Come as You Are: Helping Unmodified Clients Bypass Censorship with Server-side Evasion

Kevin Bock George Hughey Louis-Henri Merino
Tania Arya Daniel Liscinsky Regina Pogosian Dave Levin
University of Maryland

ABSTRACT

Decades of work on censorship evasion have resulted in myriad ways to empower clients with the ability to access censored content, but to our knowledge *all* of them have required some degree of client-side participation. Having to download and run anti-censorship software can put users at risk, and does not help the many users who do not even realize they are being censored in the first place.

In this paper, we present the first purely server-side censorship evasion strategies—11 in total. We extend a recent tool, Geneva, to automate the discovery and implementation of server-side strategies, and we apply it to four countries (China, India, Iran, and Kazakhstan) and five protocols (DNS-over-TCP, FTP, HTTP, HTTPS, and SMTP). We also perform follow-on experiments to understand why the strategies Geneva finds work, and to glean new insights into how censors operate. Among these, we find that China runs a completely separate network stack (each with its own unique bugs) for each application-layer protocol that it censors.

The server-side techniques we find are easier and safer to deploy than client-side strategies. Our code and data are publicly available.

CCS CONCEPTS

• **Social and professional topics** → **Technology and censorship**; • **General and reference** → *Measurement*;

KEYWORDS

Censorship; Geneva; Server-side

ACM Reference Format:
Kevin Bock, George Hughey, Louis-Henri Merino, Tania Arya, Daniel Liscinsky, Regina Pogosian, and Dave Levin. 2020. Come as You Are: Helping Unmodified Clients Bypass Censorship with Server-side Evasion. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. `https://doi.org/10.1145/3387514.3405889`

1 INTRODUCTION

For a client inside of a censoring regime to access censored content, it seems quite natural that the client would have to deploy *something*. Indeed, to the best of our knowledge, *all* prior work in censorship evasion has required some degree of deployment at the clients within the censoring regime. Proxies [12, 15], decoy routing [40, 41], VPNs, anonymous communication protocols [13], domain fronting [16], protocol obfuscation [24, 25, 37], and recent advances that confuse censors by manipulating packets [9, 21, 23, 36]—all of these prior solutions require various degrees of active participation on behalf of clients.

Unfortunately, active participation on the part of clients can limit the reach of censorship evasion techniques. In some scenarios, installing anti-censorship software can put users at risk [30]. For users who are willing to take on this risk, it can be difficult to *bootstrap* censorship evasion, as the anti-censorship tools themselves may be censored [4, 39]. Worse yet, there are many users who do not seek out tools to evade censorship because they do not even know they are being censored [44].

Ideally, servers located outside of a censoring regime would be able to help clients evade censorship *without the client having to install any extra software whatsoever*. If possible, this could result in a more open Internet for users who are otherwise unable (or unfamiliar with how) to access censored content.

To our knowledge, there has been *no* prior work that has explored evasion techniques that involve no client-side participation whatsoever. This is not for lack of want; rather, at first glance, it would appear that server-side-only techniques could not possibly provide a sufficient solution. To see why, let us consider all of the packets that are transmitted that lead up to an HTTP connection being censored due to the client issuing a GET request for a censored keyword. First, the client would initiate a TCP three-way handshake, during which the client sends a `SYN`, the server responds with a `SYN+ACK`, and the client responds with an `ACK`. Then, the client would send a `PSH+ACK` packet containing the HTTP request with the censored keyword, at which point the censor would tear down the connection (e.g., by injecting `RST` packets to both the client and the server). Note that *the only packet a server sends before a typical censorship event is just a* `SYN+ACK`—this would seem to leave very little room for a censorship evasion strategy.

In this paper, we present the first purely server-side censorship evasion strategies—11 in total, spanning four countries (China, India, Iran, and Kazakhstan). Like a recent string of papers [9, 21, 23, 36], these strategies do not involve a custom protocol, but rather operate by manipulating packets of existing applications, e.g., by inserting, duplicating, tampering, or dropping packets. We verify that each of these strategies (sometimes with small tweaks) work

SIGCOMM '20, August 10–14, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7955-7/20/08…$15.00
`https://doi.org/10.1145/3387514.3405889`

with completely unmodified clients running any major operating system.

To find these strategies, we make use of an existing tool, called Geneva [9], which has been shown to be able to automate the discovery of client-side strategies. While this required several modest extensions to the tool, we do not claim them as a primary contribution of this paper. Rather, our primary contributions are the discovery that server-side strategies are possible at all, and the various insights we have gained from follow-up experiments that explain *why* the strategies Geneva found work. Though the specific circumvention strategies may be patchable, the underlying insights they allowed us to glean are, we believe, more fundamental. These findings include:

- Server-side-only circumvention strategies are possible! We succeeded in finding them in every country we tested (China, India, Iran, and Kazakhstan) and for all of the protocols we were able to trigger censorship with (DNS-over-TCP, FTP, HTTP, HTTPS, and SMTP).
- The so-called Great Firewall (GFW) of China has a more nuanced "resynchronization state" than previously reported [9, 36].
- China uses *different network stacks* for each of the protocols that it censors; circumvention strategies that work for one application-layer protocol (e.g., HTTPS) do not necessarily work for another (e.g., HTTP or SMTP).

The rest of this paper is organized as follows. §2 reviews nation-state censorship and prior work. §3 empirically shows that, unfortunately, client-side techniques do not generalize to server-side. §4 presents our experiment methodology. We present 11 new server-side evasion strategies in §5, and through further examination, shed new light on the inner workings of censorship in China, India, Iran, and Kazakhstan. §6 explores our theory that censors employ different network stacks for each censored application. §7 shows that our server-side strategies work for a wide diversity of client OSes. We discuss deployment considerations in §8 and ethical considerations in §9. Finally, §10 concludes.

2 BACKGROUND AND RELATED WORK

Here, we review nation-state network censors, and we provide an overview of prior work on evading such censors.

2.1 Nation-state Censors

We focus on *nation-state-level censors.* These are very powerful entities who are able to inspect [21], inject [5], and sometimes also drop [31] traffic throughout their countries. Nation-state censors operate in two broad ways: *on-path* (man-on-the-side) or *in-path* (man-in-the-middle) [9, 36]. Our experiments span both kinds; we describe them here.

On-path Censors On-path (man-on-the-side) censors can obtain copies of packets, allowing them to overhear all communication on a connection. To determine whether or not to censor, these attackers perform deep-packet inspection (DPI) and typically look for keywords they wish to censor, such as DNS queries [5, 6, 43] or resources in HTTP GETs [11, 23, 36].

On-path censors are also able to inject packets to both ends of the connection. Because they are able to view all traffic on the connection, they can trivially inject packets that the end-hosts will accept—unlike traditional *off-path* attackers who must guess sequence numbers, query IDs, or port numbers [10, 20]. On-path censors have been observed to inject TCP `RST`s to tear down connections [3, 9, 11, 21, 23, 36, 38] and DNS lemon responses to thwart address lookup [5, 6].

To reconstruct application-layer messages and track sequence numbers, on-path censors maintain a Transmission Control Block (TCB) for *each* flow. A TCB comprises sequence numbers, received packets, and other information about the connection. A considerable amount of work has gone into modeling and understanding how censors *synchronize* and *re-synchronize* their TCB state with the ongoing connection's [23, 36]. Prior work has found, for instance, that the presence of a `SYN+ACK` packet with an incorrect acknowledgement number will cause the GFW to enter a "re-synchronization state," after which it will update its TCB using the next `SYN+ACK` packet from the server or the next data packet from the client [36]. Prior work tacitly assumed that censors enter these re-synchronization states in the same way regardless of the application-layer protocol being censored; we show in §5 that this is not the case.

Maintaining a TCB on a per-flow basis is challenging at scale, and thus on-path censors naturally take several shortcuts. For example, prior work has found that when on-path censors believe that a TCP connection has been terminated (e.g., if one of the endpoints sent a valid `RST` packet), then they delete the corresponding TCB and ignore subsequent packets on that connection [9, 11, 23, 36]. Such shortcuts make censors more scalable, but also more susceptible to evasion.

In-path Censors In-path (man-in-the-middle) censors also perform DPI to determine whether to block a connection, but they can do more than just inject a `RST` or lemon response. For example, an in-path censor is able to simply drop a connection's packets altogether. Alternatively (as we will see in Kazakhstan), an in-path censor can also hijack a connection entirely, inject a block-page, and prevent the client's packets from reaching the server. Evading an in-path censor requires tricking the censor into believing that a connection should not be censored, for instance by hiding the true identity of the server [13, 40, 41], obfuscating the protocol [18, 25, 37], or modifying the packets in such a way that the censor no longer recognizes the forbidden query as a target.

Measuring Censors There has been a wide range of work measuring how censors work. This can be broadly broken down into two broad categories:

First are studies into what specific content or destinations censors block [27, 28, 31, 34]. Our work is largely orthogonal to these prior efforts; our primary goal is not to discover *who* or *what* is being censored, but to understand *how* it is being censored (and evade it).

Second is the body of work that studies how censors operate [5, 7, 14, 19, 26, 42, 43]. Our work is complementary to these prior efforts, in that we are able to lend new insights into how several censors perform on-path censorship, as well as gaps in their logic and bugs in their implementations. For instance, we believe we are the first to observe that censors use different transport-layer techniques depending on the overlying application.

2.2 Evasion via Packet Manipulation

There is a long history of evading on-path and in-path censorship through the application of *packet-manipulation strategies*. At a high level, these techniques alter and inject packets at one of the communicating endpoints (typically the client). In so doing, their goal is to either de-synchronize the censor's state (e.g., by injecting TTL-limited RSTs [29]) or to confuse the censor into not recognizing a censored keyword (e.g., by segmenting TCP packets).

Client-side evasion The earliest packet-manipulation strategies to evade on-path censors come from an open-source project from 2011, sniffjoke [3]. sniffjoke introduced a handful of client-side strategies, such as injecting packets with random sequence numbers or injecting packets that shift the sequence number but corrupt the payload. Unfortunately, many of the specific strategies sniffjoke employed have long been defunct, but its broad approaches were later re-discovered by other work [23, 36].

Recently, there have been four key efforts towards evading on-path censors. Khattak et al. [21], lib·erate [23], INTANG [36], and Geneva [9] introduced myriad client-side packet-manipulation strategies. Each of these focused primarily on strategies that manipulated IPv4 and TCP packets, along with a handful of strategies that manipulated HTTP.

Our work is complementary to these prior efforts. Whereas prior published work focused strictly on client-side strategies, we explore server-side. Also, prior work focused almost exclusively on HTTP; we explore strategies for more protocols, and show that many strategies that work for one protocol do not work for another. Our results from investigating multiple protocols lead us to refine prior work's findings. For instance, Wang et al. [36] showed that the GFW was capable of reassembling TCP streams to detect censored keywords in HTTP requests; our result confirms this for HTTP, but show that the GFW is frequently incapable of doing so over FTP, indicating that censors use different transport-layer techniques depending on the application.

Server-side evasion To the best of our knowledge, all prior censorship evasion systems require some degree of client-side evasion software. Even techniques that rely on server-side features, such as domain fronting [16] or decoy routing [41], require client-side changes. However, there are two server-side strategies that are similar in spirit. In 2010, Beardsley and Qian [8] demonstrated that a variant of TCP simultaneous open was able to bypass some intrusion detection systems; these do not appear to work against censors, but we show in §5 that Geneva discovered multiple simultaneous open-based strategies that work against China's GFW. brdgrd [38] intercepted packets sent by a Tor bridge to the Tor client, and employed a relatively simple strategy—it lowered the TCP window size of outbound SYN+ACK packets. This caused Tor clients to segment their TLS handshake packets, splitting the set of supported ciphersuites across multiple TCP packets. At that time, the GFW was unable to reassemble TCP segments, and thus this strategy avoided detection and blocking. In 2013, the GFW added the ability to reassemble TCP segments, rendering brdgrd defunct. Since then, we are aware of no other work on this topic: all prior literature in this space has explored only client-side strategies [9, 21, 23, 36].

Geneva Geneva employs a *genetic algorithm* to automatically discover packet-manipulation strategies to circumvent censorship. Like with all genetic algorithms, Geneva composes basic "genetic building blocks" to form more sophisticated actions. In particular, it composes five packet-manipulation building blocks: (1) `duplicate` (which duplicates a given packet), (2) `fragment` (which fragments a packet at the IP- or TCP-layer), (3) `tamper` (which modifies fields in a packet header), (4) `drop` (which discards a packet), and (5) `send` (which sends the packet). For completeness, we include in the Appendix a short guide to Geneva's syntax; for more details, see [9]. Bock et al. showed that Geneva's simple primitives can be combined to reconstruct virtually all previously discovered circumvention strategies. Geneva trains against real censors by running from within a censoring nation-state, and it uses its genetic algorithm to generate, mutate, and evaluate new strategies. Because it is non-deterministic, Geneva is in essence like a network fuzzer [22, 35], and as a result has discovered both gaps in censors' logic as well as *bugs* in their implementations.

We extend this prior work in two fundamental ways: *First*, like all prior censorship circumvention strategies of which we are aware [13, 21, 23, 36, 40, 41], Geneva was previously only run client-side. In this paper, we apply Geneva to discover purely server-side strategies with *no* extra deployment at the client. *Second*, like with other similar systems, Geneva was only ever evaluated using HTTP, and the authors tacitly assumed that if a circumvention strategy operates only at TCP/IP, then the strategy will work for all TCP-based applications. In this paper, we evaluate over a wide range of applications (DNS-over-TCP, FTP, HTTP, HTTPS, and SMTP) and show that this assumption is false.

First, we answer a natural question: do previously discovered client-side results generalize to server-side?

3 CLIENT-SIDE STRATEGIES DO NOT GENERALIZE

Prior work has identified a wealth of client-side strategies for circumventing censorship. Some of these strategies are tailored specifically to the client; for instance, "Segmentation" strategies split up a client's HTTP GET request across multiple TCP packets, exploiting an apparent bug in some censors' packet reassembly code [9]. However, other client-side strategies appear as if they would work from the server, as well. For example, a seminal circumvention strategy has the client send a TCP RST with a TTL large enough to reach the censor but too small to reach the server [9, 21, 23, 29, 36]. As a result of this strategy, the censor believes the connection has been torn down and thus pays no attention to future packets from that connection, allowing the client to send requests that would have otherwise been censored. *Should such strategies not also work from the server?*

We experimentally evaluated whether client-side strategies can be translated to work from the server-side, as well. Starting with all 36 of the currently working client-side strategies discovered by Bock et al. [9], we manually identified 11 strategies that had no obvious server-side analog (such as Segmentation) and discarded them. All of the remaining 25 strategies involved sending an "insertion packet" (a packet that is processed by the censor but not by the server, like the TTL-limited RST) during or immediately after the 3-way handshake.

The only packet a server typically sends before the censored query is a SYN+ACK. For each strategy, we generate two new server-side analogs: one that sends the insertion packet before the SYN+ACK, and one that sends it after. We then tested these strategies with clients at vantage points within China connecting to a server we control at a vantage point in the US.

Unfortunately, *none* of these strategies worked when run server-side. This is surprising: many of the "TCB Teardown" strategies found by Bock et al. [9] involve the client sending tear-down packets (insertion packets with RST or RST+ACK flags) immediately after receiving the server's SYN+ACK; these server-side analogs also send tear-down packets immediately after the SYN+ACK, the only difference being that they come from the server. We considered the possibility that network delays were causing the server's tear-down packets to arrive at the censor after the client's censored query[1]. To account for this, we instrumented our client to delay sending its query until it received the insertion packets, but this was also unsuccessful at evading censorship.

In other words, for some of these strategies, the *only* difference was whether it was the client or the server that sent the insertion packets, and yet none of them work. We considered that the censor may be treating inbound packets differently than outbound—for instance, it may have been the case that the censor simply ignores inbound RST packets. To test for this, we also ran the server from inside China and the client in the US, but the strategies continued to fail. This indicates that the GFW tries to determine which host is the client (the one who initiated the connection), and processes the client's packets differently than the server's.

Collectively, these results show that client-side strategies *do not generalize* to server-side. Moreover, the results show that clients' and servers' packets are processed differently, and therefore the censors' shortcomings that previous work exploited client-side do not necessarily lend insight into how to circumvent from server-side. In short: server-side censorship circumvention requires a blank-slate approach.

4 SERVER-SIDE METHODOLOGY

4.1 Geneva **Extensions**

New Protocols Bock et al. [9] previously applied Geneva only to HTTP. We have extended Geneva to be able to train over a variety of applications across a variety of protocols. Specifically, we added support for DNS-over-TCP, FTP, HTTPS, and SMTP.

Non-additions We also explored applying server-side evasion to Tor Bridges and Telegram MTProxy servers [32, 33]. Although Tor and Telegram are both blocked at the IP and DNS level, we are unable to trigger active probing to private unpublished Tor bridges or MTProxies. The Tor team is aware that Tor does not currently trigger active probing, and these findings are consistent with recent reports [9, 36]. We focus our efforts on the protocols that are getting censored today, and we leave a deeper exploration of server-side training over other anti-censorship protocols to later work.

Server-side Evasion Geneva is largely agnostic to packet semantics; it is able to recompute checksums, but it is not configured to understand the meanings behind any particular packet header fields. As a result, converting Geneva from client-side to server-side was relatively straightforward, requiring only minor changes to its implementation.

Country	Vantage Points	Protocols
China	Beijing, Shanghai Shenzen, Zhengzhou	DNS, FTP, HTTP, HTTPS, SMTP
India	Bangalore	HTTP
Iran	Tehran, Zanjan	HTTP, HTTPS
Kazakhstan	Qaraghandy, Almaty	HTTP

Table 1: Client locations and protocols used in our experiments.

We configured Geneva to initialize each population pool with 300 individuals, and allowed evolution to take place for 50 generations, or until population convergence occurs. Although Geneva is capable of evolving not only how it manipulates packets but also *which* packets it triggers on, we observed that for DNS-over-TCP, HTTP, HTTPS, and SMTP, the only packet the server could trigger on before a censorship event was the SYN+ACK packet. Thus, as a slight optimization, for these protocols, we restricted Geneva to only be able to trigger on SYN+ACKs.

4.2 Data Collection Methodology

Over the span of five months, we ran Geneva server-side in six countries—Australia, Germany, Ireland, Japan, South Korea, and the US—on five protocols: DNS (over TCP), FTP, HTTP, HTTPS, and SMTP (all over IPv4). We used unmodified clients within four nation-state censors—China, India, Iran, and Kazakhstan—to connect to our servers. For each nation-state censor, we trained on each protocol for which we were able to trigger censorship; all four countries censored HTTP, but only China censored all six protocols.[2] Table 1 shows the client locations and protocols we used throughout our experiments. Within each censored regime, we find no significant difference in strategy effectiveness across the different vantage points or external servers.

Each country and protocol required a slightly different configuration to trigger censorship:

- *DNS-over-TCP (China)*: We make a censored request with an unmodified DNS client to open resolvers (Google and Cloudflare), as well as resolvers we control outside China.
- *FTP (China)*: We sign into FTP servers we control and issue requests for files with sensitive keywords as names (e.g., `ultrasurf`).
- *HTTP (all countries)*: In China, we issue GET requests with a censored keyword in the URL parameters (for instance, `?q=ultrasurf`). In India, Iran, and Kazakhstan, we issue GET requests with a blacklisted website in the `Host:` header.
- *HTTPS (China and Iran)*: We perform a TLS handshake with a forbidden URL (e.g., `youtube.com` in Iran and `www.wikipedia.org` in China) in the Server Name Indication (SNI) field.
- *SMTP (China)*: We connect to SMTP servers we control and, from our unmodified clients, send an email to a forbidden email address, `xiazai@upup.info` [17].

[1] This is not an issue when clients send both the tear-down and the query, because we can generally expect packets to arrive FIFO.

[2] Contrary to the findings by Aryan et al. [7], we find that Iran no longer censors DNS-over-TCP at all.

In all of the above settings, we configure Geneva to consider censorship to have been avoided if the connection is not forcibly torn down and if the client receives the correct, unaltered data.

Residual Censorship In China, we observe that different protocols are handled differently by the GFW. For example, over HTTP, the GFW has *residual censorship*: for approximately 90 seconds after a forbidden request is censored, all TCP requests to the server IP and port elicit tear-down packets from the GFW immediately following the three-way handshake. Prior work has documented the existence of residual censorship in some cases for HTTPS; however, we do not observe this behavior from any of our vantage points during our experiments and confirm that as of time of writing, HTTPS residual censorship is not active in China. Further, we do not observe this behavior from any of our vantage points in China for SMTP, DNS-over-TCP, or FTP; after the forbidden request on these protocols is censored, the user is free to make a second follow-up request immediately.

Evasion Success Rates It has been shown that, somewhat surprisingly, some packet-manipulation strategies succeed only *some* of the time; for instance, Bock et al. [9] found some client-side strategies that work roughly 50% of the time. Throughout the paper, we present the success rates of the various strategies Geneva has found. For DNS in particular, this requires some special consideration, because, according to RFC 7766 [2] on DNS-over-TCP: *DNS clients SHOULD retry unanswered queries if the connection closes before receiving all outstanding responses. No specific retry algorithm is specified in this document.* Censorship by the GFW qualifies as a premature connection close, and thus results in retries, but the RFC leaves the exact number of retries up to the implementer. This serves to greatly improve the success rates of any server-side strategies for DNS-over-TCP: even if the strategy works only 50% of the time, with just 2 retries (3 total queries), the success rates will improve to 87.5%.

We have found that, in practice, applications choose different numbers of DNS retries. Some `dig` versions make only 1 retry, others retry repeatedly (sometimes 3–5 times), and others allow the user to specify how many. Python's DNS library tries 3 times over TCP when faced with the GFW's TCP `RST`s. Google Chrome on Windows retries 4 times after a censorship event (for a total of 5 requests per page load). Chrome also periodically retries failed page loads (often over 20 times, we have observed). To be consistent with most DNS clients, we test all of our strategies with a maximum of 3 tries.

Follow-up Experiments At the end of each run, Geneva outputs the packet-manipulation strategies that succeeded (and failed). We then perform follow-up experiments to understand *why* the strategies work (or fail) and to glean information about how these various censors operate. We describe the specific steps we take in-line with our results.

5 SERVER-SIDE RESULTS

Here, we detail newly discovered strategies that defeat censors from the server-side. Table 2 summarizes our results across all countries (China, India, Iran, and Kazakhstan) and applications (DNS-over-TCP, FTP, HTTP, HTTPS, and SMTP).

Strategy		**Success Rates**				
#	**Description**	**DNS**	**FTP**	**HTTP**	**HTTPS**	**SMTP**
China						
–	No evasion	2%	3%	3%	3%	26%
1	Sim. Open, Injected `RST`	89%	52%	54%	14%	70%
2	Sim. Open, Injected Load	83%	36%	54%	55%	59%
3	Corrupt `ACK`, Sim. Open	26%	65%	4%	4%	23%
4	Corrupt `ACK` Alone	7%	33%	5%	5%	22%
5	Corrupt `ACK`, Injected Load	15%	97%	4%	3%	25%
6	Injected Load, Induced `RST`	82%	55%	52%	54%	55%
7	Injected `RST`, Induced `RST`	83%	85%	54%	4%	66%
8	TCP Window Reduction	3%	47%	2%	3%	100%
India						
–	No evasion	100%	100%	2%	100%	100%
8	TCP Window Reduction	–	–	100%	–	–
Iran						
–	No evasion	100%	100%	0%	0%	100%
8	TCP Window Reduction	–	–	100%	100%	–
Kazakhstan						
–	No evasion	100%	100%	0%	100%	100%
8	TCP Window Reduction	–	–	100%	–	–
9	Triple Load	–	–	100%	–	–
10	Double `GET`	–	–	100%	–	–
11	Null Flags	–	–	100%	–	–

Table 2: Summary of server-side-only strategies and their success rates. All of these strategies manipulate only TCP, and yet, against China's GFW, their success rates are application-dependent. Kazakhstan's HTTPS and Iran's DNS-over-TCP censorship infrastructure are currently inactive.

5.1 Server-side Evasion in China

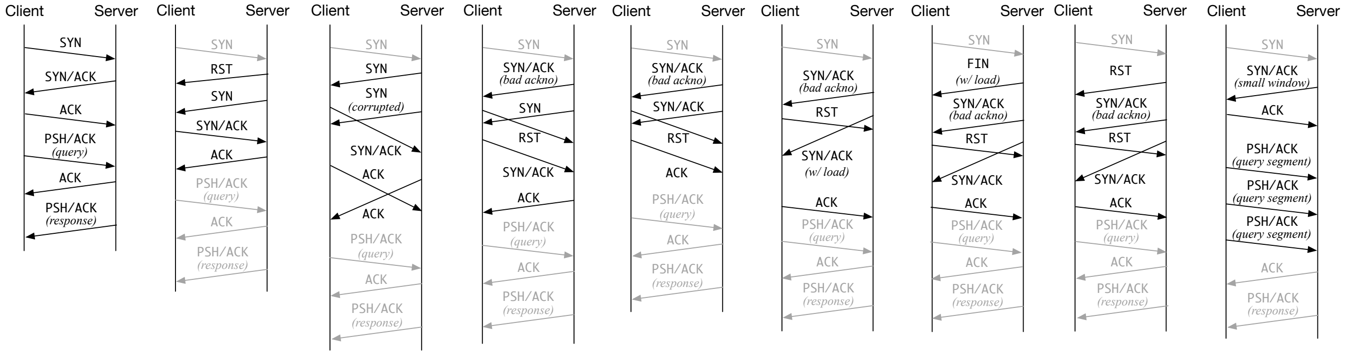
We applied Geneva from the server side against the GFW across DNS, FTP, SMTP, HTTP, and HTTPS. Geneva identified 8 distinct server-side only strategies that are successful at least 50% of the time for at least one protocol in China: 4 for DNS, 5 for FTP, 1 for SMTP, 4 for HTTP, and 2 for HTTPS. We provide packet waterfall diagrams in Figure 1 which show the resulting server- and client-behaviors when the strategies are run. Although the strategies require *no client-side modifications whatsoever*, they induce client-side behavior that assists in circumventing censorship. In the rest of this subsection, we explore each of these strategies, explain why they work, and describe what they teach us about China's GFW.

Strategy 1: Simultaneous Open, Injected `RST` (China)
DNS (89%), FTP (52%), HTTP (54%), HTTPS (14%), SMTP (70%)

```
[TCP:flags:SA]-
    duplicate(
        tamper{TCP:flags:replace:R},
        tamper{TCP:flags:replace:S})-| \/
```

Simultaneous Open Strategy 1 triggers on outbound `SYN+ACK` packets. Instead of sending the `SYN+ACK`, it replaces it with two packets—a `RST` and a `SYN`—and sends them instead. How does an unmodified client respond to this strange sequence of packets?

First, the `RST` packet is actually ignored by the client, because it does not have the `ACK` flag set and the TCP connection is not yet



Normal behavior

Strategy 1
Simultaneous open, injected RST

Strategy 2
Simultaneous open, injected load

Strategy 3
Corrupted ACK, simultaneous open

Strategy 4
Corrupted ACK, alone

Strategy 5
Corrupted ACK, injected load

Strategy 6
Corrupted load, induced RST

Strategy 7
Injected RST, induced RST

Strategy 8
TCP window reduction

Figure 1: Server-side evasion strategies in China. All of the strategies work without modifications to the client, and yet they induce client-side behavior that helps circumvent censorship. (Standard packets at the beginning and the end are grayed out to emphasize the critical differences from normal behavior.)

in a synchronized state. Despite RFC 793 [1] suggesting that the connection be torn down, we find that in practice, TCP implementations across all modern operating systems ignore this `RST`. Second, the injected `SYN` packet serves to initiate *TCP simultaneous open.*

RFC 793 [1] requires TCP implementations to support simultaneous open. Originally, simultaneous open was meant to occur when two hosts attempt to open a connection by sending `SYN` packets to each other at the same time. However, a server can simulate simultaneous open by responding to a `SYN` packet from the client with a `SYN` packet of its own. To the client, this resembles simultaneous open, since the client receives a `SYN` packet, and therefore must respond with a `SYN+ACK` packet. This strategy employs simultaneous open by first sending an inert `RST` packet, then by setting up the connection with a `SYN` packet.

When used for HTTP, Strategy 1 has a success rate of 54%. We see similar success rates for FTP and for each single DNS-over-TCP query (recall that DNS will try up to 3 times).

It is tempting to assume that this strategy works because the injected `RST` tears down the connection, and the `SYN` packet looks like an entirely new connection in the reverse direction (thereby making the censored request sent by the client ignored). However, this is not the case—as demonstrated above, injected `RST` packets either inside or outside the 3-way handshake from the server are unable to tear down a connection. Another potential theory is that the GFW simply cannot properly handle TCP simultaneous open; this too, however, is incorrect: if the `RST` is removed from the strategy, the strategy fails. Instead, we hypothesize that this strategy is far more nuanced, and is actually performing a desynchronization attack by exploiting a bug in the GFW's resynchronization state.

Prior work has hypothesized that the presence of a `RST` packet during the three-way handshake can put the GFW in a resynchronization state with about 50% probability [9, 36]. Therefore, we expect the injected `RST` packet not to tear down the connection, but instead to put the GFW into the resynchronization state. Wang et al. hypothesized that the only packets sent by the server that the GFW resynchronizes on are `SYN+ACK` packets, so the next packet for the GFW to resynchronize on is the `SYN+ACK` packet *sent by the client.* At this point, the GFW should just properly resynchronize onto our connection—but it does not. Why?

When TCP simultaneous open is performed, the sequence number does not advance during the handshake in the same fashion as it does in a regular TCP three-way handshake. During TCP simultaneous open, the `SYN+ACK` packet sent by the client retains the same sequence number as the original `SYN` packet, and 1 is not added to the sequence number until the `ACK` packet is sent. Therefore, if the GFW's resynchronization state is not aware that simultaneous open is being performed, it will synchronize onto this `SYN+ACK` packet and assume that the sequence number has already been incremented by 1, as it would be if this were an `ACK` packet finishing the regular 3-way handshake. As such, the GFW will fail to advance its sequence number by 1 when the request is sent by the client, making the GFW desynchronized by 1 byte from the real connection.

To test this theory, we instrumented a client-side request to decrement the sequence number of the forbidden request by 1 while the strategy is run on the server side. If the theory holds, we expect to experience censorship approximately 50% of the time (as this is how frequently China's censors enter the resynchronization state [36]). Indeed, when we perform this experiment, that is exactly the result we see. Note that if we perform this sequence number adjustment experiment without running the server-side strategy, we never experience censorship as expected, because the real query is now desynchronized from the connection.

This experiment suggests that Strategy 1 actually performs a desynchronization attack against the GFW, and that a bug exists in the GFW's resynchronization state handling of simultaneous open. As we will see, this bug is quite powerful, and Geneva identifies it repeatedly in our experiments.

Strangely, Strategy 1 does not work well against HTTPS. We hypothesize this is because the `RST` does not cause the GFW to enter the resynchronization state for HTTPS, but does for the other protocols. The rest of this section explores a number of cases in which TCP/IP-level attacks work well for one application-level protocol but not another; §6 offers an explanation why this occurs.

Strategy 2: Simultaneous Open, Injected Load (China)
DNS (83%), FTP (36%), HTTP (54%), HTTPS (55%), SMTP (59%)

```
[TCP:flags:SA]-
    tamper{TCP:flags:replace:S}(
        duplicate(,
            tamper{TCP:load:corrupt}),)-| \/
```

Strategy 2 also relies on simultaneous open, but with a slightly different mechanism. Rather than injecting a RST, it changes the outgoing SYN+ACK packet into two SYN packets: the first SYN is well-formed and the second has a random payload. It has comparable success to Strategy 1, though slightly worse for FTP (36% vs. 52%) and SMTP (59% vs. 70%), and better for HTTPS (55% vs. 14%).

Like with the first strategy, when the first SYN packet reaches the client, it triggers simultaneous open, prompting the client to respond with a SYN+ACK. Since both SYN packets are sent simultaneously, both likely cross the GFW before the client responds. The second SYN packet with a payload will induce the GFW to enter the resynchronization state, and like last time, the next packet available for it to resynchronize on is the SYN+ACK packet *from the client*, again desynchronizing the GFW by 1 from the connection. We confirmed this by repeating the prior experiment on this strategy.

Strategy 2 does not damage the TCP connection despite the client being unmodified. Although it is uncommon for SYN packets to carry a payload, this is permitted by the RFC (this behavior is required by TCP Fast Open), and the payload is ignored by the client (though the client does respond with an ACK to acknowledge the current sequence number).

Strategy 3: Corrupted ACK, Simultaneous Open (China)
DNS (26%), FTP (65%), HTTP (4%), HTTPS (4%), SMTP (23%)

```
[TCP:flags:SA]-
    duplicate(
        tamper{TCP:ack:corrupt},
        tamper{TCP:flags:replace:S})-| \/
```

Geneva identified one final strategy relying on simultaneous open. Strategy 3 copies the SYN+ACK packet: it corrupts the ack number of the first, and converts the second to a SYN. The SYN+ACK with the corrupted ack number induces the client to send a RST packet, before responding with a SYN+ACK to initiate the TCP simultaneous open. However, unlike Strategies 1 and 2, this strategy is the most successful for FTP.

Wang et al. [36], while studying HTTP censorship, hypothesized that a SYN+ACK from the server with an incorrect ack number is sufficient to trigger the GFW's resynchronization state. We observe that this is no longer true for; however, it *does* work for FTP censorship. Therefore, when the SYN+ACK with the corrupted ack number is sent, the FTP portion of the GFW enters the resynchronization state and resynchronizes on the next packet from the client—the RST induced by the incorrect ack number. Because the RST packet has the incorrect sequence number, the GFW will become desynchronized from the connection. Geneva also identified successful variants of this species in which the order of the two packets is reversed.

Strategy 4: Corrupt ACK Alone (China)
DNS (7%), FTP (33%), HTTP (5%), HTTPS (5%), SMTP (22%)

```
[TCP:flags:SA]-
    duplicate(
       tamper{TCP:ack:corrupt},)-| \/
```

Strategy 4 is identical to Strategy 3, but without simultaneous open. This shows that, although simultaneous open is not required to evade FTP censorship, it improves the success rate (33% vs. 65%).

Strategy 5: Corrupt ACK, Injected Load (China)
DNS (15%), FTP (97%), HTTP (4%), HTTPS (3%), SMTP (25%)

```
[TCP:flags:SA]-
    duplicate(
       tamper{TCP:ack:corrupt},
       tamper{TCP:load:corrupt})-| \/
```

Strategy 5 offers an even greater improvement in success rate. This strategy sends a SYN+ACK with a corrupted ack number, followed by another SYN+ACK with a random payload. As with the previous strategies, the corrupted ack number induces the client to send a RST packet, which the GFW resynchronizes on. This RST is critical to the strategy's success: if we instrument the client to drop this induced RST, the strategy stops being effective.

Strategy 5 is highly successful (97%), but again, largely only applicable to FTP. We do not yet understand the reason for the improvement in success rate with the inclusion of simultaneous open or an inert payload.

We draw special attention here to the specific order that the injected packets are sent (first, corrupted ack, followed by injected payload). When we reverse the order of the packets, the strategy is ineffective. However, Geneva discovered a successful species almost identical to this experimental ineffective strategy, requiring only one modification:

Strategy 6: Injected Load, Induced RST (China)
DNS (82%), FTP (55%), HTTP (52%), HTTPS (54%), SMTP (55%)

```
[TCP:flags:SA]-
    duplicate(
        duplicate(
            tamper{TCP:flags:replace:F}(
                tamper{TCP:load:corrupt},),
            tamper{TCP:ack:corrupt}),)-| \/
```

Resynchronization State, Revisited Strategy 6 replaces the outbound SYN+ACK with three packets: (1) A FIN with a random payload, (2) A SYN+ACK with a corrupted ack number, and (3) The original SYN+ACK. Note the apparent similarity with Strategy 5: an inert payload and SYN+ACK with corrupted ack are both sent to the client, but Geneva found that adding the FIN makes the strategy more effective for all but FTP. We also found that this strategy works equally well if an ACK flag is sent instead of FIN.

When the FIN (or ACK) packet with the payload arrives at the client, it is ignored, and like with previous strategies, when the corrupted SYN+ACK packet arrives, it induces a RST. However, unlike the previous strategies, this RST packet is not a critical component of the strategy, but rather a vestigial side-effect of it—if we instrument the client to drop the RST, the strategy is still equally effective. This is because the GFW is resynchronizing not on the RST, but instead on the SYN+ACK packet with an incorrect ack number.

This presents a stark difference from Strategy 5—once the corrupted ack number caused the GFW to enter the resynchronization state over FTP, the GFW did not resynchronize on the next packet in the connection (which would be a SYN+ACK with the correct sequence and ack numbers), but rather on the next packet from the client (the RST with an incorrect sequence number). This has a surprising implication: depending on the *reason* the GFW enters the resynchronization state, it *behaves differently*.

In summary, our hypothesis for the new behavior of the resynchronization state is as follows:

(1) A payload from the server on a non-SYN+ACK packet causes the GFW to resynchronize on the next SYN+ACK packet from the server or the next packet from the client with the ACK flag set for every protocol.
(2) A RST from the server causes the GFW to resynchronize on the next packet it sees from the client for each protocol except HTTPS.
(3) A SYN+ACK with a corrupted ack number only causes a resync for FTP, and it resynchronizes on the next packet from the client.

We test this theory with Strategy 7, which begins by copying the SYN+ACK packet twice. To the first duplicate, the flags are changed to RST, to the second duplicate, the ack number is corrupted, and the third is left unchanged. All three packets are then sent. The first RST packet is ignored by the client, the corrupted ACK induces the client to send a RST, and finally the client responds to the server's SYN+ACK with an ACK to properly finish the handshake.

Strategy 7: Injected RST, Induced RST (China)
DNS (83%), FTP (85%), HTTP (54%), HTTPS (4%), SMTP (66%)

```
[TCP:flags:SA]-
    duplicate(
        duplicate(
            tamper{TCP:flags:replace:R},
            tamper{TCP:ack:corrupt}),)-|
```

If our above new model for the resynchronization state holds true, we expect the first RST packet of Strategy 7 to put the GFW in the resynchronization state for every protocol but HTTPS, and resynchronize *not* on the next packet it sees in the connection or the next SYN+ACK, but on the next packet it sees from the client, which is the induced RST with an incorrect sequence number.

To test this, we instrumented a client to adjust its sequence numbers to match that in the RST packet. This resulted in censorship, indicating that the GFW indeed synchronized on this packet, and confirming our new model of GFW's resynchronization state.

Strategy 8: TCP Window Reduction (China)
DNS (3%), FTP (47%), HTTP (2%), HTTPS (3%), SMTP (100%)

```
[TCP:flags:SA]-
 tamper{TCP:window:replace:10}(
  tamper{TCP:options-wscale:replace:},)-|\/
```

TCP Window Reduction Strategy 8 works by reducing the TCP window size and removing wscale options from the SYN+ACK packet, inducing the client to segment the forbidden request. This strategy is almost the exact same strategy identified by brdgrd [38] in 2012. The fact that this strategy works at all is highly surprising—the GFW has had the capacity to reassemble segments since brdgrd became defunct in 2012. It appears that the portion of the GFW responsible for FTP censorship is incapable of reassembling TCP segments. This strategy is also the most effective at evading SMTP censorship in China, and as we show next, it is highly effective in other countries, as well.

5.2 Server-side Evasion in India & Iran

Our vantage points in India are all within the Airtel ISP, and we confirm that Airtel only censors over HTTP [43]. Our vantage points in Iran are in Zanjan and Tehran; here, HTTP, HTTPS, and DNS is censored (though DNS-over-TCP is uncensored, so we will focus on HTTP and HTTPS here).

Airtel's censorship injects an HTTP 200 with a block page with a FIN+PSH+ACK packet instead of tearing down the connection. Iran's censorship simply "blackholes" the traffic, dropping the offending packet and all future packets from the client in the flow for 1 minute. In India, as reported by Yadav et al., we also observe a follow-up RST packet from the middlebox for good measure [43].

We find that both countries only censor on each protocol's default ports (80, 443); hosting a web server on any other port defeats censorship completely. Both countries' middleboxes also do not seem to track connection state at all: sending a forbidden request without performing a three-way handshake to the server elicits a censorship response.

Given the lack of state tracking for these middleboxes, the problem of server-side evasion becomes even more challenging: there is no censor state to invalidate or teardown, so the only feasible strategies are those that mutate the client's forbidden request in a manner that cannot be processed by the censor. When deployed from the server side, Geneva identifies one such strategy in both countries that we have already seen: TCP Window Reduction (Strategy 8).

Again, simply by reducing the TCP window size of the SYN+ACK packet, it induces the client to segment the forbidden request. This works because the middleboxes in both countries appear incapable of reassembling TCP segments, so once the forbidden request is segmented, it is uncensored.

This result, combined with the similar success of this strategy in China against FTP and SMTP, suggests a pattern of generalizability for client-side strategies. Client-side strategy species that work by performing simple segmentation can be re-deployed at the server-side in the form of a strategy that *induces* simple segmentation.

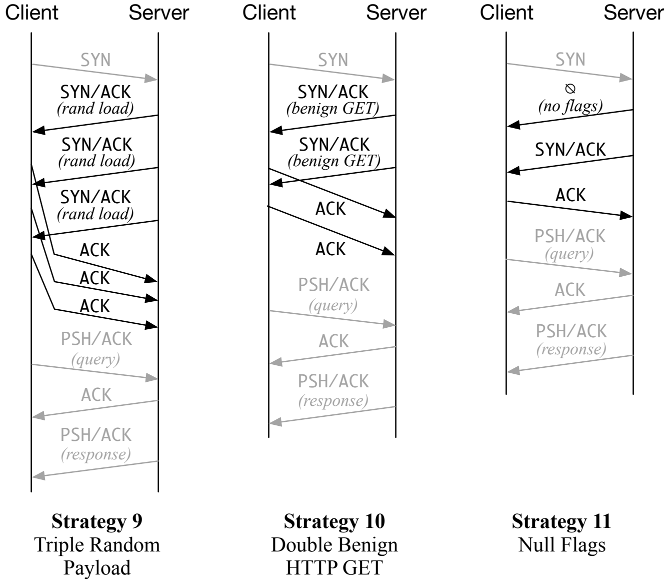


Figure 2: Server-side evasion strategies that are successful against HTTP in Kazakhstan.

5.3 Server-side Evasion in Kazakhstan

Kazakhstan has deployed multiple types of censorship. Previous works have explored weaknesses in their now-defunct HTTPS man-in-the-middle [9]. Here, we focus on their in-network DPI censorship of HTTP. Like the Airtel ISP, the censor steps in when a forbidden URL is specified in the `Host:` header of an HTTP GET request. When the censor activates, it first performs a man-in-the-middle, so all packets in the TCP stream (including the forbidden request) for approximately 15 seconds are intercepted by the censor and will not reach the server. The censor then injects a `FIN+PSH+ACK` packet with a block page to inform the user the page is blocked and the connection terminates.

We provide an overview of our successful server-side evasion strategies against Kazakhstan in Figure 2.

Strategy 9: Triple Load (Kazakhstan) *HTTP (100%)*

```
[TCP:flags:SA]-
    tamper{TCP:load:corrupt}(
        duplicate(
            duplicate,),)-| \/
```

Strategy 9 takes the outbound `SYN+ACK` packet, adds a random payload, and then duplicates it twice, effectively sending three back-to-back `SYN+ACK` packets with payloads. The payloads and duplicate packets are ignored by the client, and the client completes the 3-way handshake. This strategy works 100% of the time in Kazakhstan.

Strangely, we find that Strategy 9 is effective only if the packet with the load is sent at least three times. Increasing the number of duplicates does not reduce the effectiveness of the strategy, but removing any of them renders the strategy unsuccessful.

We find the size of the payload injected by the server does not affect the success of the strategy; whether just 1 byte is injected or hundreds, the strategy is equally effective. This suggests that it is the presence of the payloads, not the length of the payloads, that causes the censor to fail.

We also find that it is critical that each of the `SYN+ACK` packets have the payload. If we instrument the strategy instead to send just one `SYN+ACK` with a payload (either first, in the middle, or last), the strategy fails, or if we instrument the strategy to send two `SYN+ACK` with a payload (back-to-back in the beginning, back-to-back at the end, and with an empty `SYN+ACK` in between), the strategy fails. The strategy *only* works if three back-to-back packets with a payload are sent during the handshake.

We first test if this strategy is causing a desynchronization in the censor. If the censor advances its TCB upon seeing the `SYN+ACK` payload, we do not know if the censor will advance it for all of the packets, or just some subset of them. To test each of these cases, we instrumented the client to increment the sequence number of its forbidden request by single, double, and triple the length of the injected payload. However, none of these instrumented requests trigger censorship, suggesting that this attack does *not* perform a desynchronization attack against the censor.

Instead, we hypothesize the censor monitors connections specifically for patterns that resemble normal HTTP connections, and seeing payloads from the server during the handshake violates this model, causing it to ignore the connection. However, we do not understand why three payloads are required to enter this state. The next strategies identified by Geneva support this hypothesis.

Strategy 10: Double GET (Kazakhstan) *HTTP (100%)*

```
[TCP:flags:SA]-
    tamper{TCP:load:replace:GET / HTTP1.}(
        duplicate,)-| \/
```

Strategy 10 duplicates the outbound `SYN+ACK` packet and sets the load to the first few bytes of a *well-formed, benign* HTTP GET request. Since this payload is on the `SYN+ACK`, the client ignores it, and the TCP connection is unharmed, but the payload is processed by the censor. The above strategy shows the minimum portion of a HTTP GET request required for the strategy to work (if the "`.`" is removed, the strategy stops working). As long as the GET request is well-formed up to the "`.`", the strategy works; for example, the strategy works equally well if we specify the rest of the GET request or use a different or longer path. We also find that the duplicate is required for this strategy to work; if the GET is only sent once, the strategy does not work.

Frankly, we do not understand why this strategy works. We hypothesize the request is just enough to pass a regular expression or pattern matching inside the censor, and seeing the well-formed GET request is sufficient for the censor to think the server is actually the client. To confirm the censor is processing injected packets, we try probing the censor by injecting forbidden GET requests. We find two ways to inject the content such that it elicits a response from the censor: injecting two GET requests *during* the handshake, or performing simultaneous open and injecting one GET request *after* during the handshake.

We do not understand why two requests are required to elicit a response during the handshake; we hypothesize the first request is needed to break out of the censor's "handshake" state and the second request is then processed. To test this hypothesis, we try injecting a forbidden request followed by a benign request, and no censorship occurs. This indicates that when content is injected before a connection is established, it is the second request that the censor processes.

Strategy 11: Null Flags (Kazakhstan) *HTTP (100%)*

```
[TCP:flags:SA]-
    duplicate(
       tamper{TCP:flags:replace:},)-| \/
```

Strategy 11 duplicates outbound SYN+ACK packet. To the first duplicate, *all* of the TCP flags are cleared before it is sent, and the second duplicate is sent unchanged. We find this strategy works 100% of the time. Although Geneva first discovered this strategy by clearing the TCP flags, it also identified the strategy works as long as FIN, RST, SYN, and ACK are not used. We hypothesize the censor is monitoring for "normal" TCP handshake patterns, and when those patterns are violated, the connection is ignored.

Finally, as expected, Strategy 8 also works in Kazakhstan: inducing client segmentation is sufficient to defeat the censor.

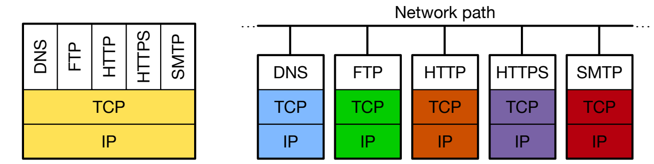
6 MULTIPLE CENSORSHIP BOXES

The server-side evasion strategies from §5 exhibit a surprising property: although they strictly operate at the level of TCP (specifically the 3-way handshake), they have varying success rates depending on the higher-layer application within a given country. This defies expectation: our evasion strategies exploit gaps in censors' logic or implementation at the transport layer, and thus those same gaps *ought* to be exploitable by *all* higher-layer applications. Exceptions to this indicate either a cross-layer violation or a different network stack implementation for each application—two phenomena that are necessarily rare in the layered design of the Internet.

The remaining explanation is that China uses distinct boxes—with distinct network stack implementations—for each of the application protocols they censor. We depict this in Figure 3.

This raises an important question: how does the censor know which box to apply? This is not as simple as triggering on port numbers; recall that, in our experiments, we randomize the server's port numbers, and yet still experience censorship for each protocol. Indeed, most of the GFW's censorship is *not port-specific*.

We posit that *each* of the GFW's separate censorship boxes individually track *all* TCP connections until it identifies network traffic that matches its target protocol (i.e., until the request). Note, however, that most of our strategies complete before the end of the 3-way handshake—before it can be determined which application is using it. Thus, if our theory is correct, then when an application-specific TCP-level strategy is used, *all* of the protocols' processing engines react, but only some of them respond incorrectly.



(a) Single censorship box

(b) Multiple censorship boxes

Figure 3: Single versus multiple censorship boxes. A standard assumption is that evasion strategies that work for one application will work for another within a given country. However, our results indicate that China's GFW uses distinct censorship boxes for each protocol, each with their own network stacks (and bugs).

Separate censoring boxes would also explain why the GFW never "fails closed"; i.e., it does not default to censorship if it observes packets that are not associated with a TCB or that it cannot parse. Our multi-box theory suggests that the GFW can *never* fail closed because, although one box may not recognize a packet, it must assume that another box might. If each censorship box were fail-closed, the GFW would destroy every connection.

To see if we can detect the presence of multiple boxes, we sought to locate them via TTL-limited censored probes [43]. We instrumented a client to perform 3-way handshakes with servers of various protocols, and then send the query repeatedly with incrementing TTLs until it elicits a response from a censor. We found that, in China, censorship occurred at the same number of hops for each protocol at each vantage point. This indicates that, if there are indeed multiple boxes, then China collocates them.

7 CLIENT COMPATIBILITY

The evasion strategies presented in §5 take advantage of esoteric features of TCP that appear to have faulty implementations in nation-state censors' firewalls. Server-side deployment risks making the server unreachable to any client that also has the same shortcomings. Conversely, strategies that work for a diverse set of clients are readily deployable. Here, we comprehensively evaluate of all of the strategies against a diversity of client operating systems, and we provide some anecdotal evidence across different link types.

Experiment Setup We formed a private network consisting of an Ubuntu 18.04.3 server running each of the server-side TCP strategies (using Apache2.4 for HTTP and HTTPS). For our clients, we used 17 different versions of 6 popular operating systems: **Windows** (XP SP3, 7 Ultimate SP1, 8.1 Pro, 10 Enterprise (17134), Server 2003 Datacenter, Server 2008 Datacenter, Server 2013 Standard, Server 2018 Standard), **MacOS** (10.15), **iOS** (13.3), **Android** (10), **Ubuntu** (12.04.5, 14.04.3, 16.04.4, 18.04.1), and **CentOS** (6, 7). We tried each protocol and each server-side strategy against each client.

OS Results We found that *all but three strategies* worked on *every* version of every client OS. The only exceptions were Strategies 5, 9, and 10, each of which failed to work on any of the versions of Windows and MacOS. These three strategies all involve sending a SYN+ACK with a payload; Linux's TCP stack ignores these, but Windows' and MacOS's do not.

However, we can slightly alter Strategies 5, 9, and 10 to make them work with all clients. The key insight is that these strategies work on Linux precisely because Linux ignores the payload (but censors do not). However, we can modify the strategy in other ways to make the client ignore the packet while the censor still accepts it; this is commonly referred to as an "insertion" packet, and there are other ways to create insertion packets [9]. For instance, we can send the payload packets with a corrupted chksum (so they are processed by the censor but not the client), and send the original SYN+ACK packet unmodified afterwards. We re-evaluated these three strategies with this modification, and found that with this small change, the strategies worked for *all* client operating systems. An area of future work is evolving strategies directly against many operating systems to avoid requiring these post-hoc modifications.

Results Can Vary by Network We close this section with an *anecdotal* observation. In addition to the tests on our private network, we also tested all strategies from a Pixel 3 running Android 10 on wifi and two cellular networks: T-Mobile, and AT&T in a non-censoring country (anonymized for submission). All strategies worked over wifi, and all worked on the two cellular networks *except* Strategies 1 and 3 for T-Mobile and Strategies 1, 2, and 3 (all of the simultaneous open strategies) for AT&T. We speculate that the failures were caused by other in-network middleboxes. This indicates that, while the *client* may not be an issue with some server-side strategies, the client's *network* might.

These results collectively demonstrate that, when deploying server-side strategies, it is important to test across a wide range of clients and network middleboxes. Fortunately, many of the strategies we have found appear to work across a very wide range of networks and client types, but for practical deployments, a global study of network compatibility would be an important and interesting avenue of future work.

8 DEPLOYMENT CONSIDERATIONS

Where to Deploy? Though we refer to them as "server-side," the strategies we have presented could be deployed at any point in the path between the censor and the server. For instance, a reverse proxy (such as a CDN), a common hosting platform (like Amazon AWS), or even a middlebox along the path (like in TapDance [40]) could run our strategies by manipulating packets in-flight. However, for ease of deployment, we anticipate that our strategies will mainly be run at whichever host is performing the 3-way handshake with the client. Our strategies incur little computation or communication overhead (at most three extra payloads), so we expect that they could be deployed even in performance-critical settings.

Which Strategies to Use? As our results have shown, strategies that work in one country or ISP do not necessarily work in another. Thus, in deployment, the server must determine which strategy to use on a per-client basis. This may prove challenging, as the server must make its determination based only on the client's SYN packet. Coarse-grained, country-level IP geolocation may suffice for nation-states that exhibit mostly consistent censorship behavior throughout their borders (like China). However, for countries with region-specific behavior (such as Iran or Russia), finer-grained determination of ISP may be required. Rapid, accurate determination of which strategies to use is an important area of future work.

9 ETHICAL CONSIDERATIONS

Ethical Experiments We designed our experiments to have minimal impact on other hosts and users. All of our testing and training was done from machines directly under our control. Geneva generates relatively little traffic while training [9] and does not spoof IP addresses or ports. We follow the precedent of evaluating strategies strictly serially, which rate-limits how quickly it creates connections and sends data. We believe this mitigates any potential impact it may have had on other hosts on the same network.

Ethical Considerations of Server-side Evasion In traditional, client-side tools for censorship evasion, the user is directly responsible for attempting to evade the censor, and is taking a deliberate action to do so. As such, the user has the opportunity to both *decide* and *consent* to the evasion, and (ideally) is knowledgeable of the risk associated with attempting to (and/or failing to) evade censorship.

However, such an opportunity may not always be present when server-side strategies are applied to traditional, non-evasive protocols (like DNS, FTP, HTTP, and SMTP). Every server-side strategy discussed in this work runs during the 3-way handshake, so the user has no in-band opportunity to be informed or consent to the server applying strategies over their connection. This raises an ethical question: Should servers have to seek informed consent from users before evading censorship on their behalf?

There are several precedents that lead us to believe that such consent is not necessary. Various evasion techniques are regularly deployed without explicit support from users, such as wider deployments of HSTS, HTTPS, or encrypted SNI, and new techniques such as DNS-over-TLS and DNS-over-HTTPS.

Whatever the answer to this question, we did not face any of these concerns during our experimentation: our servers were not public-facing, served no sensitive content, and were not connected to by anyone besides our own clients.

10 CONCLUSION

We have presented eleven server-side packet-manipulation strategies for evading nation-state censors—ten of which are novel and, to our knowledge, the *only* working server-side strategies today. Our results lend greater insight into how the national censors in China, India, Iran, and Kazakhstan operate: we find, for instance, that the GFW appears to use separate censoring systems for each application it censors, and that each such system has gaps in its logic, bugs in its implementation, and different network stacks—all of which we have shown can be exploited to evade censorship. Such heterogeneity severely complicates the process of evading censorship. Fortunately, we have shown that, by applying automated tools like Geneva [9], it is possible to efficiently evade (across multiple protocols) and understand a threat as nuanced (and buggy) as nation-state censors. Our code and data are publicly available at https://geneva.cs.umd.edu.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. We also thank our collaborators from the OTF and OONI communities, who have contributed insights and resources that made this work possible. This research was supported in part by the Open Technology Fund and NSF grants CNS-1816802 and CNS-1943240.

REFERENCES

[1] 1981. *Transmission Control Protocol.* RFC 793. RFC Editor. `https://www.rfc-editor.org/rfc/rfc793.txt`

[2] 2016. *DNS Transport over TCP - Implementation Requirements.* RFC 7766. RFC Editor. `https://tools.ietf.org/html/rfc7766`

[3] Claudio Agosti and Giovanni Pellerano. 2011. SniffJoke: transparent TCP connection scrambler. `https://github.com/vecna/sniffjoke`. (2011).

[4] agrabeli. 2017. Internet Censorship in Iran: Findings from 2014-2017. `https://blog.torproject.org/internet-censorship-iran-findings-2014-2017`. (2017).

[5] Anonymous. 2012. The Collateral Damage of Internet Censorship. *ACM SIGCOMM Computer Communication Review (CCR)* 42, 3 (2012), 21–27.

[6] Anonymous. 2014. Towards a Comprehensive Picture of the Great Firewall's DNS Censorship. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI).*

[7] Simurgh Aryan, Homa Aryan, and J. Alex Halderman. 2013. Internet Censorship in Iran: A First Look. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI).*

[8] Tod Beardsley and Jin Qian. 2010. The TCP Split Handshake: Practical Effects on Modern Network Equipment. *Network Protocols and Algorithms* 2, 1 (2010), 197–217.

[9] Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. 2019. Geneva: Evolving Censorship Evasion. In *ACM Conference on Computer and Communications Security (CCS).*

[10] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. 2016. Off-Path TCP Exploits: Global Rate Limit Considered Dangerous. In *USENIX Security Symposium.*

[11] Richard Clayton, Steven J. Murdoch, and Robert N. M. Watson. 2006. Ignoring the Great Firewall of China. In *Privacy Enhancing Technologies Symposium (PETS).*

[12] Roger Dingledine. 2012. Obfsproxy: the next step in the censorship arms race. `https://blog.torproject.org/obfsproxy-next-step-censorship-arms-race`. (2012).

[13] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium.*

[14] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. 2015. Examining How the Great Firewall Discovers Hidden Circumvention Servers. In *ACM Internet Measurement Conference (IMC).*

[15] David Fifield, Nate Hardison, Jonathan Ellithorpe, Emily Stark, Dan Boneh, Roger Dingledine, and Phil Porras. 2012. Evading Censorship with Browser-Based Proxies. In *Privacy Enhancing Technologies Symposium (PETS).*

[16] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. 2015. Blocking-resistant communication through domain fronting. In *Privacy Enhancing Technologies Symposium (PETS).*

[17] fqrouter. 2015. Detailed GFW's three blocking methods for SMTP protocol. `https://web.archive.org/web/20151121091522/http://fqrouter.tumblr.com/post/43400982633/%E8%AF%A6%E8%BF%B0gfw%E5%AF%B9smtp%E5%8D%8F%E8%AE%AE%E7%9A%84%E4%B8%89%E7%A7%8D%E5%B0%81%E9%94%81%E6%89%8B%E6%B3%95`. (2015).

[18] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. 2013. The Parrot is Dead: Observing Unobservable Network Communications. In *IEEE Symposium on Security and Privacy.*

[19] Jill Jermyn and Nicholas Weaver. 2017. Autosonda: Discovering Rules and Triggers of Censorship Devices. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI).*

[20] Dan Kaminsky. 2008. It's The End of the Cache As We Know It. `http://kurser.lobner.dk/dDist/DMK_BO2K8.pdf`. (2008).

[21] Sheharbano Khattak, Mobin Javed, Philip D. Anderson, and Vern Paxson. 2013. Towards Illuminating a Censorship Monitor's Model to Facilitate Evasion. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI).*

[22] George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS).*

[23] Fangfan Li, Abbas Razaghpanah, Arash Molavi Kakhki, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. 2017. lib.erate, (n): A library for exposing (traffic-classification) rules and avoiding them efficiently. In *ACM Internet Measurement Conference (IMC).*

[24] Richard McPherson, Amir Houmansadr, and Vitaly Shmatikov. 2016. CovertCast: Using Live Streaming to Evade Internet Censorship. In *Privacy Enhancing Technologies Symposium (PETS).*

[25] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. 2012. SkypeMorph: Protocol Obfuscation for Tor Bridges. In *ACM Conference on Computer and Communications Security (CCS).*

[26] Zubair Nabi. 2013. The Anatomy of Web Censorship in Pakistan. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI).*

[27] Kei Yin Ng, Anna Feldman, and Chris Leberknight. 2018. Detecting Censorable Content on Sina Weibo: A Pilot Study. In *Hellenic Conference on Artificial Intelligence (SETN).*

[28] Paul Pearce, Ben Jones, Frank Li, Roya Ensafi, Nick Feamster, Nick Weaver, and Vern Paxson. 2017. Global Measurement of DNS Manipulation. In *USENIX Security Symposium.*

[29] Thomas H. Ptacek and Timothy N. Newsham. 1998. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. In *Secure Networks.*

[30] Sigal Samuel. 2019. China is installing a secret surveillance app on touristsâĂŹ phones. `https://www.vox.com/future-perfect/2019/7/3/20681258/china-uighur-surveillance-app-tourist-phone`. (2019).

[31] Rachee Singh, Rishab Nithyanand, Sadia Afroz, Paul Pearce, Michael Carl Tschantz, Phillipa Gill, and Vern Paxson. 2017. Characterizing the Nature and Dynamics of Tor Exit Blocking. In *USENIX Security Symposium.*

[32] TelegramMessenger. 2019. MTProxy. `https://github.com/TelegramMessenger/MTProxy`. (2019).

[33] Inc. The Tor Project. [n. d.]. Tor Project: Bridges. `https://2019.www.torproject.org/docs/bridges.html.en`. ([n. d.]).

[34] Benjamin VanderSloot, Allison McDonald, Will Scott, J. Alex Halderman, and Roya Ensafi. 2018. Quack: Scalable Remote Measurement of Application-Layer Censorship. In *USENIX Security Symposium.*

[35] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer using Genetic Programming. In *European Symposium on Research in Computer Security (ESORICS).*

[36] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V. Krishnamurthy. 2017. Your State is Not Mine: A Closer Look at Evading Stateful Internet Censorship. In *ACM Internet Measurement Conference (IMC).*

[37] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. 2012. StegoTorus: A Camouflage Proxy for the Tor Anonymity System. In *ACM Conference on Computer and Communications Security (CCS).*

[38] Philipp Winter. 2012. brdgrd (Bridge Guard). `https://github.com/NullHypothesis/brdgrd`. (2012).

[39] Philipp Winter and Stefan Lindskog. 2012. How the Great Firewall of China is Blocking Tor. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI).*

[40] Eric Wustrow, Colleen M. Swanson, and J. Alex Halderman. 2014. TapDance: End-to-Middle Anticensorship without Flow Blocking. In *USENIX Annual Technical Conference.*

[41] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. 2011. Telex: Anticensorship in the Network Infrastructure. In *USENIX Annual Technical Conference.*

[42] Xueyang Xu, Morley Mao, and J. Alex Halderman. 2011. Internet Censorship in China: Where Does the Filtering Occur?. In *Passive and Active Network Measurement Workshop (PAM).*

[43] Tarun Kumar Yadav, Akshat Sinha, Devashish Gosain, Piyush Kumar Sharma, and Sambuddho Chakravarty. 2018. Where The Light Gets In: Analyzing Web Censorship Mechanisms in India. In *ACM Internet Measurement Conference (IMC).*

[44] Li Yuan. 2018. A Generation Grows Up in China Without Google, Facebook or Twitter. `https://www.nytimes.com/2018/08/06/technology/china-generation-blocked-internet.html`. (2018).

APPENDIX: GENEVA'S SYNTAX

For completeness, we include in this appendix a review of Geneva's syntax, which we use throughout the paper. For more details, please see [9].

Actions Geneva forms *action sequences* by composing any number of its five genetic building blocks, all of which mirror the packet manipulations that can occur at the network layer:

(1) `duplicate`(A_1, A_2) duplicates a given packet and applies action sequence A_1 to the first copy and then A_2 to the second.

(2) `fragment{protocol:offset:inOrder}`(A_1, A_2) performs IP-level packet fragmentation or transport-layer packet segmentation, thereby replacing one packet with two packets, and can deliver the fragments in- or out-of-order. It applies A_1 to the first fragment and A_2 to the second.

(3) `tamper{protocol:field:mode[:newValue]}`(A) modifies a particular `field` in the `protocol` header (or payload) of the packet. There are two `mode`s: `replace` changes the `field` to the `newValue`, whereas `corrupt` sets the `field` to an equal number of random bits. `tamper` recomputes the

appropriate checksums and lengths, unless `field` itself is a checksum or length; `corrupt` does not recompute checksums.

(4) `drop` discards the packet.
(5) `send` sends the packet.

Note that these primitives can be composed to construct *any* stream of packets, so long as `tamper` supports their `protocol` and `field`. In its original implementation [9], Geneva's `tamper` supported modifications of IPv4 and TCP; we explain in §4 how we extend this to also support IPv6, UDP, DNS, and FTP.

Triggers Geneva applies each action sequence only to packets that match a particular `protocol:field:value`. For example, a trigger of `TCP:flags:S` would apply to all TCP `SYN` packets. Geneva's triggers demand an *exact* match: for instance, `TCP:flags:S` does *not* match `SYN+ACK` packets.

Syntax Geneva represents its packet-manipulation strategies with a domain-specific language that composes the above actions. Geneva's syntax for representing a trigger:action-sequence pair is `[<trigger>]-<action sequence>-|`. A Geneva strategy can have a trigger:action-sequence pair for both inbound and outbound packets; the syntax for this is `<outbound> \/ <inbound>`.

As an example, Strategy 1 in §5 includes an outbound action sequence that triggers on `SYN+ACK` packets. It duplicates the `SYN+ACK` packet: it converts the first copy into a `RST` packet by overwriting the TCP flags, and likewise it converts the second copy to a `SYN` packet, and sends them both in that order. (Note that there is no "`send`" listed for the `tamper`; to simplify presentation, when no action is given, it defaults to `send`.) This strategy has no inbound action sequence. As we have shown, this is an effective server-side evasion strategy for evading censorship of DNS, FTP, HTTP, and HTTPS in China.