based on Π_{IP_i}. Figure 3 shows an example for $R = 2$ with two randomly partitioned halves.

We note that INTIPS are collected from the network operator as the true composition of the network. Each edge server that hosts some legitimate or proxy services will produce an entry in INTIPS, whereas unused IP addresses that do not provide active Internet services are in ALLIPS$\setminus$INTIPS. In other words, this technique assumes that there is resource slack in terms of allocated but unused IP addresses at the edge. Specifically, the size of a partition, which is $|$ALLIPS$|/R$, must be sufficient for all active services INTIPS. Measurements show that the utilization of publicly routable IPv4 address spaces sits between 50% to 60% [70]–[73]. This gets us close to $R = 2$, but we will significantly reduce this requirement (deferred to §4.3) so that address-constrained edge networks can also deploy NetShuffle.

Supporting long-lived connections. NetShuffle cycles through all partitions $\{\text{IPPART}_1, \text{IPPART}_2, \ldots, \text{IPPART}_R\}$ in R epochs, finishing a round. At this point, it needs to regenerate a fresh mapping using addresses in IPPART$_1$, cycling through the partitions in a second round. The shuffle generated for the (R+1)-th epoch (in the second round) will be different from that for the 1-st epoch (in the first round), because every shuffle is randomly generated and independent. Nevertheless, the same set of EXTIPS in Π_{IP_1} are now provided to external clients, so ambiguity arises. When we receive a packet destined to some EXTIP$\in$IPPART$_1$, we are no longer sure whether this connection was served with mappings from the first or second round; i.e., for long-lived connections that span more than R epochs (one round), the previous method of distinguishing the mapping based on non-overlapping partitions would no longer work.

Our solution keeps two connection tables, CONN and CONN^{-1}, for long-lived connections in the incoming and outgoing directions, respectively. For these connections, we expire their full mappings but two entries remain in these connection tables to keep state. Consider a connection (Figure 4) that started in the 1-st epoch (in the 1-st round) and continued onto the (R+1)-th epoch (in the 2-nd round). Its CONN entry uses $\langle$SRCIP, SRCPORT, DSTIP$\rangle$ as key, where SRCIP and SRCPORT are from the external client, and DSTIP records the EXTIP provided to the client earlier. The value is the service's INTIP in the expired mapping of the 1-st epoch, even though we are currently using a newly generated shuffle in the (R+1)-th epoch. Similarly, a CONN^{-1} entry performs the reverse translation for the same connection. Long-lived connections' expired mappings are thus maintained in CONN and CONN^{-1}. Insertions to the connection tables are performed when the first packet from a connection arrives, and deletions are upon connection teardown (e.g., TCP RST/FIN) or timeout. The latter relies on a switch mechanism, where the control plane software sweeps entries that have not been triggered for a threshold, and garbage-collects them if resources run low. This enables us to clear UDP state, where the communication does not have an explicit teardown process. The switch consults connection tables before the full mappings, to ensure the former takes precedence for a correct translation.

connectionIn			
srcIP	srcPort	dstIP	IntIP
11.11.11.11	94332	93.184.216.110	93.184.216.200

connectionOut			
srcIP	dstPort	dstIP	ExtIP
93.184.216.200	94332	11.11.11.11	93.184.216.110

Figure 4: Connection tables when a client at `11.11.11.11` connects to the `beta.shuffle.edu` proxy hosted at `93.184.216.200`, and whose `ExtIP` at connection time (i.e., potentially multiple epochs in the past) is `93.184.216.110`.

NHP ExtIP-to-IntIP mapping table		
ExtIP	IntPort	IntIP
93.184.216.110	22	93.184.216.200
93.184.216.110	21	93.184.216.143
93.184.216.122	21	93.184.216.200

(a) NHP compaction.

HP ExtIP-to-IntIP mapping table		
ExtIP	IntPort	IntIP
93.184.216.110	443	93.184.216.231
93.184.216.122	443	disambiguation required
93.184.216.122	443	

(b) HP compaction.

Figure 5: Examples of NHP/HP compaction mechanisms.

4.3. Compacting the shuffle

We now address the problem that the asynchronous shuffle requires an R-way partition of ALLIPs and a considerable resource slack—i.e., $|\text{INTIPs}| \leq |\text{ALLIPs}|/R$. The Internet's IPv4 address space has been fully allocated, so the current 50%-60% IPv4 utilization will increase over time [70]–[73]. In fact, the *address efficiency* of network services is important in many settings [74]–[76], as IPv4 addresses are universally scarce. For the same reason, NetShuffle must minimize its footprint to increase deployability.

We draw insights from a recent address efficiency solution [74], but tilt it sideways for switch-resident execution. The idea is that a service provider with limited IPv4 addresses can host many services that are using hostname-based protocols (e.g., HTTPS) at the same server IP and port. Clients visiting different HTTPS domains access the same IP, and TLS' Server Name Indication (SNI) extension is used to determine which service is actually requested. Our compacted shuffle uses a similar strategy for hostname-based protocols (HP), with distinct challenges arising from switch hardware execution. Further, we also compact the shuffle for non-hostname-based protocols (NHPs).

Compacting NHP services. We start with NHP services (e.g., FTP, SSH), where no hostname is included in the protocol itself so disambiguation must happen at the TCP/IP level. Our key idea is to assign the same EXTIP to multiple internal INTIPs as long as their services are listening on different ports. This amplifies the available mapping space by accounting for the ports, while still avoiding ambiguity in the mapping. For instance, consider a network with three servers (and thus three INTIPs), which provide SSH and FTP services (see Figure 5(a)). We will denote the list of distinct NHP service ports as INTPORTS={22, 21}. A single EXTIP can be assigned to multiple INTIPs, as their clients will connect to distinct service ports. This idea is akin to DNAT (destination network address translation), but NetShuffle is different in that (1) this is transparently applied to an already-deployed network without modifying its addressing scheme, and (2) the mapping is constantly changing. Thus, the full shuffle is ALLIPs×INTPORTS→INTIPs with an amplification factor |INTPORTS|, and the partitioning gives $\text{IPPART}_i \times \text{INTPORTS} \rightarrow \text{INTIPs}$ for the i-th epoch. This significantly cuts slack requirements, by a factor of |INTPORTS|.

Compacting HP services. For HP, NetShuffle unlocks even more flexibility by disambiguating clients above the TCP/IP level. As shown in the last two rows of Figure 5(b), we have a single ⟨EXTIP, INTPORT⟩ pair representing two different INTIPs. At the extreme, NetShuffle can map all HP services of the same type (e.g., all HTTPS services at the edge) to a single pair. This is possible because HPs will embed hostnames, which NetShuffle uses for disambiguation (see §4.4). NetShuffle only requires one ⟨EXTIP, INTPORT⟩ pair in a given epoch to undergo disambiguation. All other pairs are allocated the same way as in NHP compaction, to reduce the frequency of disambiguation.

4.4. Compacted HP disambiguation

To achieve greater address efficiency, Fayed et al. [74] propose a compacted HP connection scheme using a reverse proxy. To disambiguate requested domains, a reverse proxy uses the server's TCP stack to finish the three-way handshake with a client (without yet knowing the HTTPS domain being requested) and then waits until the CLIENTHELLO TLS message (carrying the SNI). The proxy then forwards the packets to the correct HTTPS instance on the server. However, in NetShuffle, the switch dataplane does not have a TCP stack, necessitating a different approach with a hardware/software codesign.

Switch-mediated TCP handshakes. Upon receiving a TCP SYN to an EXTIP that hosts compacted HP services, the switch hardware will generate a random sequence number for the reverse direction (as a regular TCP stack would), and constructs a SYN/ACK packet to finish the handshake. At this point, we do not yet know which server is the intended destination, so the switch will also pump the SYN packet into its control plane software via the PCIe bus for buffering until the next CLIENTHELLO message arrives. When the message arrives, the switch hardware extracts the SNI field (see below), and matches this field with a known server name within the edge network to produce the ⟨INTIP,INTPORT⟩ pair. This information is again sent to control plane software, which now releases the buffered SYN packet to the actual server, and completes another TCP handshake "on behalf of" the client. Thus, the switch now holds two TCP connections and serves like a "reverse proxy." Finally, it offloads the proxying process to hardware by installing an entry to CONN and CONN^{-1}. These tables perform address/port translation and edit the TCP sequence numbers to splice the two connections together. Subsequent packets from this connection only go through hardware editing without software overheads.

Accelerating hostname extraction. NetShuffle extracts hostnames from CLIENTHELLO messages in hardware whenever possible, by parsing TLS headers until the SNI field is extracted. The challenge lies in parsing variable-length fields in TLS/TCP headers, which produces large header parsing state [77]. We optimize for the common cases in several ways. First, we leverage measurements from billions of CLIENTHELLO packets [55], [78] to encode in hardware common TLS header variations (e.g., due to cipher suites) covering 75% of the cases. Similarly, we use results from another measurement [79] to support 99% of TCP header variations. Second, we observe that our parser need not recognize arbitrary hostname lengths but only those used in the specific edge network. This enables optimizations for parsing hostnames of interest. NetShuffle addresses unsupported corner cases by relegating them to the control plane software.

5. Implementation

We have implemented NetShuffle using $\sim$8 000 lines of code (LoC). Our data plane implementation uses $\sim$4 000 LoC in $P4_{16}$, allowing us to express custom packet parsing and packet processing operations. Our control plane implementation consists of another $\sim$4 000 LoC written in Python. We have released our code as free open source software [80].

The Tofino switch. NetShuffle is implemented on the Intel Tofino 1 programmable switching ASIC, which is available off-the-shelf. Figure 6 depicts its architecture. Packets arriving at the switch ingress interface follow through a number of *match+action units* (MAUs), organized into stages. MAUs contain entries (matching against specific packet headers or metadata) and their corresponding actions (e.g., modifying packet headers). These entries are populated by the controller after the MAUs are initialized by the P4 program running in the data plane. We use Tofino's incremental checksum update engine to recompute checksums after header modification within the hardware, and send packets to the controller via PCIe for buffering. The switch also exposes an efficient hardware mechanism called "digests" that can compress per-flow data (sending a subset of header fields), and performs automatic deduplication and batching which we use for connection table entry installation.

NetShuffle's layout on Tofino's switch architecture. As shown in Figure 6, Stage 1 includes MAUs for processing traffic that is not shuffled. In Stage 2, MAUs will determine if a packet is incoming/outgoing and whether the packet is NHP or HP-compacted. This influences the CONN and Π_{IP} tables that the packet will be matched against in subsequent stages. A packet that is part of an existing connection will stop matching after Stage 5. New packets are sent to their associated Π_{IP_i} table and dropped if no matches are found; otherwise it is forwarded and a digest is created for CONN entry installation. Depending on the header fields that have been parsed/modified, a different checksum engine will be applied to it (specified by the checksum engine MAU) within the deparser. The TCP state MAU is applied to new HP-compacted packets for custom processing (see §4.4).

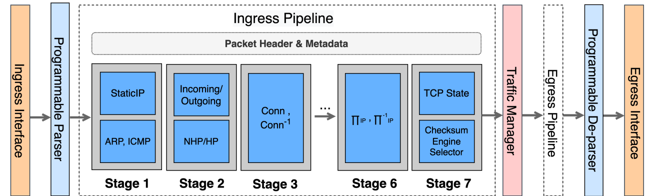


Figure 6: Simplified layout of NetShuffle on a Tofino switch.

6. Evaluation

We evaluate NetShuffle in three dimensions: a) How fast can NetShuffle perform the shuffle for networks of varying sizes (§6.2)? b) How much overhead does NetShuffle incur to the network deployment (§6.3)? and c) How well does NetShuffle work with live network deployments (§6.4)?

6.1. Experimental setup

We use two setups for NetShuffle evaluation. Our *testbed* experiments are conducted in a local cluster which allows us to drill down into the detailed operations of NetShuffle and perform various stress tests. Our *live* experiments are conducted over the Internet, where the NetShuffle switch is deployed as the border router of a dedicated /24 IPv4 subnet in a university network. This setup enables us to observe the behaviors of Internet clients with and without NetShuffle. In both cases, NetShuffle executes in a NetBerg Aurora 710 hardware switch equipped with Intel Tofino P4 programmable ASICs. The switch has 32x100Gbps ports in the data plane, and a 4-core 2.20 GHz Intel Xeon D-1527 CPU as the control plane. The authoritative DNS server for this network uses `bind9` and is hosted in a GCE (Google Cloud Engine) `e2-medium` instance (1 vCPU and 4GB RAM). We use $R = 2$ (two epochs per shuffling round).

Setting appropriate NetShuffle epoch durations. Existing studies have shown that DNS TTLs are by and large respected on the Internet. Using 15k vantage points, Moura et al. [81] show that DNS TTLs are honored by the vast majority of recursive resolvers. Further, prior work shows that common browsers (e.g., Chrome, Safari, Firefox) and OSes (e.g., MacOS, Linux, and Windows) generally upper bound additional DNS TTL caching to one minute [82], [83]. To ensure these results still hold, we validated these findings by running our own experiments (see Appendix A for details). Our results matched the prior studies with the exception that the current version of Microsoft Edge now exhibits TTL caching of up to one minute as well.

Unless otherwise noted, we use an epoch duration of one minute. There are two main motivations for this: First, we do not expect a censor to possess the resources to keep up with mapping updates at a sub-minute granularity—indeed, censors have been found to update their blocklists at much larger timescales [84]. Second, censors already face an additional hurdle to link particular clients to long-term NetShuffle proxy usage. Specifically, censors would need

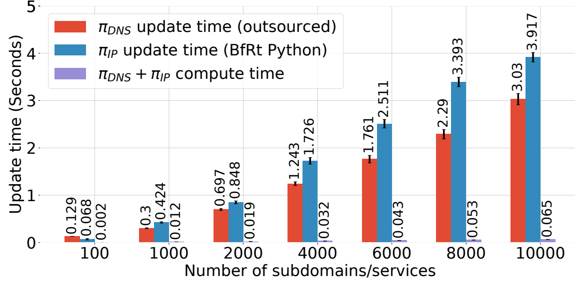


Figure 7: Overhead: Compute and install Π_{DNS} and Π_{IP}.

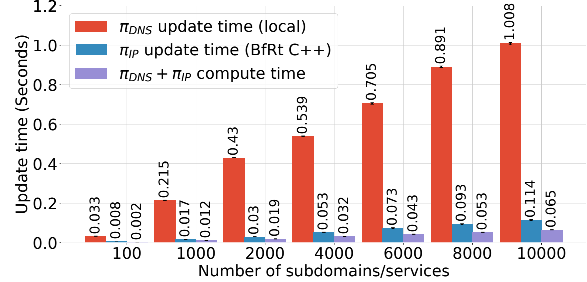


Figure 8: NetShuffle can be optimized for even faster shuffle.

to keep track of any clients' historical connections to the NetShuffle-enabled edge network to determine if a client is accessing a proxy at any given time, since ExtIPs are frequently shuffled. Third, DNS TTLs are typically set to at least one minute [85] (e.g., the minimum TTL is one minute in CloudFlare DNS [86], and ten minutes in Google Domains [87]). Thus, we design NetShuffle to operate comfortably at minute-scale TTL values, and we use one minute as the strictest TTL value.

6.2. Shuffle speeds

We first measure the shuffling speeds for various network sizes in the testbed setup to understand the minimum epoch time—thus the fastest shuffling speed—that is achievable with a practical setup. At the NetShuffle side, the shuffling speeds are determined by the time it takes to perform mapping updates to Π_{IP}, which is located at the NetShuffle switch, and Π_{DNS}, which is hosted at the DNS authoritative name server. At the client side, Internet asynchrony and DNS caching further introduce additional sources of shuffle speed constraints.

NetShuffle mapping updates. We present microbenchmarks that measure the speed of mapping updates at the NetShuffle side, and break it down into two components: i) the controller computes new entries for both Π_{IP} and Π_{DNS}, and ii) the controller then installs these entries into the switch and the DNS server. Further, we test NetShuffle with varying network sizes by changing the numbers of subdomains as well as the online services (that is, the services hosted in the edge network), both from 100 to 10000. We perform 50 trials for each setting and record the average and standard deviation (σ).

Figure 7 shows the results: larger network sizes roughly linearly increase the time it takes for the shuffle. Moreover, computing the shuffles takes negligible time—e.g., only 65ms for the largest tested setting. The shuffle speed is predicated upon the time to update the Π_{IP} and Π_{DNS} mappings, and the former takes more time since the controller needs to operate on the programmable switch ASIC to install a set of control plane entries. Nevertheless, even with 10000 services, it only takes 3.9s on average ($\sigma = 0.12$s) to populate all Π_{IP} to the switch. The DNS updates are performed by initiating the new mappings from the NetShuffle controller to our outsourced DNS server (see §6.1). The DNS update time (< 3s across sizes) includes the network latency.

This is already much faster than our proposed epoch duration of one minute, but we present an additional experiment to demonstrate that the shuffle time can be easily improved, if desired, by implementation optimizations. Figure 8 plots the same data with two optimizations. First, for the Π_{IP} mapping, we used the C++ API provided by the switch instead of the Python bindings which are the default for NetShuffle. This optimized version is able to install entries in batches, reducing the Π_{IP} installation time from 3.9s to 0.1s ($\sigma = 3$ms) with the largest setup. For the Π_{DNS} mapping, we colocated the authoritative server to be a local Dell PowerEdge R350 machine (eight CPUs and 32 GB memory) in the same cluster, eschewing the network latency to the GCE instance. This further reduces the Π_{DNS} installation time from 3s to 1s ($\sigma = 6$ms).

To conclude, NetShuffle's mapping update times are fast and stable (low σ). By initiating update instructions taking into account this overhead, we can ensure updates will complete in a timely manner.

6.3. NetShuffle switch overhead

Next, we measure the overheads of the NetShuffle switch at both the data plane and the control plane.

Data plane overhead. Packets traversing the NetShuffle switch are matched against different Π_{IP} and Conn tables in the P4 program, whereas a basic switch does not incur such additional processing. To quantify this overhead, we measured the per-packet latency through the NetShuffle switch across 1 000 packets, and found the latency to be 350 nanoseconds. When installed with a basic forwarding program, the P4 switch incurs a per-packet latency of 290 nanoseconds. Thus, NetShuffle incurs an additional latency of 60 nanoseconds, a negligible overhead given that typical Internet RTTs are on the order of tens to hundreds of milliseconds [88], [89]. In terms of throughput, the switch hardware uses pipelined processing, and we found the data plane throughput to be stable at 99.8Gbps per port (100Gbps linespeed) when stress-tested with an in-switch hardware packet generator. Thus, the additional latency does not degrade throughput.

Hardware resource utilization. Next, we measure the Tofino switch's resource usage when NetShuffle is deployed

with all the configurations detailed so far. Figure 9 shows varying numbers of concurrent connections that NetShuffle is configured to support and their SRAM usage. This is bottlenecked by the capacity of the CONN and CONN^{-1} switch tables, which hold one entry each for a single connection and only consume SRAM resources as they perform exact matches. We note that P4 programs are compiled onto the hardware in an "all or nothing" fashion, i.e., a P4 program only compiles successfully if it fits within the switch's resource constraints. This means the maximum number of connections a P4 program can support is determined at compilation time rather than runtime.

To assess the maximum number of concurrent connections supported by our switch, we iteratively compiled our P4 with larger table sizes until compilation failure. The results show that NetShuffle can support up to $\sim$220k concurrent connections (using 52.40% of switch SRAM). To put this number into perspective, a typical top-of-rack switch at cloud datacenter scale only needs to support 10k to 100k simultaneous connections at any given time [90]. Thus, NetShuffle is able to support a large number of connections at the edge network.

Table 1 further breaks down the resource utilization for 220k connections, with reasonably low utilization overall. NetShuffle uses VLIWs for action instruction memory, gateways that implement control flow branches, and hash bits for packet header transformations. ALUs are typically used to access registers, but NetShuffle only keeps state in the control plane software. The above results suggest that NetShuffle allows for the simultaneous execution of other typical functions that are needed at the border switch, such as encapsulation, rate limiting, or other security functionalities such as DDoS protection. Finally, these resource constraints are specific to Tofino 1, which is our switch model, and newer versions (e.g., Tofino 2 [91]) have even more abundant resources.

Control plane overhead. Next, we quantify the overhead due to the TCP handshake mediation. As discussed in §4.3, this only occurs for the TCP handshaking phase for a single HP compacted $\langle$EXTIP, INTPORT$\rangle$ pair.

First, we measured the maximum amount of such TCP handshake traffic NetShuffle can buffer for mediation. We saturated the switch with SYN packets sent by the hardware generator to compacted HP services, since such packets represent the bottleneck as they trigger the generation of both a digest and a copy of the packet to be sent to the controller (§4.3). We found that NetShuffle was able to perform packet buffering at a rate of $\sim$100k packets per second (pps). To place this value into perspective, we note that this is significantly larger than the median flow arrival rate reported for popular Facebook services [92], at 500 new flows per second, and is more than sufficient to support our expected user base.

Second, we found the additional end-to-end latency of mediated handshake process to be 48 ms. Thus, the TCP mediation process only incurs minimum overhead at the control plane and for the mediated connections. To improve performance, one may rewrite the control plane software using C++, use more cores for processing, or offload the controller to a stronger standard server. This communication bottleneck can be further alleviated by using, for example, an RDMA connection that could achieve much higher bandwidth between the P4 switch and the server (34Gbps over 40Gbps NIC) [93]. These are interesting optimizations relegated to future work.

Control plane resource utilization. NetShuffle requires the usage of the control plane for performing the disambiguation of compacted EXTIPs for HP services (§4.4). First, NetShuffle uses only 4 out of the 8 available CPU threads (mentioned in §6.1) to perform the computation. Second, as shown in Figure 10, in our most demanding scenario with 220k incomplete connections, the memory used for holding packet digests and buffer space is only 31MB, which is only a fraction of the available memory on the control plane. Further, we note that memory usage scales linearly, because each connection requires the same amount of state (in terms of digests and packet buffer space), which are cleared after connection establishment with connection table rule installment. Thus, NetShuffle only incurs minimum overheads at the switch control plane.

6.4. Live network deployment

Next, we present experimental results of a NetShuffle deployment in a dedicated slice of a university network with a /24 IPv4 subnet. This subnet has been publicly accessible to Internet clients for over five years, and we have upgraded its border router to execute NetShuffle. The network hosts a variety of services, such as web, email, and download servers. We also set up additional services for experimentation. On a regular day, the network exchanges about 2 TB of Internet traffic. The upstream and downstream links from/to this edge network use 1Gbps links.

Deploying NetShuffle—including switch installation, topology setup, and service configuration—only took less than one person/day. It was a drop-in deployment that did not require modifications to clients or services, nor to protocols (e.g., DHCP) or existing firewalls. At the time of writing, NetShuffle has been in deployment for a month and we have never experienced any anomalies due to its use.

We perform our evaluation using our lowest epoch duration of one minute (see §6.1). In the following tables/figures, the baseline refers to the deployment without NetShuffle.

Transparent shuffle. To demonstrate that NetShuffle operates transparently to the existing network deployment, we measure the throughput over time for connections established between an Internet client (deployed on an external network) and various services provided by the edge network.

For this experiment, we have set up six dedicated services that are only used by our Internet clients, to ensure that the measurements are from clients that we can control. SCP bulk file download results in long-lived connections that span multiple epochs, while the rest produce short-lived connections that usually complete within a single epoch. The Apache server was configured to accept HTTP+HTTPS

TABLE 1: Resource utilization on the Tofino switch.

Resource	Map RAM	TCAM	Hash bits	Meter ALU
Utilization	18.75 %	2.43 %	16.95 %	0 %
Resource	**VLIW**	**Gateway**	**Hash units**	**Logical TableID**
Utilization	10.23 %	13.54 %	0 %	32.29 %

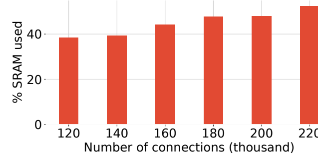


Figure 9: SRAM util. with different number of connections.

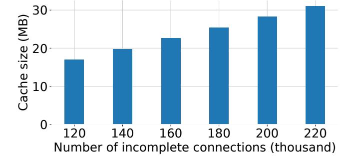


Figure 10: Memory overhead of HP compaction.

TABLE 2: Apache benchmarks (ab) with increasing concurrent connections (C). NetShuffle incurs negligible differences.

Setup	Property	NetShuffle C:10	Baseline C:10	NetShuffle C:100	Baseline C:100	NetShuffle C:1000	Baseline C:1000	NetShuffle C:3000	Baseline C:3000
Apache HTTP (16356 bytes)	Average time/request*	14.859 ms	14.877 ms	1.492 ms	1.493 ms	0.246 ms	0.236 ms	0.322 ms	0.324 ms
	Transfer rate	1.07 MBps	1.07 MBps	10.63 MBps	10.62 MBps	64.42 MBps	67.22 MBps	49.22 MBps	48.97 MBps
	Completion rate	100%	100%	100%	100%	100%	100%	100%	100%
Apache HTTPS (16356 bytes)	Average time/request*	25.647 ms	25.667 ms	2.623 ms	2.612 ms	0.899 ms	0.923 ms	0.942 ms	0.943 ms
	Transfer rate	633.22 KBps	632.73 KBps	6.04 MBps	6.07 MBps	17.65 MBps	17.18 MBps	16.84 MBps	16.82 MBps
	Completion rate	100%	100%	100%	100%	100%	100%	100%	100%

*Mean across all concurrent requests.

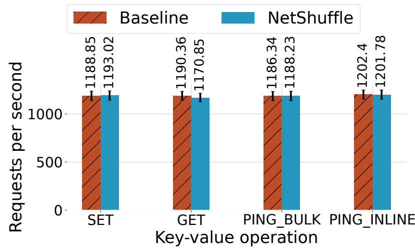


Figure 11: Redis K-V store benchmarks with 50 concurrent connections. NetShuffle incurs negligible overhead.

traffic. Hysteria [94] and Shadowsocks [66] are two popular proxy services used to circumvent censorship; a single webpage access (e.g., querying `www.wikipedia.org`) through either proxy establishes two distinct connections—between the client and proxy, and between the proxy and the webpage (covert destination).

We show the results for five consecutive epochs (2.5 rounds), where every epoch lasts for one minute, in Figure 12. For all services, we observe negligible throughput variations across epochs (and across rounds). We also conducted a more fine-grained evaluation for Apache and Redis, which come with standard benchmarking toolsets (i.e., `ab` for Apache and `redis-benchmark` for Redis). `ab` produces similar results for varying numbers of HTTP and HTTPS connections (document size is 16356 bytes) with and without NetShuffle (Table 2). Figure 11 shows similar takeaways with Redis.

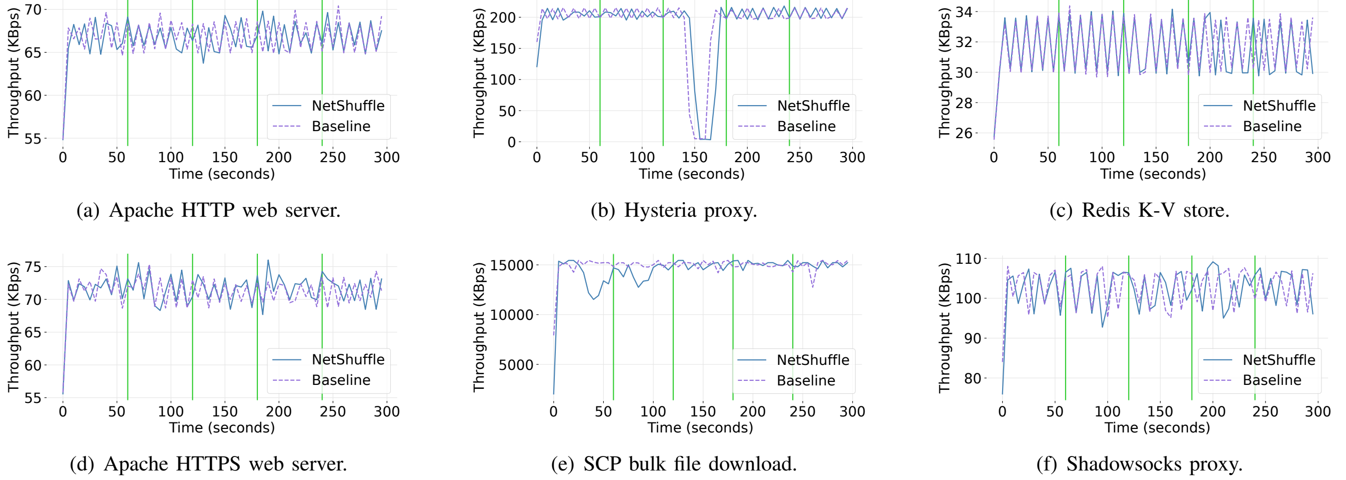
Client diversity. Next, we performed two experiments that demonstrate that NetShuffle also operates transparently to the existing clients in the wild. First, we conducted a measurement study on the campus networks' existing LibreOffice mirror server. Figure 13 shows the CDF of the percentage of clients that failed to complete their downloads (every 3 minutes), with or without NetShuffle enabled, and it shows negligible difference. For both cases, our measurements directly rely on our server logs, where downloads with HTTP error codes in the 4xx and 5xx ranges are considered failures. Note that we exclude various forms of non-errors, such as error 404s paired with download URLs that are not served whatsoever by the campus network.

Table 3 presents additional statistics. Since we captured results for NetShuffle and the baseline at different times, we examined the number of unique clients and aggregate transfer volumes to make sure that our results in Figure 13 are of comparable volume. We also conducted an `ab` benchmark test against the mirror server during the two time windows, targeting the homepage of the mirror URL from a controlled client. Since the metrics with NetShuffle are nearly equivalent to the baseline, this shows that the mirror service was indeed accessible even with NetShuffle enabled (at least from the viewpoint of our controlled client).

Next, we conducted a separate test using the RIPE Atlas globally distributed infrastructure to measure transparent forwarding, from the viewpoint of the clients themselves. We used 100 unique RIPE Atlas probes selected uniformly from each geographic location offered by the platform to validate worldwide accessibility (e.g., China, Albania, and Russia). Each probe repeatedly issued TLS requests to our target server located within the campus network, for a period that spanned over 4 epochs (2 rounds) to verify that connectivity can be established despite the constant shuffles.

The results in Figure 14 demonstrate that the request completion times are negligibly different with and without NetShuffle. The overall completion rate is $\sim 97\%$ for both NetShuffle and the baseline due to instability with the probing infrastructure; we reran probes that had any failures until only successes were observed throughout the period.

Circumvention efficacy. As a final experiment, we deployed a dozen vanilla Tor bridges on AWS (without NetShuffle) and in our campus network (with and without NetShuffle enabled). We intentionally instantiated *unobfuscated* bridges (as opposed to more covert censorship circumvention systems) to make it as easy as possible for the censor to detect and block their use.



(a) Apache HTTP web server.
(b) Hysteria proxy.
(c) Redis K-V store.
(d) Apache HTTPS web server.
(e) SCP bulk file download.
(f) Shadowsocks proxy.

Figure 12: NetShuffle operates transparently across multiple epochs, demarcated by vertical lines, for a variety of services.

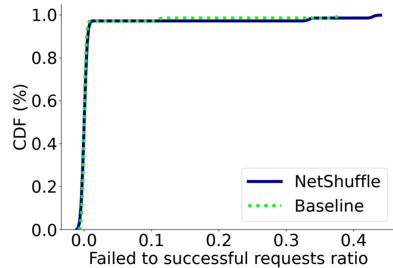


Figure 13: NetShuffle transparently serves large traffic volumes to LibreOffice clients.

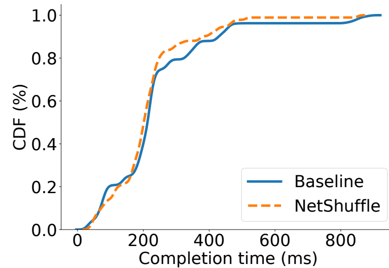


Figure 14: TLS request completion times from 100 RIPE Atlas probes.

TABLE 3: NetShuffle processes similar amounts of LibreOffice traffic (vs. baseline) at near equivalent performance.

Property	NetShuffle	Baseline
Unique clients	335	390
Transfer volume	110.90 GB	129.11 GB
Window length	2.5 hours	2.5 hours
`ab`: doc. length	1152 bytes	1152 bytes
`ab`: avg. time/req.	173.25 ms	173.53 ms
`ab`: transfer rate	7.56 KBps	7.57 KBps

We also created virtual machines in two datacenters located in China (Beijing and Shanghai) that hosted Tor clients that attempted to connect to our bridges. Our baseline experiment disabled NetShuffle, and we measured the time it took for the censor to block the bridge addresses. We then enabled NetShuffle for the bridges located in our campus network, and conducted the same experiment.

In the baseline case, the average time to blocking is 13 minutes, but when protected by NetShuffle, the bridges never experienced any blocking. This indicates that our shuffle speed is faster than the Great Firewall (GFW) is able (or willing) to probe bridges. We further repeated the experiments with a range of NetShuffle epochs ranging from 1 to 5 minutes. This suggests that even for vanilla Tor bridges, which are known to be easy to block (and indeed targeted by the censor), NetShuffle can raise them to a new degree of unblockability out of the box.

A limitation of evaluating any newly proposed censorship resistance system (including NetShuffle) against a mature censorship system (e.g., the GFW) is that the latter necessarily has no knowledge of the former. We do not claim that our current success at evading the GFW is indicative of future *evasion* against a NetShuffle-aware GFW or against other censors. However, we highlight two takeaways from our live deployment experiment: as discussed in the paper, NetShuffle is designed to be robust against a censor who is aware of its usage. Even if the GFW did block our bridge, it would need to keep pace with the shuffling to prevent future accesses. The significant time lapse between our accesses and the GFW's probes suggests that the GFW at least currently does not have the resources (or is unwilling) to quickly perform its probe and blocking operations. In order to match our current shuffling speed, the GFW needs to operate an order of magnitude faster, so NetShuffle raises the bar for censorship significantly. Second, we stress that deploying NetShuffle was easy. Unlike decoy routing systems that are difficult to deploy due to their dependence upon a cooperating core service (e.g., an ISP), we were able to easily switch to NetShuffle and use it for over a month at zero cost on our edge network (besides the switch purchase costs). This makes opt-ins much easier and can engage a larger support base to further raise the censorship cost.

7. Security Analysis

Next, we discuss several potential attacks to NetShuffle and describe how we can foil or mitigate such attacks.

Controller buffer flooding. An adversary may attempt to trigger a denial-of-service attack by sending a large number of packets for compacted HP services without completing

a TCP connection handshake. This will cause excessive buffering at the switch control plane. To mitigate this attack, NetShuffle monitors connections' 4-tuple access frequency until a configurable access threshold is met and can then a) reset a connection and drop its associated buffers, b) enforce per-client rate limiting, or c) issue compaction changes, where the controller changes the single HP compacted ⟨ExtIP, IntPort⟩ pair used at the next epoch, so that their packets do not need to be mediated anymore.

Connection table flooding. An adversary can also attempt to overwhelm the Conn tables by flooding the switch with packets with unique 4-tuples so each packet triggers an entry installation. If left unattended, this can quickly consume all available table space. To mitigate this, NetShuffle places inactive connections (after observing a source IP for more than a given threshold) into a blocklist for a pre-determined amount of time, using the same control plane mechanisms for managing hardware idle-timeouts. The controller can directly terminate blocklisted flows through the analysis of the digests before installing them to the Conn tables.

Finally, we note the above SYN flooding and other types of DoS attacks can be addressed by well-studied techniques developed in programmable switches [95], [96]. We leave their integration as interesting future work.

Subdomain blocking. Censors can register as a client and obtain proxy identifiers. However, it is hard for a censor to build an effective blocklist. First, advanced proxy distribution schemes as well as probe-resistant proxy implementations are well-studied, and they can be applied to NetShuffle to mitigate against enumeration. Second, given the large space of subdomain names (see §4.1) or even new second-level domains, NetShuffle provides a steady supply of new subdomain names for shuffle proxies, turning subdomain name enumeration into a continuous grind for the censor.

To better understand this, we analyzed the certificate transparency logs provided by Scheitle et al. [67] and found over 9 million distinct subdomain names that are deployed in operation. To illustrate the frequent use of infrequent domain names, we find that more than 25% of subdomains used in US university campuses and Fortune 500 companies (who are reasonable candidate edge networks for NetShuffle) fall outside the top-600K most frequently used names. Thus, censors cannot easily block subdomain names that are uncommonly seen without potentially incurring high collateral damage. Finally, studies show that censors typically choose not to use allowlists [9], [97] due to the added complexity of maintaining up-to-date allowlists as new Internet services are constantly added or updated.

Full edge network blocking. While it is not easy to gauge a censor's perceived collateral for taking down an entire network, we believe that the common assumption about the censor's aversion to collateral damage applies here as well. Like core networks, edge networks host many domains, services and IP ranges that are useful to the censored region. It is known that censors prefer fine-grained approaches (e.g., keyword blocking, and targeted DNS and IP based blocking) [98]; in fact, techniques have been developed specifically to avoid overblocking [41] as it results in negative economic impact [99]–[102]. Censors blocking entire IP ranges has been reported before, but only in a temporary manner (e.g., during elections) [99], [103] and intentionally targeting a complete Internet shutdown instead of narrowly at specific circumvention services. Moreover, even if a censoring nation does not make use of a given edge network's services, their international peers may. Analogous to the risk of spillover DNS pollution [104]–[106], the potential unintended consequences of blocking an entire edge network can act as an additional deterrent. Censoring nations typically have their own censorship policies and do not behave congruently [98], [107], [108]. Thus, even if one nation decides to block a given edge network, we expect there will be other nations that want to avoid overblocking.

8. Discussion

We hope NetShuffle will start a discussion in our community on edge networks' role in censorship resistance.

Collateral damage quantification. Although censors' prior behavior gives us some insights into how they might respond to techniques deployed at the edge (see §7), a precise quantification of collateral damage via measurement studies would be useful as future work. Such studies require significant operational experience and extensive follow-up research on edge-based resistance systems, since they are an underexplored support base and existing data is scant. As a comparison, after the initial publication of DR in 2011, significant subsequent studies [27], [109]–[112] were needed to quantify its collateral damage. (Even if censors are capable of rerouting traffic around certain core networks to reduce collateral damage, edge-based resistance systems are not as easily bypassed by routing-based evasion.)

Future deployment. By providing a solution that is easy to deploy, performant, and functionally correct, NetShuffle expands the possible deployment locations of censorship-resistance systems beyond end-user proxies and core networks, thus potentially promoting follow-up studies of edge networks as an additional deployment base to expand the anti-censorship arsenal. In terms of incentives, since NetShuffle targets a new support base at the edge, which is currently a blank space, a reliable assessment of their full deployment incentives is best done by the test of time. This is analogous to the trajectory of DR from its initial conception [5] to a real-world deployment [44] over several years. We believe NetShuffle could follow a similar if not faster trajectory, since we were able to deploy the first NetShuffle instance with one person/day effort.

9. Related Work

In-network traffic deflection. Decoy routing (DR) [5], [8], [10], [24], [113] envisions cooperative ISPs deploying traffic redirection routers within their networks. Unfortunately, most proposed DR systems operate at software speeds, comporting significant costs for network operators [9].

A few approaches aim to combat this overhead. Siegebreaker [7] leverages software-defined networks to breakdown DR functionality over multiple network elements and defray these costs, but requires extensive modifications to proxies and wastes bandwidth on regular hosts. Conjure [9] allows clients to connect to a large amount of "free" IP addresses managed by cooperating ISPs. However, at the edge, such free IP addresses are harder to come by (see §3). NetShuffle is a defense that is tailored for the edge and operates within minimal address space requirements. A recent idea, REDACT [114], argues for deploying decoy routers at the border of public cloud data centers, but it does not design a concrete system. Moreover, its goal is to lessen the impact of routing around decoys attacks [27], [109]–[111], significantly different from NetShuffle.

Shuffle-based reconnaissance defenses. This class of techniques aims to prevent an attacker from performing a reliable reconnaissance operation on a target network infrastructure, e.g., by periodically shuffling the server's network address [115]–[117]. The high-level idea is also similar in spirit to DNS fast-fluxing [118]–[121] attacks. Recent work aimed at protecting user privacy, such as MIMIQ [122] and RAVEN [123], prevent network adversaries from identifying the source of a given flow within an edge network or correlating several flows. Both tools require QUIC protocol's connection migration capabilities to perform frequent changes to clients' IP addresses without breaking the connections. Similar to RAVEN, NetShuffle also relies on programmable switches for IP shuffling, but toward a different goal – to prevent adversaries from identifying to which hosts within the edge network an external client is connected. NetShuffle does not aim to prevent an adversary from associating which packets belong to the same flow. Another related work, SPINE [124], prevents an intermediate AS from inferring which two hosts are communicating across two trusted, SPINE-enabled ASes. It leverages programmable switches to encrypt packets' source and destination IP addresses, as well as TCP sequence and acknowledgment numbers to thwart the grouping of packets into a given flow. In contrast, NetShuffle targets a censorship circumvention setting and aims at making proxy services hard to block.

10. Conclusion

By separating proxy services from their identifiers, and by performing continuous network shuffles for moving target defense, NetShuffle raises regular proxies to a new degree of unblockability. It further engages a support base—edge networks—which so far have received little attention in censorship evasion techniques. NetShuffle leverages emerging programmable switches for in-network execution that is transparent to the network deployment and services. Our testbed and live deployment results show that NetShuffle integrates with real-world services seamlessly and executes with negligible overhead while performing the shuffle.

Acknowledgments

We thank Lior Zeno of the Technion Accelerated Computing Systems Lab (ACSL) for his unwavering guidance and insight on all aspects related to the Tofino switch hardware, and the anonymous reviewers at IEEE S&P 2024 for their helpful comments and suggestions. This work was partially funded by the National Science Foundation through grants CNS-1704189, CNS-1801884, CNS-1942219, CNS-2016727, CNS-2106388, CNS-2106751, and CNS-2214272; a VMware Early Career Faculty Grant; the Georgetown University Callahan Family Chair Fund; and NSERC under grant RGPIN-2023-03304.

References

[1] S. Hellmeier, R. Cole, S. Grahn, P. Kolvani, J. Lachapelle, A. Lührmann, S. F. Maerz, S. Pillai, and S. I. Lindberg, "State of the world 2020: autocratization turns viral," *Democratization*, vol. 28, no. 6, pp. 1053–1074, 2021.

[2] Freedom House, "Freedom on the net. the rise of digital authoritarianism," *Washington, DC*, 2018, https://freedomhouse.org/sites/default/files/2020-02/10192018_FOTN_2018_Final_Booklet.pdf.

[3] C. Bocovich and I. Goldberg, "Secure asymmetry and deployability for decoy routing systems." *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 3, pp. 43–62, 2018.

[4] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, 2004.

[5] A. Houmansadr, G. T. Nguyen, M. Caesar, and N. Borisov, "Cirripede: Circumvention infrastructure using router redirection with plausible deniability," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 187–200.

[6] E. Wustrow, C. M. Swanson, and J. A. Halderman, "TapDance: End-to-Middle Anticensorship without Flow Blocking," in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 159–174.

[7] P. K. Sharma, D. Gosain, H. Sagar, C. Kumar, A. Dogra, V. Naik, H. Acharya, and S. Chakravarty, "Siegebreaker: An sdn based practical decoy routing system." *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 3, pp. 243–263, 2020.

[8] E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman, "Telex: Anticensorship in the network infrastructure," in *Proceedigns of the 20th USENIX Security Symposium*, 2011.

[9] S. Frolov, J. Wampler, S. C. Tan, J. A. Halderman, N. Borisov, and E. Wustrow, "Conjure: Summoning proxies from unused address space," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2215–2229.

[10] J. Karlin, D. Ellard, A. W. Jackson, C. E. Jones, G. Lauer, D. P. Mankins, and W. T. Strayer, "Decoy routing: Toward unblockable internet communication," in *USENIX workshop on free and open communications on the Internet (FOCI 11)*, 2011.

[11] "Lantern," https://getlantern.org/.

[12] "Psiphon," http://psiphon.ca/.

[13] M. Wei, "Domain shadowing: Leveraging content delivery networks for robust blocking-resistant communications," in *Proceedings of the 30th USENIX Security Symposium*, 2021, pp. 3327–3343.

[14] R. MacKinnon, "China's censorship 2.0: How companies censor bloggers," *First Monday*, 2009.

[15] E. Morozov, "Liberation technology: whither internet control?" *Journal of Democracy*, vol. 22, no. 2, pp. 62–74, 2011.

[16] P. M. Figliola, *Promoting Global Internet Freedom: Policy and Technology*. Congressional Research Service, 2013, vol. 23.

[17] "[torproject] Tor blocking events in Belarus from 2020-2021," https://gitlab.torproject.org/tpo/anti-censorship/censorship-analysis/-/blob/main/reports/2020/belarus/2020-belarus-report.md.

[18] Tor Project, Inc., "Tor Manual: Bridges," https://tb-manual.torproject.org/bridges/#using-moat, 2022.

[19] R. Dingledine, "Research problems: Ten ways to discover Tor bridges," https://blog.torproject.org/research-problems-ten-ways-discover-tor-bridges/, Tor, Blog Post, 2011.

[20] "How the great firewall of china is blocking tor," in *Proceedings of the 2nd USENIX Workshop on Free and Open Communications on the Internet*, Bellevue, WA, 2012.

[21] M. Nasr, S. Farhang, A. Houmansadr, and J. Grossklags, "Enemy at the gateways: Censorship-resilient proxy distribution using game theory," in *Proceedings of the* 26th *Annual Network & Distributed System Security Symposium*, 2019.

[22] A. Dunna, C. O'Brien, and P. Gill, "Analyzing china's blocking of unpublished Tor bridges," in *Proceedings of the 8th USENIX Workshop on Free and Open Communications on the Internet*, 2018.

[23] D. Fifield, C. Lan, R. Hynes, P. Wegmann, and V. Paxson, "Blocking-resistant communication through domain fronting." *Proceedings on Privacy Enhancing Technologies*, vol. 2015, no. 2, pp. 46–64, 2015.

[24] B. VanderSloot, S. Frolov, J. Wampler, S. C. Tan, I. Simpson, M. Kallitsis, J. A. Halderman, N. Borisov, and E. Wustrow, "Running refraction networking for real," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 4, 2020.

[25] "[tor-project] domain fronting to app engine stopped working," https://gitlab.torproject.org/legacy/trac/-/issues/25804.

[26] "Verge: Amazon web services starts blocking domain-fronting, following google's lead," https://www.theverge.com/2018/4/30/17304782/amazon-domain-fronting-google-discontinued.

[27] D. Gosain, A. Agarwal, and S. Chakravarty, "The devil's in the details: Placing decoy routers in the internet," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, 2017, pp. 577–589.

[28] C. MacCarthaigh, "[AWS Security Blog] Enhanced domain protections for Amazon CloudFront requests," https://aws.amazon.com/blogs/security/enhanced-domain-protections-for-amazon-cloudfront-requests/, 2018.

[29] S. Gallagher, "Gogle Disables "Domain Fronting" Capability Used to Evade Censors," Ars Technica, https://arstechnica.com/information-technology/2018/04/google-disables-domain-fronting-capability-used-to-evade-censors/, 2018.

[30] "Microsoft: Securing our approach to domain fronting within Azure," https://www.microsoft.com/en-us/security/blog/2021/03/26/securing-our-approach-to-domain-fronting-within-azure/.

[31] J. Holowczak and A. Houmansadr, "Cachebrowser: Bypassing chinese censorship without proxies using cached content," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, Denver, Colorado, USA, 2015, pp. 70–83.

[32] "Wedge 100bf-32x 100gbe data center switch," https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=335.

[33] B. Tian, J. Gao, M. Liu, E. Zhai, Y. Chen, Y. Zhou, L. Dai, F. Yan, M. Ma, M. Tang *et al.*, "Aquila: a practically usable verification system for production-scale programmable data planes," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2021, pp. 17–32.

[34] T. Pan, N. Yu, C. Jia, J. Pi, L. Xu, Y. Qiao, Z. Li, K. Liu, J. Lu, J. Lu *et al.*, "Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches," in *Proceedings of the ACM Special Interest Group on Data Communication Conference*, 2021, pp. 194–206.

[35] F. Douglas, W. P. Rorshach, W. Pan, and M. Caesar, "Salmon: Robust proxy distribution for censorship circumvention." *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 4, pp. 4–20, 2016.

[36] Y. Sovran, A. Libonati, and J. Li, "Pass it on: social networks stymie censors." in *Proceedings of the 7th international conference on Peer-to-peer systems*, 2008.

[37] Z. Zhang, W. Zhou, and M. Sherr, "Bypassing Tor exit blocking with exit bridge onion services," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Virtual Event, USA, 2020, pp. 3–16.

[38] L. Tulloch and I. Goldberg, "Lox: Protecting the social graph in bridge distribution," *Proceedings on Privacy Enhancing Technologies*, vol. 2023, no. 1, 2023.

[39] Q. Wang, Z. Lin, N. Borisov, and N. Hopper, "rBridge: User reputation based Tor bridge distribution with privacy preservation." in *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, 2013.

[40] D. Fifield, N. Hardison, J. Ellithorpe, E. Stark, D. Boneh, R. Dingledine, and P. Porras, "Evading censorship with browser-based proxies," in *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 2012, pp. 239–258.

[41] D. Fifield, *Threat modeling and circumvention of Internet censorship*. University of California, Berkeley, 2017, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-225.pdf.

[42] ——, "Turbo tunnel, a good way to design censorship circumvention protocols," in *Proceedings of the 10th USENIX Workshop on Free and Open Communications on the Internet*, 2020.

[43] K. MacMillan, J. Holland, and P. Mittal, "Evaluating snowflake as an indistinguishable censorship circumvention tool," https://arxiv.org/abs/2008.03254, 2020.

[44] S. Frolov, E. Wustrow, F. Douglas, W. Scott, A. McDonald, B. VanderSloot, R. Hynes, A. Kruger, M. Kallitsis, D. G. Robinson, S. Schultze, N. Borisov, and J. A. Halderman, "An ISP-Scale Deployment of TapDance," in *Proceedings of the Applied Networking Research Workshop*, Montreal, QC, Canada, 2018.

[45] B. Lawson and J. Rexford, "Decoy switching: Circumventing censorship with emerging switch hardware," https://www.cs.princeton.edu/~jrex/papers/written_final_report.pdf.

[46] H. Zolfaghari and A. Houmansadr, "Practical censorship evasion leveraging content delivery networks," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Vienna, Austria, 2016, pp. 1715–1726.

[47] "Signal: A letter from amazon," https://signal.org/blog/looking-back-on-the-front/.

[48] "AWS: Enhanced Domain Protections for Amazon CloudFront Requests," https://aws.amazon.com/blogs/security/enhanced-domain-protections-for-amazon-cloudfront-requests/.

[49] D. Gosain, M. Mohindra, and S. Chakravarty, "Too close for comfort: Morasses of (anti-) censorship in the era of CDNs," *Proceedings on Privacy Enhancing Technologies*, vol. 2021, no. 2, pp. 173–193, 2021.

[50] "Tor blog: The Trouble with CloudFlare," https://blog.torproject.org/trouble-cloudflare/.

[51] "[CNBC] Amazon drops Parler from its web hosting service, citing violent posts," https://www.cnbc.com/2021/01/09/amazon-drops-parler-from-its-web-hosting-service.html.

[52] "[Cloudflare blog] Why We Terminated Daily Stormer," https://blog.cloudflare.com/why-we-terminated-daily-stormer/.

[53] "Tor: Fact Sheet: Cloudflare and the Tor Project," https://people.torproject.org/~lunar/20160331-CloudFlare_Fact_Sheet.pdf.

[54] M. Ziv, L. Izhikevich, K. Ruth, K. Izhikevich, and Z. Durumeric, "ASdb: a system for classifying owners of autonomous systems," in *Proceedings of the ACM Internet Measurement Conference*, 2021, pp. 703–719.

[55] S. Frolov and E. Wustrow, "The use of TLS in Censorship Circumvention," in *NDSS*, 2019.

[56] "Tor: Bridges," https://tb-manual.torproject.org/bridges/.

[57] "Snowflake," https://snowflake.torproject.org, accessed: 2022-05-13.

[58] L. Zeno, D. R. Ports, J. Nelson, D. Kim, S. Landau-Feibish, I. Keidar, A. Rinberg, A. Rashelbach, I. De-Paula, and M. Silberstein, "{SwiSh}: Distributed shared state abstractions for programmable switches," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 171–191.

[59] "[Akamai] IPv6 Adoption Visualization," https://www.akamai.com/internet-station/cyber-attacks/state-of-the-internet-report/ipv6-adoption-visualization.

[60] "[Cisco] 6lab - The place to monitor IPv6 adoption," https://6lab.cisco.com/stats/.

[61] "[Google] IPv6 statistics," https://www.google.com/intl/en/ipv6/statistics.html.

[62] S. Jia, M. Luckie, B. Huffaker, A. Elmokashfi, E. Aben, K. Claffy, and A. Dhamdhere, "Tracking the deployment of IPv6: Topology, routing and performance," *Computer Networks*, vol. 165, p. 106947, 2019.

[63] S. Frolov and E. Wustrow, "{HTTPT}: A {Probe-Resistant} proxy," in *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20)*, 2020.

[64] B. Birtel and C. Rossow, "Slitheen++: Stealth TLS-based decoy routing," in *Proceedings of the 10th USENIX Workshop on Free and Open Communications on the Internet*, 2020.

[65] S. Frolov, J. Wampler, and E. Wustrow, "Detecting probe-resistant proxies." in *Proceedings of the* 27th *Annual Network and Distributed System Security Symposium*, 2020.

[66] "shadowsocks-libev," https://github.com/shadowsocks/shadowsocks-libev.

[67] Q. Scheitle, O. Gasser, T. Nolte, J. Amann, L. Brent, G. Carle, R. Holz, T. C. Schmidt, and M. Wählisch, "The rise of certificate transparency and its implications on the internet ecosystem," in *Proceedings of the Internet Measurement Conference 2018*, 2018, pp. 343–349.

[68] L. Izhikevich, G. Akiwate, B. Berger, S. Drakontaidis, A. Ascheman, P. Pearce, D. Adrian, and Z. Durumeric, "ZDNS: a fast DNS toolkit for internet measurement," in *Proceedings of the 22nd ACM Internet Measurement Conference*, 2022, pp. 33–43.

[69] P. Mockapetris, "Domain Names: Implementation and Specification," IETF, RFC 1035, 1987.

[70] P. Richter, M. Allman, R. Bush, and V. Paxson, "A primer on IPv4 scarcity," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 2, pp. 21–31, 2015.

[71] A. Dainotti, K. Benson, A. King, K. Claffy, M. Kallitsis, E. Glatz, and X. Dimitropoulos, "Estimating internet address space usage through passive measurements," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 1, pp. 42–49, 2013.

[72] A. Dainotti, K. Benson, A. King, B. Huffaker, E. Glatz, X. Dimitropoulos, P. Richter, A. Finamore, and A. C. Snoeren, "Lost in space: improving inference of IPv4 address space utilization," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 6, pp. 1862–1876, 2016.

[73] S. Zander, L. L. Andrew, and G. Armitage, "Capturing ghosts: Predicting the used IPv4 space by inferring unobserved addresses," in *Proceedings of the ACM Internet Measurement Conference*, 2014, pp. 319–332.

[74] M. Fayed, L. Bauer, V. Giotsas, S. Kerola, M. Majkowski, P. Odintsov, J. Sitnicki, T. Chung, D. Levin, A. Mislove, C. A. Wood, and N. Sullivan, "The Ties that un-Bind: Decoupling IP from web services and sockets for robust addressing agility at CDN-scale," in *Proceedings of the 2021 ACM SIGCOMM Conference*, 2021, pp. 433–446.

[75] P. Gigis, M. Calder, L. Manassakis, G. Nomikos, V. Kotronis, X. Dimitropoulos, E. Katz-Bassett, and G. Smaragdakis, "Seven years in the life of hypergiants' off-nets," in *Proceedings of the ACM Special Interest Group on Data Communication Conference*, 2021, pp. 516–533.

[76] R. P. Singh, T. Brecht, and S. Keshav, "IP address multiplexing for VEEs," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 36–43, 2014.

[77] S. Bai, H. Kim, and J. Rexford, "Passive OS fingerprinting on commodity switches," in *Proceedings of the 8th IEEE International Conference on Network Softwarization*, 2022, pp. 264–268.

[78] "TLSFingerprint.io," https://tlsfingerprint.io/.

[79] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. Maguire Jr, P. Papadimitratos, and M. Chiesa, "A high-speed load-balancer design with guaranteed per-connection-consistency," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, 2020, pp. 667–683.

[80] "NetShuffle Source Code." 2023, https://github.com/patrickkon/NetShuffle.

[81] G. C. Moura, J. Heidemann, M. Müller, R. d. O. Schmidt, and M. Davids, "When the dike breaks: Dissecting DNS defenses during DDoS (extended)," in *Proceedings of the ACM Internet Measurement Conference*, 2018.

[82] T. Hernandez-Quintanilla, E. Magaña, D. Morató, and M. Izal, "On the reduction of authoritative dns cache timeouts: Detection and implications for user privacy," *Journal of Network and Computer Applications*, vol. 176, p. 102941, 2021.

[83] A. Klein and B. Pinkas, "Dns cache-based user tracking." in *NDSS*, 2019.

[84] D. Fifield and L. Tsai, "Censors' delay in blocking circumvention proxies," *arXiv preprint arXiv:1605.08808*, 2016.

[85] G. C. Moura, J. Heidemann, R. d. O. Schmidt, and W. Hardaker, "Cache me if you can: Effects of DNS time-to-live," in *Proceedings of the Internet Measurement Conference*, 2019, pp. 101–115.

[86] "CloudFlare DNS," https://www.cloudflare.com/dns/.

[87] "Google Domains," https://domains.google.com/registrar/.

[88] U. Bauknecht and T. Enderle, "An investigation on core network latency," in *Proceedings of the 30th IEEE International Telecommunication Networks and Applications Conference*, 2020, pp. 1–6.

[89] J. Kwon, J. A. García-Pardo, M. Legner, F. Wirz, M. Frei, D. Hausheer, and A. Perrig, "SCIONLab: A next-generation Internet testbed," in *Proceedings of the 28th IEEE International Conference on Network Protocols*, 2020, pp. 1–12.

[90] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 15–28.

[91] "Intel® tofino™ 2," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[92] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 123–137.

[93] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, "Generic external memory for switch data planes," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, 2018, pp. 1–7.

[94] "Hysteria.network," https://github.com/apernet/hysteria.

[95] Z. Liu, H. Namkung, G. Nikolaidis, J. Lee, C. Kim, X. Jin, V. Braverman, M. Yu, and V. Sekar, "Jaqen: A {High-Performance}{Switch-Native} approach for detecting and mitigating volumetric {DDoS} attacks with programmable switches," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3829–3846.

[96] D. Scholz, S. Gallenmüller, H. Stubbe, and G. Carle, "SYN flood defense in programmable data planes," in *Proceedings of the 3rd P4 Workshop in Europe*, 2020, pp. 13–20.

[97] C. Bocovich, *Recipes for Resistance: A Censorship Circumvention Cookbook*. University of Waterloo, 2018, http://hdl.handle.net/10012/13595.

[98] R. Sundara Raman, P. Shenoy, K. Kohls, and R. Ensafi, "Censored planet: An internet-wide, longitudinal censorship observatory," in *Proceedings of the ACM SIGSAC conference on computer and communications security*, 2020, pp. 49–66.

[99] "[Carnegie Endowment] Government Internet Shutdowns Are Changing. How Should Citizens and Democracies Respond?" https://carnegieendowment.org/2022/03/31/government-internet-shutdowns-are-changing.-how-should-citizens-and-democracies-respond-pub-86687.

[100] "[Bloomberg] World's Worst Internet Clampdown Cost Myanmar $3 Billion in 2021," https://www.bloomberg.com/news/articles/2022-01-04/world-s-worst-internet-clampdown-cost-myanmar-3-billion-in-2021.

[101] S. Woodhams and S. Migliano, "[Top10VPN] Cost of Internet Shutdowns 2021: Government Internet Shutdowns Cost $5.5 Billion in 2021," https://www.top10vpn.com/research/cost-of-internet-shutdowns/2021/.

[102] "[Deloitte] The economic impact of disruptions to Internet connectivity," https://globalnetworkinitiative.org/wp-content/uploads/2016/10/GNI-The-Economic-Impact-of-Disruptions-to-Internet-Connectivity.pdf.

[103] "[accessnow] #KeepItOn update: who is shutting down the internet in 2021?" https://www.accessnow.org/who-is-shutting-down-the-internet-in-2021/.

[104] S. Cho, R. Nithyanand, A. Razaghpanah, and P. Gill, "A churn for the better: Localizing censorship using network-level path churn and network tomography," in *Proceedings of the 13th International Conference on emerging networking experiments and technologies*, 2017, pp. 81–87.

[105] N. P. Hoang, A. A. Niaki, J. Dalek, J. Knockel, P. Lin, B. Marczak, M. Crete-Nishihata, P. Gill, and M. Polychronakis, "How great is the great firewall? measuring China's DNS censorship," *arXiv preprint arXiv:2106.02167*, 2021.

[106] R. Padmanabhan, A. Filastò, M. Xynou, R. S. Raman, K. Middleton, M. Zhang, D. Madory, M. Roberts, and A. Dainotti, "A multi-perspective view of internet censorship in Myanmar," in *Proceedings of the ACM SIGCOMM 2021 Workshop on Free and Open Communications on the Internet*, 2021, pp. 27–36.

[107] "[Freedom House] Countering an Authoritarian Overhaul of the Internet," https://freedomhouse.org/report/freedom-net/2022/countering-authoritarian-overhaul-internet.

[108] A. A. Niaki, S. Cho, Z. Weinberg, N. P. Hoang, A. Razaghpanah, N. Christin, and P. Gill, "ICLab: A global, longitudinal internet censorship measurement platform," in *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020, pp. 135–151.

[109] M. Nasr and A. Houmansadr, "Game of decoys: Optimal decoy routing through game theory," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, Vienna, Austria, 2016, pp. 1727–1738.

[110] M. Nasr, H. Zolfaghari, and A. Houmansadr, "The waterfall of liberty: Decoy routing circumvention that resists routing attacks," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2037–2052.

[111] M. Schuchard, J. Geddes, C. Thompson, and N. Hopper, "Routing around decoys," in *Proceedings of the ACM Conference on Computer and Communications Security*, Raleigh, North Carolina, USA, 2012, pp. 85–96.

[112] A. Houmansadr, E. L. Wong, and V. Shmatikov, "No direction home: The true cost of routing around decoys." in *NDSS*. Citeseer, 2014.

[113] C. Bocovich and I. Goldberg, "Slitheen: Perfectly imitated decoy routing through traffic replacement," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Vienna, Austria, 2016, pp. 1702–1714.

[114] A. Devraj, L. Wang, and J. Rexford, "REDACT: refraction networking from the data center," *ACM SIGCOMM Computer Communication Review*, vol. 51, no. 4, pp. 15–22, 2021.

[115] T. E. Carroll, M. Crouse, E. W. Fulp, and K. S. Berenhaut, "Analysis of network address shuffling as a moving target defense," in *Proceedings of the IEEE international conference on communications*, 2014, pp. 701–706.

[116] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow random host mutation: transparent moving target defense using software defined networking," in *Proceedings of the 1st workshop on Hot topics in software defined networks*, 2012, pp. 127–132.

[117] G. Cai, B. Wang, X. Wang, Y. Yuan, and S. Li, "An introduction to network address shuffling," in *Proceedings of the 18th IEEE international conference on advanced communication technology*, 2016, pp. 185–190.

[118] J. Gardiner, M. Cova, and S. Nagaraja, "Command & control: Understanding, denying and detecting-a review of malware c2 techniques, detection and defences," *arXiv preprint arXiv:1408.1136*, 2014.

[119] T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling, "Measuring and detecting fast-flux service networks." in *Proceedings of the 16th Annual Network & Distributed System Security Symposium*, 2008.

[120] J. Nazario and T. Holz, "As the net churns: Fast-flux botnet observations," in *Proceedings of the 3rd International Conference on Malicious and Unwanted Software*, 2008, pp. 24–31.

[121] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, "Your botnet is my botnet: analysis of a botnet takeover," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 635–647.

[122] Y. Govil, L. Wang, and J. Rexford, "MIMIQ: Masking IPs with migration in QUIC," in *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2020.

[123] P. M. Liang Wang, Hyojoon Kim and J. Rexford, "RAVEN: Stateless Rapid IP Address Variation for Enterprise Networks," *Proceedings on Privacy Enhancing Technologies*, vol. 2023, no. 3, 2023.

[124] T. Datta, N. Feamster, J. Rexford, and L. Wang, "Spine: Surveillance protection in the network elements," in *9th USENIX Workshop on Free and Open Communications on the Internet*, 2019.

[125] "nsupdate," https://linux.die.net/man/8/nsupdate.

Appendix A. Validating browser and OS DNS TTL Caching

We sought to confirm prior work's finding that common browsers (e.g., Chrome, Safari, Firefox) and OSes (e.g., MacOS, Linux, and Windows) generally upper bound additional DNS TTL caching to one minute [82], [83]. Violations of DNS cache TTL policies—in particular, instances in which cached DNS responses are used beyond their specified lifetimes—could endanger NetShuffle's transparency goals since affected clients may use stale IP addresses that would no longer be associated with the intended service.

We confirmed that Linux 4.15, Linux 5.3, Windows 11, and Mac OSX 12.6 honor DNS TTLs and do not cache DNS results longer than the TTL indicates. For our experiments, we modified our TTLs from 5 minutes down to 1 second. To measure this, we issued DNS queries from different platforms to the recursive DNS resolver serving our testbed, initially with a TTL of 5 minutes, to populate the client's local DNS cache, and then checked if the client environment respects the TTL as specified in the DNS record (whether it would issue a new DNS query when the domain is requested, as soon as the TTL expires). We then repeat the experiment by updating the DNS record (using the `nsupdate` [125] utility) on our network's authoritative DNS server (§6.1) with a TTL value that is decremented by 1, until we reach a TTL of 1 second. We repeated the test for 30 trials for further confirmation.

Prior work notes that browsers sometimes implement their own DNS caches [81]–[83]. Using an approach similar to our aforementioned experiment, we tested the DNS caching behavior of Chrome (on Windows, Mac, and Android), Firefox (on Windows and Mac), Edge (on Windows and Mac), and Safari (on Mac). In more detail, for each browser, we fetched a webpage under our control. We experimented with different TTL values for the authoritative record for that domain. When the TTL expired, we mapped the domain to a different IP address. We then evaluated whether subsequent fetches from the browser (without restarting it) would honor the TTL (in which case a new lookup would occur and the browser would successfully render the page) or use a stale cached resolution (that would result in an unsuccessful page fetch). Our evaluation used TTL values down to one minute since prior work reported a one minute upper bound for most browsers' DNS caching [82], [83]. In all cases, we found that the browser successfully rendered the page, indicating their support of DNS TTLs of one minute.

Appendix B.
Meta-Review

B.1. Summary

This paper presents "NetShuffle", a censorship circumvention system that discourages a censor from blocking a prohibited site by putting it behind a shared public access IP, that the censor might not be willing to block because of "collateral damage". The technique is similar to domain fronting, but differs in implementation and in the size and amount of networks that could use the approach.

B.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field
- Creates a New Tool to Enable Future Science

B.3. Reasons for Acceptance

1) This work presents a new point in the design space of censorship circumvention technologies.
2) The output of this work could enable future science on how to efficiently deploy/utilize proxies.
3) The work could support subsequent research towards understanding how to optimize deployments of such systems.

B.4. Noteworthy Concerns

The core contribution of this work relies on the claim that censors would be less willing to block the smaller networks using the "NetShuffle" approach, as opposed to larger networks using the existing and established "domain fronting" approach. This claim is asserted but not justified in the paper, meaning the efficacy of the approach is currently unknown. Understanding the ultimate impact of this work would therefore require wider deployment and follow-up evaluation of "NetShuffle" systems.

Appendix C.
Response to the Meta-Review

Our argument in relation to core networks isn't that they are more likely to be blocked by censors when compared with edge networks; instead, our main argument is that core networks are not perfect and have their own disadvantages (e.g., core networks can and have performed self-censorship), which we have discussed in paragraph 3 of §1 and §2.2. This motivates the search for new solutions in other domains (we advocate for an edge-network-centric solution in this paper).

However, it is true that our contribution relies on the claim that edge networks should pose some level of deterrence to censors, which we have argued for in §7 and §8.