Detecting and Evading Censorship-in-Depth: A Case Study of Iran's Protocol Filter

Kevin Bock Yair Fax Kyle Reese Jasraj Singh Dave Levin
University of Maryland

Abstract

As the censorship arms race advances, some nation-states are deploying "censorship-in depth," composing multiple orthogonal censorship mechanisms. This can make it more difficult to both measure and evade censorship. Earlier this year, Iran deployed their *protocol filter* that permits only a small set of protocols (DNS, HTTP, and HTTPS) and censors connections using any other protocol. Iran composes their protocol filter with their standard censorship, threatening the success of existing evasion tools and measurement efforts.

In this paper, we present the first detailed analysis of Iran's protocol filter: how it works, its limitations, and how it can be defeated. We reverse engineer the fingerprints used by the protocol filter, enabling tool developers to bypass the filter, and report on multiple packet-manipulation strategies that defeat the filter. Despite acting concurrently with and on the same traffic as Iran's standard DPI-based censorship, we demonstrate that it is possible to engage with (and defeat) each censorship system in isolation. Our code is publicly available at https://geneva.cs.umd.edu.

1 Introduction

Censoring nation-states employ defense-in-depth, layering multiple orthogonal censorship mechanisms to make it more difficult to communicate with certain destinations or via certain protocols. Typically, such "censorship-in-depth" involves wholly different systems, such as combining lemon DNS responses [2, 14], IP blocking [5, 18], and TLS SNI blocking [4]. As a result, each form of censorship targets different packets, and can often be studied and defeated in isolation.

Far less common are censorship mechanisms that directly compose with one another, and target the same packets. In such situations, it is more difficult to study censorship because the two mechanisms' side effects can be conflated, and it is more difficult to evade censorship because one must evade *both* mechanisms simultaneously.

In early 2020, Iran launched such a form of censorship-in-depth by deploying their protocol filter. A *protocol filter* only allows a small list of protocols to be used, and censors protocols it forbids. A similar system in Iran was first reported on by Aryan et al. [3] in 2013, but to the best of our knowledge was not used for years until it was turned back on this year. We are also unfamiliar with any work detailing how Iran's protocol filter works or how to evade it—underscoring the difficulties inherent in measuring and circumventing censorship-in-depth.

In this paper, we present a detailed analysis of Iran's protocol filter: how it works, its limitations, and how it can be defeated. Even though the protocol filter operates concurrently with and on the same traffic as Iran's standard deep packet inspection (DPI)-based censorship, we demonstrate that it is possible to engage with each censoring mechanism in isolation. That is, we show how to evade the filter only, the regular censorship system only, and both in tandem. We used Geneva [4, 5], a genetic algorithm that automatically trains against censoring regimes in a black-box fashion to learn how to evade them. We report on the three evasion techniques Geneva discovered, as well as the results from our follow-on experiments that expose what the filter targets and what protocol fingerprints it uses.

The rest of the paper is organized as follows. §2 reviews prior work in measuring Iranian censorship. §3 describes our methodology and vantage points we use for our experiments. §4 presents our analysis of the protocol filter. §5 discusses how the protocol filter can be evaded. Finally, §6 concludes.

2 Prior Work

Iranian censorship has been studied in broader efforts to measure global censorship [6, 10–13, 15]. There have been fewer studies specific to how Iran's censorship operates. Notably, Anderson proposed a technique for detecting censorship via throttling in Iran [1].

The most closely related study to ours was a 2013 study by Aryan et al. [3]. They observed throttling between two vantage points that affected SSH, custom obfuscated SSH, and custom obfuscated HTTP. Since HTTP and HTTPS were unaffected, the authors hypothesized that Iran had deployed a protocol filter and were throttling connections that did not match HTTP and HTTPS. This behavior disappeared shortly after Iran's June 2013 election, and to the best of our knowledge, there have been no further reports on protocol filtering.

The censorship system observed by Aryan et al. in 2013 differs significantly from what we observe in 2020. First, the censorship mechanism is different; the prior system throttled forbidden protocols, but we observe outright dropping of all packets for some period of time. Second, the affected ports appear to be different; Aryan et al. observed filtering of SSH but not HTTP, but we find this no longer to be true (we find more nuanced behavior, and test a wider set of protocols). We are the first to delve deeply into how the protocol filter works and how to evade it, and thus cannot compare our results directly.

3 Methodology

We performed our experiments from 6 vantage points geographically dispersed within Iran: Fars, Isfahan, Khorasan, Razavi, Tehran, and Zanjan. These contain a mix of both residential and business networks.

In our experiments to measure the protocol filter (§4), we performed active measurements from these vantage points to servers we controlled outside of Iran, in Amazon EC2, Microsoft Azure, and DigitalOcean (located geographically in Japan, Ireland, the United States, Australia, and India). We find no significant difference in the behavior of the protocol filter across any of our vantage points or external servers, nor did we observe any change in the behavior of the filter during the course of our experiments to the time of writing.

To develop new evasion strategies (§5), we used Geneva [4, 5], an open-source genetic algorithm that trains directly against live censors and automatically discovers censorship evasion strategies. Geneva operates over an input packet stream that triggers censorship (such as an HTTP GET request with a censored keyword), composes multiple packet manipulations (such as duplicating packets, fragmenting them, and tampering with them), and mutates them over a series of "generations" until it discovers ways to evade censorship. Geneva need only run at one side of a connection, either the client [5] or the server [4]. We employ both in this study, and show that Iran's protocol filter can be evaded from either side.

4 Protocol Filter

In this section, we explore how Iran's protocol filter operates and whom it affects, and we detail precisely what properties it looks for when filtering DNS, HTTP, and HTTPS traffic.

4.1 How Iran's Protocol Filter Works

We performed active measurements to answer the following questions about the mechanics of the protocol filter:

How does the protocol filter censor forbidden protocols? Once a connection is observed to be communicating with a disallowed protocol, the protocol filter censors the connection.

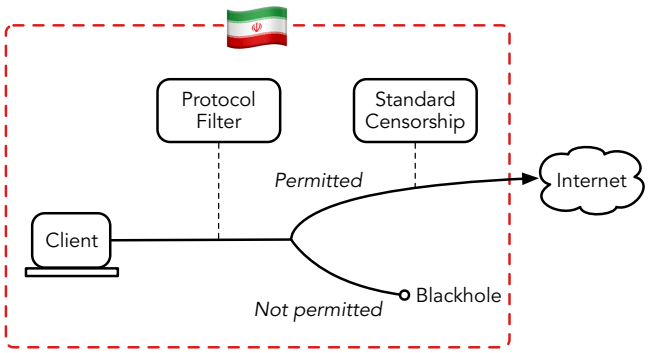

Figure 1: Iran's layered censorship system, employing defense in depth. Note that the order of censorship systems is unknown; this is simply a graphical depiction.

The filter censors connections by dropping all packets *from the client* in the flow[1]. The protocol filter can be triggered manually by sending any data stream on a monitored port that does not resemble a permitted protocol. Packets within the censored flow from the server are unaffected: the client still receives all of the packets sent by the server even after the protocol filter has been tripped. However, because the client cannot acknowledge or respond to any data, the connection is effectively censored.

Which ports and protocols does the filter monitor? From our vantage points, we made connections to servers we controlled outside of Iran and repeatedly sent messages containing just the string "`test`" (a payload that is not compliant with any of the protocols we tested) between sleeps for every possible destination port value (0-65535). Connections that time out identify which ports are likely affected by the filter. We repeated this experiment three times to validate our results.

We find that Iran's protocol filter affects only TCP traffic, and only on ports 53 (commonly DNS), 80 (commonly HTTP), and 443 (commonly HTTPS). Traffic sent on any other port is not filtered (and is therefore also not subject to Iran's standard censorship, which only operates over these same ports).

We then sent well-formatted messages of a variety of protocols (DNS, HTTP, HTTPS, SMTP, and SSH) on these ports. Of these, we find that the filter permits only DNS, HTTP, and HTTPS traffic. However, none of these are bound to their standard ports: the filter matches all three protocols on any of the three ports.

How many packets does the filter monitor? To answer this question, we sent multiple packets with non-protocol data (e.g., "`test`") before well-formatted allowed protocol

[1] We define "flow" to refer to the unique four-tuple of source and destination IP addresses and ports.

#IPs	Provider
1,453	Amazon Technologies Inc.
565	Cloudflare, Inc.
229	Akamai Technologies, Inc.
171	Amazon.com, Inc.
167	Fastly
146	DigitalOcean, LLC
97	Amazon Data Services Limited
92	RIPE Network Coordination Centre
64	Linode
60	Amazon Data Services

Table 1: Top 10 providers for *affected* IP addresses.

#IPs	Provider
4,541	Cloudflare, Inc.
1,465	*Unknown*
657	Google, LLC
657	Alisoft
580	Amazon Technologies, Inc.
544	Asia Pacific NIC
537	RIPE Network Coordination Centre
287	Alibaba.com LLC
277	Amazon.com, Inc.
253	Akamai Technologies, Inc.

Table 2: Top 10 providers for *unaffected* IP addresses

data. We determined that the filter monitors the first two *data-carrying* packets from the client at the start of a connection. If either of those two packets matches a protocol fingerprint, the flow is unharmed; if no packet does, the second packet and rest of the flow are dropped.

How long does the filter censor an offending flow? To test this, we intentionally tripped the protocol filter, waited an interval of time, and then sent non-data-carrying packets in the censored flow. Recall that once we trigger the filter, these packets will be dropped if the filter is still censoring our flow. We repeat this experiment with time intervals from 1 second to 90 seconds, each time using different source ports to avoid experiments conflicting with one another.

We find that, once tripped, the filter will continue to drop the offending flow's network traffic for 60 seconds, but each time an additional packet is sent in a flow, the 60 second timer resets. This means that, in practice, because TCP will retransmit packets that are not acknowledged, an offending flow will be affected by the filter for much longer than 60 seconds.

Is the protocol filter bidirectional? "Bidirectional" censorship systems do not differentiate between the client being the host inside or outside the nation-state. Iran's standard censorship system operates bidirectionally; it can be triggered by making requests from outside the country to servers inside the country (or vice versa). As a result, bidirectional censorship is often easier for researchers to study.

However, we find that the filter is *not* bidirectional: it only affects connections where the client is inside Iran. The server also receives almost no indication censorship has taken place. Recall that packets from the server are unaffected: unlike with the Great Firewall of China, which sends `RST`s in both directions [16], Iran's protocol filter only affects the packets sent by the client. This makes it difficult to identify and study the protocol filter without vantage points within Iran.

Can the protocol filter reassemble TCP segments? We repeatedly made valid but segmented DNS, HTTP, and HTTPS requests on filtered ports[2]. We find that segmenting our requests too many times incurs censorship from the protocol filter, indicating that, like Iran's regular censorship infrastructure [3–5], the filter is incapable of reassembling TCP segments. We also note that the filter also does not check the checksums of the packets it processes.

4.2 Whom the Filter Is Applied To

During our experiments, we noticed that the protocol filter is not applied to all server IP addresses. We find that whether or not an IP address is filtered is consistent between our vantage points; we could not identify any destination IP addresses for which the protocol filter was active from one vantage but inactive from another.

To identify which IP addresses are affected by the filter, we tested the effects of the filter on the Alexa top-20,000 most popular websites. To avoid the effects of DNS censorship or requesting IP addresses inside of Iran (as the requests would not cross the filter), we used `dig` outside of Iran to get IP addresses for all 20,000.

Inside of Iran, we set up an experiment with two conditions. Our experiment The first condition was a control: we made normal GET requests to all 20,000 IP addresses and recorded the success or failure of each request. The second condition tested for the filter: we requested all 20,000 IP addresses again, this time sending "`G`", "`ET`", and "`/`" in separate messages[3]. IP addresses that respond in the first condition but time out in the second condition are likely affected by the protocol filter. We perform this experiment ten times to validate the results.

Over all ten experiments, 3,595 IP addresses (17.9%) tripped the filter at least eight times. Of those, 3,499 were affected all ten times (17.4%), and 278 (1.4%) IP addresses were affected 3–7 times. Tables 1 and 2 show the number of IP addresses per provider that were affected and unaffected by the protocol filter, respectively. Overall, we find that IP

[2]We disabled Nagle's algorithm for this experiment to avoid spurious segment reassembly interfering with our results.

[3]We performed this experiment over raw sockets, with Nagle's algorithm again disabled.

address provider is not correlated with whether the filter affects an IP address or not, but some prefixes are affected significantly more heavily than others.

Case Study: Cloudflare We explore how Cloudflare in particular is affected by Iran's protocol filter, as Cloudflare hosts the most IP addresses from our dataset. Cloudflare makes its entire list of IP addresses publicly available[4]. Many of these prefixes are prohibitively large; instead of testing every IP address in each prefix, we sampled 256 IP addresses at random from each prefix to test. We performed a similar experiment to the one above: given a Cloudflare IP address, we made two requests to it (first normally, then segmented); IP addresses that respond in the first condition but time out in the second condition are likely affected. We repeated this experiment five times for each prefix.

We found that only two of Cloudflare's prefixes contained IP addresses that are affected by the filter: `104.18.0.0/16` and `104.31.82.0/24`. All of the IP addresses we tested in both of these prefixes were affected by the filter, but none of the IP addresses from the other prefixes were. It is unclear why these prefixes are targeted specifically. We were unable to identify any commonality between the sites hosted on these prefixes compared to unaffected prefixes.

We also performed traceroutes to a sample of the affected and unaffected IP addresses owned by Cloudflare. We were unable to identify consistent routing differences between them. At this time, it is not clear why the protocol filter affects the IP addresses it does.

4.3 Protocol Fingerprints

By repeatedly, manually tweaking the payloads of permitted protocols and observing what gets censored and what does not, we reverse engineered the filter's fingerprints for each protocol. Knowing the fingerprints can be a powerful tool for evaders: recall that the filter only monitors the first two data-carrying packets, and thus sending compliant packets at the start of a flow can allow all subsequent packets to bypass the filter. Since the filter will match any of these fingerprints on all three ports, any fingerprint can be used on any protocol-filtered ports.

DNS Fingerprint To match the protocol filter's fingerprint for DNS-over-TCP, the following conditions must be met:

1. The TCP payload must be at least 12 bytes long.
2. The query/response (`qr`) field must be 0.
3. The question count must be less than 15.
4. The answer count must be 0.
5. The structure of the TCP payload must be a valid DNS-over-UDP header, not a DNS-over-TCP header.

For example, the following message would be permitted by the DNS fingerprint:

```
\x00\x00\x01\x00\x00\x01
\x00\x00\x00\x00\x00\x00
```

The last requirement appears to be a bug in the implementation of the DNS fingerprint. Recall that the DNS-over-UDP header is slightly different than DNS-over-TCP's; over TCP, the DNS header includes a `length` field [9]. Since the filter is only active over TCP but does not take the extra field into account, it will *never match* a legitimate DNS-over-TCP packet. We believe the reason this oversight has not caused a significant issue is because DNS-over-TCP generally only requires a single data-carrying packet from the client, but Iran's protocol filter only begins dropping packets on the second data-carrying packet.

However, the faulty DNS fingerprint does still pose a problem: clients can reuse DNS-over-TCP connections [7]. In such cases, the filter would allow the first query, but block any subsequent queries made within 60 seconds.

HTTP Fingerprint To match the HTTP fingerprint, the following conditions must be met:

1. The TCP payload must be at least 8 bytes long.
2. The payload must start with one of the following HTTP verbs: `GET`, `POST`, `HEAD`, `CONNECT`, `OPTIONS`, `DELETE`, or `PUT`.
3. The HTTP verb must be followed by one space.

Note that two HTTP verbs are not supported by the protocol filter: `PATCH` and `TRACE`. Any website in the affected IP address space that uses either of these would be censored.

For example, a message permitted by the HTTP fingerprint is: `GET testing123`.

HTTPS Fingerprint To match the HTTPS fingerprint, the following conditions must be met.

1. The TCP payload must be at least 41 bytes long: 5 bytes for the TLS header, 36 bytes for the TLS Client Hello.
2. The length field of the TLS Header must correctly describe the length of the Client Hello.
3. The TLS version header (bytes 2 and 3 of the TCP payload) must be TLS 1.0 (`\x03\x01`), 1.1 (`\x03\x02`), or 1.2 (`\x03\x03`).

The last requirement makes no practical difference; real TLS 1.x Client Hellos all have TLS 1.0 in this field.

Also, the last requirement again appears to be an error in the design of the protocol filter. It allows TLS versions 1.0, 1.1, and 1.2 to be declared, but this version field is not used accurately in practice: TLS servers must accept any two byte value in this field so long as the first byte is `\x03` [8, Appendix E].

The HTTPS fingerprint *does not* filter specific HTTPS connections or applications; it simply enforces that generic TLS is used. As a result, censorship evasion tools that use TLS

[4] https://www.cloudflare.com/ips/

will likely be unaffected by the protocol filter at this time, as they will fulfill the above fingerprint requirements by default. This also means the protocol filter would spare more secure DNS transport protocols, such as DNS-over-HTTPS and DNS-over-TLS, if those protocols were used over one of the affected ports.

After the first 5 bytes of the packet (the type, version, and the length), the protocol filter does not check any of the remaining contents of the Client Hello. So long as the first 5 bytes match the fingerprint and the packet is of the proper length, the rest of the packet can comprise arbitrary data and bypass the filter.

An example message that matches the HTTPS fingerprint is: `\x16\x03\x01\x02\x00` followed by 512 null bytes, where `\x16` is the indication of a handshake, `\x03\x01` is TLS version (1.0), and `\x02\x00` is the length of the Client Hello (512 bytes).

Using Fingerprints We find that any of the fingerprints can be used to evade the filter. This presents an opportunity for censorship evasion tool developers: by sending any fingerprint at the start of a connection (or injecting it as an "insertion packet" [5, 16, 17]), we can ensure the filter will permit the rest of the flow, regardless of the actual protocol used. As we will see in the next section, Geneva also independently discovers strategies to inject innocuous fingerprints from the client-side.

5 Evading the Protocol Filter

In this section, we demonstrate how to evade Iran's protocol filter. We begin by demonstrating that known evasion strategies developed against Iran's standard censorship infrastructure do not apply to the protocol filter.

5.1 Old Strategies Do Not Apply

We first explored whether we could apply the same strategies that work against Iran's regular censorship system (affecting HTTP and HTTPS) to evade the protocol filter.[5] The only functioning strategy in Iran we are aware of is simple segmentation: simply splitting the censored request into multiple packets to take advantage of the censor's inability to reassemble TCP segments. We find that no other strategies identified by Geneva or prior work defeats Iran's censorship system.

Unfortunately, the effectiveness of the segmentation strategy depends on its implementation: it does not necessarily generalize, and at worst, can be *counterproductive* to evasion. In the worse case, if the HTTP request is segmented at a byte index less than 8, although the regular HTTP censor can no longer recognize it, the first packet will not match the protocol filter fingerprints and incur censorship. However, if the HTTP request is segmented such that the first segment fulfills the requirements of the HTTP fingerprint (it is at least 8 bytes long and is well-formed), and the `Host:` header is split across the second segment, the strategy can defeat *both* the protocol filter and the HTTP censor.

Importantly (and as we will see throughout this section), merely evading the regular censorship system does not necessarily imply defeating the protocol filter.

5.2 Evolving New Strategies

To identify new strategies to defeat the protocol filter, we leveraged Geneva, an open-source genetic algorithm designed to evolve packet-manipulation strategies to evade censorship [5]. Unlike most anti-censorship systems, Geneva does not require deployment at both ends of the connection: it runs exclusively at one side (client or server) and defeats censorship by manipulating the packet stream to confuse the censor without impacting the underlying connection. Geneva's packet manipulation strategies are expressed in a domain-specific language [5]; we describe each in plain English, but to allow us to unambiguously express strategies, we also present them using Geneva's language.

Geneva evaluates strategies with a fitness function, which returns a numeric score that captures how successful a given strategy is at evading censorship. Strategies that receive a higher score are more likely to survive and pass their "genetic code" to the next generation. Geneva tries to perform some forbidden action while a strategy manipulates the packet sequence: if the forbidden action succeeds, the fitness function rewards the strategy; if it fails, the strategy is punished. To apply Geneva to the protocol filter, we wrote a custom fitness function. Our custom fitness function connected to a vantage point outside of Iran and repeatedly sent messages to intentionally trip the filter. As Geneva allows for new fitness functions to be added dynamically, this required no changes to Geneva itself. Using this fitness function, we can test and train strategies directly against the filter. Note that this fitness function *does not* try to trigger the standard censorship system.

We deployed Geneva against the protocol filter with a single evolution from the client-side. We follow the original training hyperparameters outlined by Bock et al. [5] and configure Geneva with a population pool of 200 individuals and 50 generations. In under two hours, it discovered three simple strategies that defeat it. All of the strategies discussed herein have a 100% success rate against the protocol filter.

5.3 Discovered Evasion Strategies

Strategy 1: Innocuous Fingerprint The simplest strategy Geneva identified was to inject a `PSH/ACK` packet with a corrupt checksum and an innocuous HTTP request as the payload

[5] Contrary to the 2013 findings by Aryan et al. [3], from our vantage points, we find that Iran's *standard* censorship infrastructure no longer targets DNS-over-TCP at all.

immediately following the 3-way handshake. This trivially serves to bypass the filter, as it matches the protocol fingerprints. However, because the checksum is corrupt, the server will not accept this packet. There are other variants of this strategy that ensure that the filter processes the packet but the server does not, such as setting the TTL large enough to reach the censor but too small to reach the server [5].

We note that we did not need to encode anything in Geneva for it to discover this strategy; Geneva already has the capacity to replace the TCP payload with a well-formed query for several protocols within its `tamper` primitive.

Strategy 1: Innocuous Fingerprint

```
[TCP:flags:PA]-duplicate(
    tamper{TCP:load:replace:GET%20testing123}(
        tamper{TCP:chksum:corrupt},),
    ),)-| \/
```

Strategy 2: Double `FIN`s This strategy works by sending two additional packets *before* the 3-way handshake starts: two empty packets with the `FIN` flag set. To the server, the `FIN` packets are ignored, as they are not a part of an active connection, but the filter processes them and causes it to ignore the rest of the connection. We do not understand why this strategy works, though we hypothesize the `FIN` packets trick the filter into thinking it has already missed the relevant data packets, causing it to ignore the rest of the flow.

Strategy 2: Double `FIN`

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

Although Geneva discovers this strategy with two `FIN` packets, we find that sending more than two `FIN` packets also works.

Strategy 3: Nine `ACK`s The final client-side strategy we present is stranger than the first two: this strategy works by sending *nine copies* of the `ACK` packet during the 3-way handshake. This causes the filter to ignore the rest of the flow. This strategy works 100% of the time, and does not affect the underlying TCP connection. We hypothesize this works because the filter has some internal limit on the number of packets it will process for a given flow.

This strategy does not require `ACK` packets to work: any combination of non-data-carrying packets, including `RST`s or `SYN`s, is also effective. The nine injected packets also need not have the correct `seq` or `ack` numbers: the strategy defeats the protocol filter regardless.

Strategy 3: Nine `ACK`s

```
[TCP:flags:A]-duplicate(
    duplicate(duplicate,duplicate),
    duplicate(duplicate,duplicate(
        duplicate(duplicate,),
        ))
    )-|
```

This strategy presents us with an opportunity to evade the protocol filter *from the server side*. Server-side censorship evasion allows completely unmodified clients to connect directly to a server while the server subverts censorship on behalf of the clients [4].

Since Strategy 3 is effective with any set of TCP flags, if a server can induce the client to send nine non-data-carrying packets before it sends its forbidden request, we can defeat the protocol filter. We can accomplish this using a trick from prior deployments of Geneva: by sending multiple `SYN+ACK` packets during the three-way handshake with a corrupted `ack` number, we induce the client to respond with multiple `RST` packets.

Strategy 4: Nine Induced `RST`s, Server Side

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

Strategy 4: Nine Induced `RST`s This strategy sends nine corrupted `SYN+ACK`s, followed by one unaltered `SYN+ACK`. This induces the client to send nine `RST` packets with corrupted sequence numbers before sending its normal `ACK`, thereby evading the protocol filter.

We note that all of these strategies defeat the protocol filter *only*, not the regular censorship system that works in tandem. These allow us to bypass the filter and study Iran's existing DPI censorship system in isolation.

6 Conclusion

Early this year, Iran took the latest step in censorship-in-depth by deploying a protocol filter alongside their standard censorship infrastructure. In this paper, we have performed a deep investigation into Iran's protocol filter. Using vantage points within Iran and servers outside, we empirically demonstrated how the protocol filter works, what its fingerprints are, and to

a lesser extent whom it filters. Also, using Geneva [4, 5], an automated tool for discovering censorship evasion strategies, we identified four ways to bypass the protocol filter—three from client-side and one from server-side. Our results collectively show that Iran's two censorship systems can still be studied in isolation, and bypassed together.

As the censorship arms race advances, we anticipate censorship-in-depth to become increasingly common. Iran has had a greater capacity for censorship than they have exercised in the past, and the protocol filter can pose a threat to existing deployments of censorship-evasion tools (VPNs, Tor, etc.). Our results present a path forward that other anti-censorship researchers and activists can take to quickly and thoroughly detect, understand, and evade censorship-in-depth. To this end, we have made our code (including the custom fitness function for Geneva) publicly available at https://geneva.cs.umd.edu

Acknowledgments

We thank our shepherd David Fifield and the anonymous reviewers for their helpful feedback. We also thank 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] Collin Anderson. Dimming the Internet: Detecting Throttling as a Mechanism of Censorship in Iran. In *arXiv preprint arXiv:1306.4361*, 2013.

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

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

[4] Kevin Bock, George Hughey, Louis-Henri Merino, Tania Arya, Daniel Liscinsky, Regina Pogosian, and Dave Levin. Come as You Are: Helping Unmodified Clients Bypass Censorship with Server-side Evasion. In *ACM SIGCOMM*, 2020.

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

[6] CAIDA IODA: Internet Outage Detection and Analysis. https://ioda.caida.org/.

[7] J. Dickinson, S. Dickinson, R. Bellis, A. Mankin, and D. Wessels. DNS Transport over TCP - Implementation Requirements. https://tools.ietf.org/html/rfc7766, March 2016. RFC 7766.

[8] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol: Version 1.2. https://tools.ietf.org/html/rfc5246, August 2008. RFC 5246.

[9] Paul Mockapetris. Domain Names - Implementation and Specification. https://tools.ietf.org/html/rfc1035, November 1987. RFC 1035.

[10] Arian Akhavan Niaki, Shinyoung Cho, Zachary Weinberg, Nguyen Phong Hoang, Abbas Razaghpanah, Nicolas Christin, and Phillipa Gill. ICLab: A Global, Longitudinal Internet Censorship Measurement Platform. In *IEEE Symposium on Security and Privacy*, 2020.

[11] OONI: Open Observatory of Network Interference. https://ooni.org/.

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

[13] Ram Sundara Raman, Adrian Stoll, Jakub Dalek, Armin Sarabi, Reethika Ramesh, Will Scott, and Roya Ensafi. Measuring the Deployment of Network Censorship Filters at Global Scale. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[14] Sparks, Neo, Tank, Smith, and Dozer. The collateral damage of Internet censorship by DNS injection. *SIGCOMM Computer Communication Review*, 42(3):21–27, 2012.

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

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

[17] Zhongjie Wang, Shitong Zhu, Yue Cao, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, Kevin S. Chan, and Tracy D. Braun. SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[18] Philipp Winter and Jedidiah R. Crandall. The Great Firewall of China: How It Blocks Tor and Why It Is Hard to Pinpoint. *;login:*, 37(6):42–50, 2012.