GET /out: Automated Discovery of Application-Layer Censorship Evasion Strategies

Michael Harrity, Kevin Bock, Frederick Sell, and Dave Levin, *University of Maryland*

https://www.usenix.org/conference/usenixsecurity22/presentation/harrity

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

`GET /out`: Automated Discovery of Application-Layer Censorship Evasion Strategies

Michael Harrity Kevin Bock Frederick Sell Dave Levin
University of Maryland

Abstract

The censorship arms race has recently gone through a transformation, thanks to recent efforts showing that new ways to evade censorship can be discovered in an automated fashion. However, all of these prior automated efforts operate by manipulating TCP/IP headers; while impressive, deploying these have proven challenging, as header modifications often require greater privileges than are available to censorship circumvention apps. In that line of work, the application layer has gone largely unexplored. This is not without reason: the space of application messages is much larger and far less structured than TCP/IP headers.

In this paper, we present the first techniques to automate the discovery of new censorship evasion techniques purely in the application layer. We present a general solution and apply it specifically to HTTP and DNS censorship in China, India, and Kazakhstan. Our automated techniques discovered a total of 77 unique evasion strategies for HTTP and 9 for DNS, all of which require only application-layer modifications, making them easier to incorporate into apps and deploy. We analyze these strategies and shed new light into the inner workings of the censors. We find that the success of application-layer strategies can depend heavily on the type and version of the destination server. Surprisingly, a large class of our evasion strategies exploit instances in which censors are *more* RFC-compliant than popular application servers. We have made our code publicly available.

1 Introduction

Internet censorship by nation-state actors affects billions of users worldwide. While there are many forms of censorship—including blocking all transnational connections [1] and misinformation campaigns [22]—the most pervasive form of censorship comes in the form of in-network firewalls that monitor traffic for certain keywords or domain names and inject packets to tear-down connections (via TCP RSTs [47,59]) or misdirect clients (via spoofed DNS responses [6]).

For decades, an arms race has been waged between censoring nation-states and the researchers and activists seeking to enable a more free and open Internet. Recently, this arms race has led to powerful new mechanisms that *automate* the discovery of censorship circumvention strategies. In particular, *Alembic* [52], Geneva [14], and SymTCP [60] use varying techniques to find ways to manipulate TCP and IP headers in ways that confuse a censor but maintain end-to-end correctness between client and server. These techniques have arguably transformed the censorship arms race, allowing researchers to rapidly discover new evasion strategies, sometimes in a matter of hours [12].

Although powerful, by focusing only on TCP and IP headers, these tools suffer from several limitations:

Difficulty of deployment. As a practical matter, manipulating TCP and IP headers requires administrative privileges on most platforms. Some platforms limit such access (most mobile platforms do not have options for raw IP sockets), and some tools are reluctant to seek root privileges in the first place (notably, Tor [23]). Ideally, censorship evasion could take place by manipulating only *application-layer* data, which could take place in unprivileged usermode.

Lack of UDP support. Each of these prior tools only supported TCP-based applications. While this is extremely useful—spanning HTTP, HTTPS, and even DNS over TCP—it misses out on arguably the most important and common protocol: DNS (over UDP). Without reliable and uncensored DNS, users and applications would have to know IP addresses of the services they wish to connect to, which is untenable. However, UDP is such a simple protocol that manipulating UDP headers alone is unlikely to lead to viable censorship evasion strategies. Again, it would be ideal to explore how to alter application-layer data to evade censorship.

Surprisingly, despite advances in fuzzing techniques in other domains, techniques to automate the discovery of censorship evasion strategies in the application space remain relatively unexplored. At the time we started this project, we were unaware of any application-layer fuzzers that could generalize to multiple protocols and be modified to train against nation-state censorship infrastructure.

To address this, we present what we believe to be the first work that automatically discovers application-layer censorship evasion strategies. We build from an existing censorship evasion tool, Geneva [14], and extend it with application-layer fuzzing, and new fitness functions. The fuzzing engine we have built is not our primary contribution; indeed, it is a relatively standard fuzzer. What is surprising, however, is that, to the best of our knowledge, fuzzers have not been applied to censors at all.

Why study censorship of unencrypted protocols? HTTPS adoption is on the rise for most of the web [25], and browsers have started to request HTTPS by default [17]. Likewise, with development of encrypted DNS transports, such as DNS-over-TLS (DoT), DNS-over-HTTPS (DoH), and DNS-over-QUIC (DoQ), why study "vanilla" DNS? Despite the availability of more secure alternatives, unencrypted protocols are still heavily used around the world. Unencrypted DNS dominates; encrypted DNS alternatives are not yet widely adopted anywhere [37]. HTTP traffic is also still unfortunately prevalent in censored regimes. As of the time of this writing, HTTP traffic comprises nearly 20% of all traffic out of China to Cloudflare [21]. Worse yet, many censored websites still do not support HTTPS. We issued HTTPS requests to all the domains in Citizenlab's censorship test lists [19] and found that 18% of them did not support HTTPS, and 52% of the domains on their China-specific list did not load over HTTPS. Lastly, censors have grown increasingly hostile to new privacy advances in HTTPS, blocking TLS 1.3's ESNI [15], and launching HTTPS man-in-the-middle attacks [53, 54, 64]. Taken together, we believe HTTP and DNS will be prevalent in censored regimes for the foreseeable future. Our work shows that HTTP and DNS censorship can be evaded in easily deployable ways.

Contributions We make the following contributions:

- We take the first steps toward automating the discovery of application-layer censorship evasion strategies. These are easier to deploy than their headers-only counterparts.
- We use our fuzzer to perform a widescale empirical study in several countries (China, India, and Kazakhstan), two protocols (HTTP and DNS), and many different versions of server software.
- We discover and report on 77 unique circumvention strategies for HTTP and 9 for DNS. We describe many of these strategies in detail, and provide the full list in the appendix.
- We perform a thorough analysis of these strategies to gain new insights into how censorship is implemented in different places and how evasion strategies generalize at the application layer.

To enable the community to build on our results, we have made our code publicly available at:

https://geneva.cs.umd.edu

Roadmap The rest of this paper is structured as follows: §2 presents background and related work. §3 describes the design of our fuzzer, and the specific application to DNS and HTTP. §4 describes our experimental methodology. §5 presents our results from training over HTTP and §6 presents our results from training over DNS. We discuss these results, and what we can learn about censors in §7, and address ethical considerations in §8. Finally, §9 concludes.

2 Background and Related Work

In this section, we review nation-state network censors and provide an overview of prior work on fuzzing and past approaches to automate censorship evasion.

Nation-state censorship In this work, we focus on nation-state Internet censorship, which seeks to control which destinations and what content those in the nation can access on the Internet. Censorship infrastructures are made up of middleboxes, which rely on Deep Packet Inspection (DPI) to parse packet payloads to look for keywords or domains they wish to censor. Nation-state censors perform censorship in myriad ways: researchers have identified censors that inject TCP `RST`s to tear down connections [4, 14, 20, 44, 47, 59, 63], spoof DNS responses with incorrect answers to thwart address lookup [6, 7], send HTTP content for a block page [14, 67], or even drop traffic altogether [13].

As it is most relevant to this work, we draw special attention here to the mechanisms used to censor HTTP and DNS by nation-states. Censors commonly filter HTTP traffic in one of two ways: either by examining the requested domain (via the Host header), or by searching for forbidden keywords in the request string itself [13, 14, 67]. Censors in India and Kazakhstan examine the Host header, while the Great Firewall of China (GFW) uses both techniques. All three of these countries perform HTTP censorship differently. Airtel's ISP in India injects a block page to the user, the GFW injects `RST+ACK` packets to tear down the connection, and the Kazakhstani censor drops the offending traffic (and subsequent traffic) from the client. To censor DNS, censors commonly inject responses that contain an incorrect IP address. As of the time of this writing, China has deployed three independent DNS censorship systems running in parallel, each with their own fingerprints and block-lists [8]. Although some DNS and HTTP servers are censored by IP-blocking, we focus in this work on the active censorship performed at the application level.

All the nation-state censors we study in this paper only examine client requests: they do not parse server responses for forbidden content. Although there have been instances in the past of censors parsing server responses for censorship, this does not apply to the censors we study [67].

Another commonality amongst the nation-state censors we study in this work is that they *fail open*, meaning if they are unable to parse or censor a request, it will be allowed through. In the future, censors could theoretically switch to a fail-closed system, but prior work has noted that this could be costly and cause significant collateral damage [13].

One distinguishing factor between nation-state censorship and other middlebox deployments is the use of residual censorship, a punitive form of censorship used by some nation-states (such as China). With residual censorship, for a short period of time after a user makes a forbidden request, follow-up requests—even innocuous ones—will continue to be censored [11]. As we will see in §3, residual censorship can complicate censorship evasion.

Automatically Circumventing Censors Researchers have developed a myriad of techniques to evade censorship, such as tunneling traffic [33, 34, 42, 57, 62, 68], masking the true destination of traffic [23, 24, 35, 41, 65, 66], disguising traffic as another protocol [51, 58, 61], interfering with a censor's ability to track or parse traffic [44, 47, 59, 67], or avoiding the censoring country altogether [46, 48].

A recent area of work has explored mechanisms to *automatically* discover ways to evade censorship [14, 52, 60]. These approaches identify ways to modify the packet stream in such a way that the connection and request remain valid, but the censor is unable to correctly tear down the connection. Such automated approaches enable researchers to respond more quickly to new censorship events [12, 13, 15] or to scale the number of middleboxes under study [10]. For this work, due to the number of DNS resolvers, HTTP servers, and censoring countries, we use an automated approach for discovering application layer strategies.

We are familiar with three existing approaches to automating censorship evasion: Geneva [14], SymTCP [60], and *Alembic* [52]. Although each of these systems takes a different approach, the high level goal is the same: to find a sequence of packets that cause the censor to be unable to tear-down a connection (while preserving the connection itself). Geneva uses a genetic algorithm, and treats censors and destinations as black boxes, not unlike a fuzz tester. *Alembic* and SymTCP require access to the source code to perform symbolic execution of the server's implementations of TCP/IP. Requiring source code is reasonable when focusing on TCP/IP-based evasion strategies, as low-level network protocol implementations are unlikely to change frequently or vary significantly amongst different servers. However, application-layer code can change often and vary widely across servers. Thus, for this work, we chose to extend Geneva's black-box approach; we detail our design in §3.

Application Fuzzing Fuzz testers [45] mutate inputs non-deterministically in an effort to evaluate the correctness, security, and coverage of programs. Most relevant to our work is the space of grammar-based fuzzers, which define a grammar for inputs to the target protocol, and differential-based fuzzers, which send fuzzed inputs to multiple systems to identify any differences in behaviors. Grammar-based fuzzers (including those based on genetic algorithms) have been used successfully against many targets [5], including web applications [55] and other popular protocols [39]. The Peach Fuzzer is a grammar-based protocol fuzzer that allows a user to specify an input grammar, but only its Community Edition is available since Gitlab purchased it in 2020 [38]. WFuzz is another powerful fuzzer for HTTP web servers, but it has no support for other protocols or extending its grammar [49].

Our work differs from these fuzzers in two important ways. First, our work has a different goal from traditional fuzzers: instead of searching for modified inputs that elicit incorrect behavior from the application, our work must find a modified input that elicits *correct* behavior from the application but incorrect behavior from the eavesdropping censor. Second, our goal is not just to find any output that evades a censor, but rather to identify a modification that can be made to an *existing user query* to enable the user to bypass the censor. Whereas fuzz testers traditionally generate inputs, our approach generates what amounts to small pieces of code (built from Geneva's manipulation primitives) that are in turn applied to inputs (user traffic). Therefore, we search over the space of packet-manipulation actions, not over the input space (packets) itself.

Most similar to this work is a concurrent work T-Reqs [43], a grammar-based differential HTTP fuzzer used for discovering HTTP Request Smuggling attacks. HTTP Request Smuggling is the process of modifying an HTTP request such that a firewall or proxy fails to identify a second, hidden request. Although HTTP Request Smuggling is similar in spirit to censorship evasion, the goals are slightly different: with censorship evasion, our goal is not to sneak a second request past a censor, but to get the original request through. T-Reqs created a detailed context-free grammar for the HTTP specification, and randomly mutated inputs to discover differences in how popular HTTP proxies and servers handle content. With modification, T-Reqs (or other grammar-based fuzzers) could likely also be applied to censorship evasion.

3 Fuzzer Design

In this section, we detail the design and implementation of our fuzzer to automatically discover censorship circumvention strategies for HTTP requests and DNS queries.

Prior approaches to automating censorship evasion techniques have taken a fuzzing approach (Geneva [14]) or a symbolic execution approach (SymTCP [60] and *Alembic* [52]) to identify successful modifications to network packets. Our

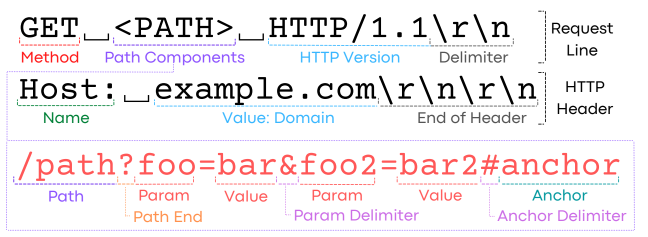


Figure 1: **Structure of an HTTP request** for `example.com`. Note that "␣" denotes where whitespace is required by the RFC, typically 1 space. Typically, HTTP Requests contain multiple headers separated by a `\r\n`.

goal is to work for a wide range of server vendors and versions. As a result, we will not always have access to the source code for every application layer server we need to train with (such as Google's public DNS resolver). Therefore, we take the fuzzing approach, and specifically extend Geneva's genetic algorithm to the application layer space.

A brief review of Geneva Geneva [14] builds censorship evasion strategies out of small, individual manipulation primitives (called *actions*) that can modify a packet. Geneva's actions mirror those that can take place on an IP network (duplicate, tamper, fragment, drop, and send). Each action takes parameter values, which Geneva chooses either at random or from packet captures of previous strategies. Geneva composes actions into *action trees*: duplicate and fragment have two children (the two copies or two halves of the packet), tamper has one child (the modified packet), and send and drop have no children. The action tree represents a packet-manipulation "program," and is executed via an in-order traversal of the tree. Each action tree has an associated *trigger* to describe which packets it should be applied to. While the individual manipulation actions are simple, Bock et al. [14] showed that composing them can be expressive enough to transform any set of packets into virtually any other set of packets. However, they focused almost exclusively on TCP/IP headers.

Geneva's genetic algorithm evaluates each strategy against a live censor by applying the strategy to a request for a forbidden resource. It assigns a numeric *fitness value* based on its success, overhead, and complexity in obtaining the forbidden resource. Geneva's genetic algorithm uses the fitness values to decide which strategies should survive to the proceeding generations and propagate.

Extending Geneva to application-layer requests We observe that in abstract, manipulating individual packets is tantamount to manipulating smaller components of a broader request. To translate this approach to the application-layer space, we identify the constituent units of the broader requests for HTTP and DNS. Though HTTP starts with a few constant fields (Method, Path, Version), the majority of an HTTP request is made up of a variable number of smaller

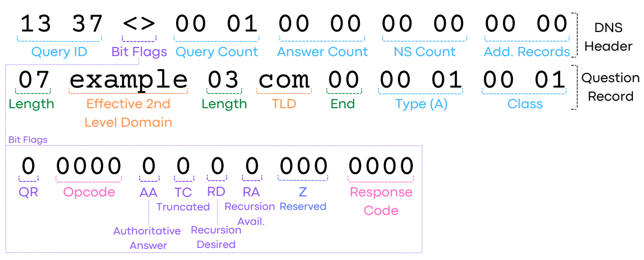


Figure 2: **Structure of a DNS request** for `example.com`. Note that the Bit Flags field (detailed in the lower box) is two bytes wide. Although DNS requests typically only contain one Question Record, the RFC [50] allows for multiple DNS Questions to be included with no separator between them.

HTTP headers. DNS requests, too, are comprised of constant fields, followed by a variable number of DNS question records. Therefore, we will allow our manipulations to access the constant fields and chain together modifications that affect the variable fields (HTTP Headers and DNS Question records, respectively). We note that even beyond the scope of this paper, other popular application layer protocols follow this pattern; for example, TLS packets usually have many TLS Messages and TLS Extensions.

3.1 Grammars

Next, we define a grammar that allows us to parse and modify these requests.

HTTP Grammar We specifically scope this work to HTTP Version 1 (HTTP/1.0 and HTTP/1.1). The HTTP protocol grammar is specified by RFCs 2616, 7230, 7231, 7232, 7233, 7234, 7235, and 3986 [9, 26–32]. An HTTP Request starts with the HTTP Method (sometimes called a "verb"), which defines the type of request, followed by a single space. Next, a request contains the request path, which specifies the resource location the HTTP request is for, as well as any HTTP parameters and values for the request. The path generally starts with a `/`, and if HTTP parameters are included, a `?` denotes the end of the path and the start of the query parameters. RFC 3986 specifies that in certain circumstances, other characters may mark the start of the path, but these are restricted to specific circumstances [9]. Multiple parameters may be specified within the request line by delimiting them with a `&`. After the path, a single space separates the HTTP version, and HTTP headers comprise the remainder of the request. The end of the starting line containing the method, path, and version is ended with a `\r\n`. Each line within the HTTP header is delimited with a `\r\n`, and the end of all the headers is marked with an empty line followed by a `\r\n`. This will look like a header followed by `\r\n\r\n`, signifying all following data is the message body. Using this grammar, our system will parse the given HTTP request to extract the constant fields (Method, Path, Version), and variable headers into a list. See Figure 1 for an example HTTP request.

DNS Grammar In this work, we focus specifically on normal DNS Requests, so extensions or other DNS technologies (such as DNSSEC or running DNS over other protocols) are out of scope. The structure of DNS queries are defined by RFC 1035 [50]. DNS Queries are comprised of a set of fixed constant fields, followed by a variable number of DNS Question Records which specify the domains to lookup. By convention, DNS Queries usually only have 1 DNS Question (and as we will see in Section 6, many DNS servers will only respond to queries with 1 DNS Question), but the RFC still permits multiple Question Records in a request. See Figure 2 for the fields in a DNS Query.

3.2 Manipulations

Now that we can parse HTTP and DNS requests, our goal will be to design simple manipulation primitives that can be composed together such that for a given application, a strategy can transform any request into any other request. Therefore, our actions must be able to add, remove, or manipulate any constituent components of the request. We will define `duplicate` and `drop` to add or remove components from a request, but most importantly, we must be able to modify one of these components. Unfortunately, application-layer data is significantly less structured than packet headers, and HTTP headers in particular are primarily composed of raw, unstructured text. We require a new set of actions that will allow us to modify unstructured text.

Inserting New Bytes We define a new modification primitive to insert new bytes into a given header or question record:

```
insert(<VALUE>, <WHERE>, <COMPONENT>, <NUM>)
```

The action takes four parameters, which control what bytes are inserted, where within the existing text they should be inserted (`start`, `middle`, `end`, `random`), which component should be affected, if applicable (such as HTTP header `name` or `value`), and the number of times the bytes should be inserted. As the genetic algorithm runs, these parameters can be mutated and learned through the process of evolution.

Replacing Bytes We define a second modification primitive to allow our system to replace existing bytes within a given header or question record:

```
replace(<VALUE>, <COMPONENT>, <NUM>)
```

The action takes three parameters, what bytes should replace the existing text, which component should be affected, if applicable (such as HTTP header `name` or `value`), and the number of times the bytes should be placed in that location. This action also incorporates the ability to delete the component, by replacing it with a value of an empty string. As the genetic algorithm runs, these parameters can be mutated and learned through the process of evolution.

Changing String Case We define this action to take in a string and change the case of all alphabetical characters in the header name and value.

```
changecase(<CASE>)
```

This action takes one parameter, which is what case all letters should be changed to. It can change all characters to lower or upper case, or randomly assign each letter to be upper or lower case, irrespective of its current case. Nothing will happen to non-alphabetical characters.

3.3 Evaluating Evasion Strategies

In this work, we do not modify Geneva's underlying genetic algorithm, but we do modify how it evaluates each candidate strategy. We evaluate strategies directly against real-world censors by using them to modify a request for forbidden resources, sending the resulting request across a censor to a destination server, and checking that the request did not trigger censorship and successfully obtained the forbidden content. Each time we train the genetic algorithm, we initialize it with a clean slate with no access to prior results or knowledge of the censorship system. Our system executes each training run for a pre-specified number of generations or until population convergence occurs. After our modified Geneva automatically discovers new evasion strategies, we follow up with manual post-hoc analysis to understand *why* and *in what conditions* the strategies work.

We note that our system is tolerant to transient network failures. Some transient failures are self-correcting: for example, during training a transient failure of the censorship system itself could cause a strategy to be mistaken as successful. In subsequent generations however, the strategy would (correctly) fail, receive a negative fitness, and not propagate to future generations, and this is handled under the hood within Geneva's existing genetic algorithm [14].

HTTP Evaluation To evaluate HTTP strategies, our system makes a request that either contains a forbidden Host header, or a forbidden keyword in the request string. To train for HTTP strategies, we run our system from vantage points we control within a censored country and make a request to a server we control outside the censored country. This allows us to control the server type and version.

Our design must account for the effects of *residual censorship*. In China, for 90 seconds after the censor tears down a forbidden request, any follow-up request to the same three-tuple (server IP, server port, and client IP) will result in censorship, even if that request is benign. Fortunately, China's HTTP censorship is active on every destination port. Therefore, we use a different destination port within a large range of ports for every strategy, and forward all of these ports to a single port the server runs on. With this design, we are able to train quickly without suffering the effects of residual censorship.

DNS Evaluation To evaluate DNS strategies, our system applies each strategy to a DNS request that contains a DNS Question Record for a forbidden domain.

Recall that the Great Firewall of China runs three separate DNS censorship systems, and any subset of them can respond to a forbidden query. The GFW does not drop the offending query packet, so in addition to the DNS injectors, the intended destination of the request will also receive it and respond. As a consequence, if a client within China makes a forbidden DNS query to a reachable DNS server outside of China, the client could get anywhere from 0 to 4 DNS responses (up to three from the injectors, optionally followed by the real uncensored response). Since any strategy could affect the response or any of the censors or the destination server itself, it is difficult to identify whether a given DNS response constitutes censorship without issuing a follow-up query to the IP address in the response, which is slow.

To avoid this problem, we run training for DNS outside of China. To evaluate a strategy, our system applies the strategy to a query for a forbidden domain (such as `google.sm`). First, the resulting modified query is sent to an uncensored DNS server, such as an open resolver, like Google's `8.8.8.8`. If the strategy successfully gets a response from the DNS server, we know the query is valid, and the strategy receives a higher fitness value. Next, we send the same modified query into China to a machine under our control that is not running any DNS server at all. In this case, if the query gets any DNS responses, we know these responses originated from the Great Firewall (and punishes the strategy's fitness value).

Importantly, as with HTTP (and applications from prior work), we give a lower fitness value to a strategy that breaks the underlying request than if the resulting request was still valid but experienced censorship. This encourages the genetic algorithm to explore the space of strategies that preserve the validity of the original request, but can impact the censor.

3.4 Evasion Proxy for Ease of Use

To make our strategies useful for real users, we developed a standalone "proxy" application, which applies a given strategy to live traffic. This proxy application accepts the original strategy syntax, so any of the strategies presented herein can be copied and used, with no further set up. We tested this proxy by browsing with it through our vantage point in India to multiple forbidden websites, and validate that these strategies can be used on real user traffic. We make this proxy available with our publicly released code.

DNS Resolver Org.	Resolver Address
Cloudflare	1.1.1.1
Google	8.8.8.8
Quad9	9.9.9.9
OpenDNS	208.67.222.222
CleanBrowsing	185.228.168.168
ComodoSecure	8.26.56.26
DNS.Watch	84.200.69.80
Verisign	64.6.64.6

Table 1: DNS Open Resolvers we conduct experiments with. All of these open resolvers are accessible from within China.

4 Experiment Methodology

In this section, we describe our experiment methodology for training our system. As we will see, many application-layer strategies only work with specific destination servers; therefore, we need to repeatedly train to different popular servers for DNS and HTTP.

HTTP Servers On September 3rd 2020, we downloaded a list of the most popular HTTP servers currently in use from W3Techs [2] and BuiltWith [3]. According to both resources, Apache[1] was the most popular (with 36.5% and 35% estimated market share from each respective resource) and Nginx[2] was the second most popular (with 32.5% and 34% share respectively). W3Techs identified Cloudflare's hosting as the third most popular (15.7%), and both identified Microsoft IIS as the next most popular (7.9% and 13% respectively). For this work, we choose to focus on the servers with the maximal market share: Apache and Nginx. Deployments of Apache and Nginx span many versions; we selected the four most popular versions for each, according to W3Techs [2], specifically 2.4.6, 2.4.18, 2.4.29, and 2.4.43 for Apache and 1.13.4, 1.14.1, 1.16.1, and 1.19.0 for Nginx.

DNS Resolvers Most DNS traffic is handled by large resolvers; in 2019, DNS Observatory studied over 1 trillion DNS transactions and found that over 60% of them were handled by just 1,000 nameservers and flowed to authoritative servers run by less than 10 organizations [36]. For this reason, we choose to train directly with the most popular open resolvers. We tested if these resolvers are affected by IP-blocking censorship by making innocuous DNS lookups from our vantage point within China, and found that none are affected and all are reachable. See Table 1 for a full list of the resolvers against which we test.

Vantage Points We obtained vantage points in China (Beijing), India (Bangalore), and Kazakhstan (Almaty) to use in our experiments. We also set up servers we controlled in un-

[1]`https://www.apache.org`
[2]`https://www.nginx.com`

censored countries in Europe (Ireland), Japan (Tokyo), and the United States (at our university) to conduct experiments.

To train our system in these countries, our system triggers censorship depending on the country and type of censorship. For HTTP, in India and Kazakhstan, we sent an HTTP request with a forbidden domain in the Host header (`youporn.com`). Recall that China censors HTTP both by censoring keywords in the HTTP parameter list and by examining the Host header, so we train in China against both types of censorship (specifically, using the forbidden word `ultrasurf` as an HTTP parameter and `youporn.com` in the Host header). For DNS, we send a DNS query containing a question for a domain forbidden by China between two hosts we control across the censor. Recall that the landscape of DNS censorship is more complex in China than with HTTP, with three parallel DNS censorship injectors. We specifically choose to train with only those domains that are affected by all three censorship systems, such as `google.sm`.

Like all censorship research, our results are limited by the censorship we can access and test with; still, we believe that testing against three different censors for HTTP and DNS is sufficient breadth to demonstrate the generalizability of this technique.

HTTP Experiment Methodology We ran our experiments over the span of seventeen months, starting in December 2020. We evaluated against a diverse set of censorship types: India, Kazakhstan, China-Host, and China-keyword. For all four types of censors, and for all eight types/versions of HTTP servers, we conducted 5 training runs (160 in total). Each training run executed with a population pool of 500 individuals for 50 generations.

For each HTTP server, for training runs with Host header based censorship, we configure the server with a `VirtualHost` to require the Host header; this prevents a strategy from "succeeding" by simply removing, or mangling the forbidden value from the request. For keyword-based censorship training, the fitness function requires that the forbidden keyword is present in the outbound request. Note also that we limited our system to only actions at the application layer space, so TCP segmentation is not permitted, and the fitness function cannot make additional requests.

To avoid residual censorship in China, we ensured that no two strategies used the same destination port within a 90-second window. In particular, we allocated 15,000 contiguous ports, assigned each port to one strategy, and used `iptables` on the server to redirect all of these ports to a single port that hosted the server. Since residual censorship lasts for 90 seconds, we evaluated fewer than 167 strategies per second (15,000/90) so as not to exhaust our ports.

We evaluate each strategy serially, with no sleep in between. On average, the fitness function for HTTP evaluates 1-2 strategies per second and each HTTP request is initially 40 bytes. For example, an initial HTTP request (before it is modified by a strategy) in India is:

```
GET / HTTP/1.1\r\n
Host: youporn.com\r\n\r\n
```

We also tested if this technique is applicable to servers outside our control by training to 12 censored domains over HTTP (6 in KZ, 6 in IN); we show the successful results of these experiments in §5.3.

DNS Experiment Methodology For DNS, we chose to train against all three of China's DNS Injectors simultaneously, so the resulting strategies could be applied to any forbidden domains. We can do this by using a domain that appears on all three injectors' block-lists. We reached out to Anonymous et al.—who originally discovered that the GFW's DNS infrastructure was powered by three injectors—and the authors provided a list of domains that appeared on each injectors' block-lists [8]. By choosing which domain name we used to trigger censorship, we can tailor our training to specific DNS injectors. For this work, we chose to use `google.sm`, which appears on the block-lists for all three injectors.

For each of the 8 DNS resolvers we train with, we conduct 5 training runs. We use the same hyperparameters for training as with HTTP: each training run is executed with a population pool of 500 individuals over 50 generations.

Since DNS runs on UDP, the fitness function can evaluate the strategies much more quickly—about 20 strategies per second—and each request is initially 27 bytes. The total network load for DNS training to an open resolver is approximately 11kbps, and lasts than less approximately 20 minutes per training run; these network loads should be negligible for resolvers of this size. Fortunately, residual censorship is not a concern for DNS in China, allowing us to train more quickly.

Post-Hoc Analysis After each training run for DNS and HTTP, we perform manual analysis to investigate the strategies our system discovers and perform manual experiments to understand why each strategy works. We also follow precedent from prior Geneva work: after each training run, we disable any fields or actions that dominated the search space to encourage strategy diversity. For example, if the first training run discovers that any changes to a specific field always evade censorship and those strategies quickly dominate, we remove that field from the proceeding training runs to encourage the algorithm to discover new strategies.

Strategy Success Rates After we completed all the training runs, we re-tested every discovered strategy against every other server version in each country. We tested every DNS strategy 1,000 times and HTTP strategy 100 times. We did not observe any differences in the success rates of our strategies from when they were initially collected to this success rate testing.

Manual Verification To confirm that the strategies we discovered work the way we expect, we performed several additional manual verification steps. First, we manually ran every strategy presented in this paper against every server type and confirmed we receive the correct server response page. For a more rigorous check for a subset of our servers, we also compared server responses to unmodified requests and requests modified by our strategies and confirmed they were byte-wise identical. Finally, as mentioned in §3.4, we manually tested a sample of strategies in India with a real web browser using our proxy server and validated that we could browse blocked websites successfully. We emphasize that these manual steps were done strictly for verification and understanding; our modified Geneva discovered the strategies in a fully automated fashion.

5 HTTP Results

In this section, we will detail our results from training our system against various forms of HTTP censorship around the world. Specifically, we train against Host- and Keyword-based censorship in China, and Host-based censorship in India and Kazakhstan. For a strategy to succeed, it must modify a request sufficiently to evade censorship, while still being accepted by the destination server.

5.1 Summary Results

We only report on strategies for which at least one HTTP server we tested correctly responded. For each successful strategy, there are often many ways to craft successful variants of that strategy that functionally do the same thing. Thus, to give a more conservative count of the number of strategies we discover, we only report on strategies that work for a unique reason.

In total, we identify 77 unique HTTP strategies. We manually performed experiments to understand how they work and determine their success rate against each country and HTTP server. The strategies' success around the world varies, but we were able to find multiple strategies against every censor we trained against. We found the most successful strategies against Airtel's censorship in India: of the 77 strategies we discovered, an 56 of them bypassed the Indian censor. A total of 29 strategies bypass the Kazakhstani censor. In China, we found a total of 22 evasion strategies that evaded path-based censorship, and 27 strategies that evaded the host-based censorship.

As we will see, the number of strategies we discover against each censor does not necessarily imply that the censor is non-compliant with the RFCs. On the contrary, our results suggest that the *more* RFC-compliant a censor is, the more opportunities there are for evasion.

Due to space constraints, we cannot discuss every strategy we discovered. Instead, in this section, we will describe each strategy family and give examples of where and why they work. We list all 77 unique HTTP strategies in Tables 2 and 4.

5.2 Evasion Strategies

Version Mangling The first strategy we discuss is surprisingly simple: corrupting the HTTP version. The resulting request would seem to be in violation of the RFC, as RFC 7230 (Section 2.6), specifies that servers should respond with an error page if they receive an unknown version. However, the RFC also admits that a server may respond anyway "if it is known or suspected that the client incorrectly implements the HTTP specification and is incapable of correctly processing later response versions". We find that several server versions (Apache 2.4.6 and 2.4.18) choose to be maximally permissive and ignore malformed versions, responding normally. We also find that the tested versions of Nginx will respond normally if the version is corrupted by inserting a `%` character (`%25`).

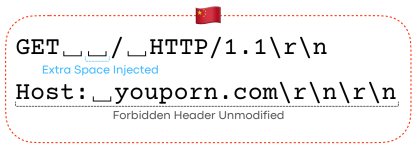
This strategy evades censorship for both types of HTTP censorship in China, which is surprising: the HTTP version appears *after* the path that contains the forbidden keyword. This suggests that the censor validates the HTTP Version or will only perform DPI on the packet if the Version has an expected value. Version mangling also defeats censorship in India.

Kazakhstan, on the other hand, will censor a request with a corrupted version unless enough bytes are inserted into the field to lengthen it to 1,434 bytes long. At this point, the censor ignores the request, and we can evade successfully. We do not believe the Kazakhstani censor is doing any validation of the version; instead, we believe it is more likely that the censor has a limit to the number of bytes it will buffer before processing it.

Four Element Request Line The HTTP RFCs specify that the request line should be split on whitespace between the three request line parameters. We discovered a class of strategy that inserts a space into the middle of a field within the path or the version, in such a way that the important aspects of the path and HTTP parameters can still be understood. We believe this strategy works for the same reason that HTTP version mangling does. When a censor's DPI splits the request line, the third component is no longer a well-formed HTTP version. These strategies are also in violation of the RFC, but are still understood by versions of Apache.

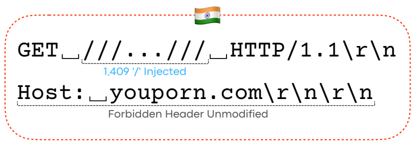
The reason these strategies work is the initial path is being interpreted as the real path, HTTP server logs confirmed this, whereas the whitespace is creating a new request line element that might be interpreted as the version. We found these strategies worked in China and India, but not in Kazakhstan, which is consistent with our results from HTTP Version mangling.

Changing Case In HTTP requests, there are some components that the RFCs specify should be case-sensitive, including the method (RFC7230 Section-3.1.1) and version (RFC7230 Section-2.6), while others that should be case-insensitive, like header names (RFC7230 Section-3.2). We discovered strategies that change the case of the method, version, or of the `Host` header name itself (such as to `host`). All



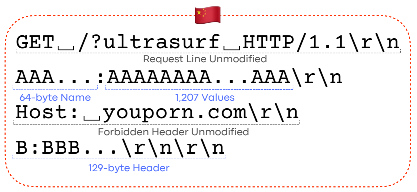
```
GET␣␣/␣HTTP/1.1\r\n
Host:␣youporn.com\r\n\r\n
```

(a) *Request Line Whitespace*: Inserting an extra space between the Method and Path evades Host-based censorship in China. The censor assumes that there will only be one whitespace character in that location, but the RFC [31] permits more.



```
GET␣///...///␣HTTP/1.1\r\n
Host:␣youporn.com\r\n\r\n
```

(b) *Induced Segmentation*: Evades Airtel's censorship in India by forcing the request to be segmented across two TCP packets. The entire request, with headers, is larger than the Ethernet MTU, but India's censorship does not properly handle segmentation.



```
GET␣/?ultrasurf␣HTTP/1.1\r\n
AAA...:AAAAAAAA...AAA\r\n
Host:␣youporn.com\r\n
B:BBB...\r\n\r\n
```

(c) *Sandwich Strategy*: Evades keyword- and Host header-based censorship in China. This breaks the parsing in such a way that the censor cannot process the host header, which is needed for path reconstruction.

Figure 3: **Examples of three HTTP strategies we discover.** Each of these strategies defeats censorship for a different censor or mechanism (Header-based in China, in India, and Keyword-based in China).

of these work in India, but do not work in China or Kazakhstan. These strategies tell us that the Airtel censor is too strict in how it processes HTTP requests.

Request Line Whitespace RFC 7230 specifies that a single space should delimit between the Method, Path, and Version fields, but that servers should ignore extraneous whitespace before the method and after the version, and treat any contiguous blocks of whitespace as a single space [31, Section 3.5]. The RFC classifies "whitespace" as space (URL-encoded: `%20`), horizontal tab (`%09`), vertical tab (`%0B`), form feed (`%0C`), or bare carriage return (%0D). It also states that servers should treat newlines (`%0A`) as a `\r\n`, or the intended line delimiter.

These rules permit a wide variety of ways to modify a request line without altering syntax, and we found a total of 33 unique strategies that take advantage of inserting some form of whitespace within the request line. Some of these strategies are simple: in China, we can insert a single additional space after the HTTP Method and evade Host-based censorship (though this does not work for keyword-based censorship). We present an example in Figure 3(a) . Other strategies in this family are more complicated: in Kazakhstan, if a strategy inserts 1,434 whitespace characters after any item in the request line, it will evade the censor. We find that the strategy can get away with inserting only one whitespace character if it inserts it before the method. The Indian censor we tested was the most brittle with respect to whitespace. We discover other strategies in this class that work by inserting certain patterns of additional whitespace between the HTTP version and the `\r\n`. For example, appending a `\n\t` to the Version is not sufficient to evade the Indian censor, but `\n\t\n\t`, (or any number of spaces), will evade.

Although not all of our servers under test correctly responded to all of these strategies, most of them did, and whitespace-inserting strategies remain the strategy class that is most broadly successful across server and censor types.

Host Header Whitespace Similar to inserting whitespace around the request line, we also discovered 21 strategies that involve inserting certain amounts of specific whitespace characters around the Host header. RFC 7230 defines the correct format for headers as:

```
<NAME>:<OPT WSPACE><VALUE><OPT WSPACE>
```

where `<OPT WSPACE>` is optional whitespace, consisting only of spaces and horizontal tabs (RFC 7230, section 3.2) [31]. Strategies in this class insert additional whitespace into the optional whitespace locations or even around the header name itself.

In China, inserting whitespace before the header name (which is not RFC compliant), successfully evades Host-based censorship, but not path-based censorship. This suggests the GFW fails to parse headers that begin with whitespace, but it can still parse and identify forbidden keywords in the path. In India, we find that if a strategy inserts a whitespace character before or after the Host header name, or a single newline character around the Host header value, it will evade the censor.

In Kazakhstan, we found similar rules for which strategies work and why. We find that inserting one space after the header value or anywhere around the name evades. Using tabs or newlines instead of spaces works only slightly changes the requirements: inserting one tab anywhere around the header name or value or a newline anywhere except the end of the header, evades censorship.

Induced Segmentation One simple-seeming strategy we discovered in India works by simply inserting more data anywhere in the request to make it at least 1,449 bytes long. We present an example in Figure 3(b) . What is special about this number of bytes? With an HTTP request at least 1,449 bytes long, the added bytes for IP (20 bytes), and TCP headers (32 bytes, including the timestamp option) total 52, bringing the request size up to 1501 bytes. Since this is exactly one byte past the Ethernet MTU (1500 bytes) [40], we conclude that this strategy works by inducing segmentation. Prior work has found that the Indian censor can be evaded by simple segmentation, which supports this hypothesis [13].

We observe a similar strategy in Kazakhstan, but slightly more complexity is required. Instead of inducing segmentation anywhere in the request, our system discovered that if a strategy induces segmentation specifically at the byte index between the Host header name and value, it will evade censorship. It accomplishes this by inserting enough bytes such that the 1,449th byte is the last byte before the host header value, and the final two bytes before the host header value must both be spaces. We do not understand why two spaces are required for this strategy to work. These strategies are perfectly RFC-compliant, and every server we tested responded correctly. We found no evidence that this type of strategy has any effect on China's censors, however many of these strategies still evade in China due to other unrelated reasons, such as whitespace insertion or long header names.

Path Confusion Another family of strategies we discovered involves adding characters, parameters, or anchors to the path that are ignored by the server, but processed by the censor. For example, the strategy that inserts a single `?` *before* the start of the path evades in India and China (for both header and keyword censorship). Technically, `?` is only allowed to start a path if the path is empty, but we find that every Apache version we tested still correctly processed the path and the request. Another strategy in this family works by inserting a new very long HTTP parameter (at least 1,003 bytes long) before the forbidden keyword; this only works in China.

Host Header Shield The next strategy we discuss evades China's keyword and host-based censorship. Recall that inserting a single space after the HTTP Method is sufficient to evade China's Host-based censorship, but does not evade its keyword censorship. Our system found that by *also* inserting a new header before the host header with a header name that is at least 64 bytes long, it could evade both keyword and Host censorship simultaneously. This only works if whitespace is inserted before the HTTP Method or between the Method and Path, not anywhere else in the request line.

Why does this strategy work? It seems strange that adding a space before the path is required to evade Host-based censorship, and adding a long header before the Host header is required to evade keyword-based censorship (although we note this is sufficient on its own to evade header censorship). Our results suggest that a 64+ byte header name prevents the GFW from reading any further headers, which explains why the longer header is enough to defeat header censorship. We believe that the added space in the request line forces the GFW to look for the Host header before it processes the path. If the strategy does not include the modified header, or includes it after the Host header, the GFW inspects the path correctly, but if we interfere with this search for the Host header, the GFW fails to check the contents of the path.

Sandwich Strategy The last type of strategy we will analyze creates a sandwich of headers around the Host header, and we find that if these headers are crafted in the correct way, we can bypass keyword and header censorship in China and India. We present an example in Figure 3(c) .

In China, we find the following constraints:

- The first header that appears in the packet must have at least 64 characters in the header name.
- Enough data must be transferred in the headers such that some header's value starts at least 1280 bytes away from the start of the headers (first character of header value is at least the 1281st byte after the request line)
- The last header must be at least 129 bytes total (including ending `\r\n` and the separator ":")
- The Host header cannot be the first or last header.

This type of strategy works in both header- and path-based censorship, though we note it is technically overkill to defeat header-based, as a single long (64+ byte) header is enough. We also found that many sandwich strategies work in India, but only because the header size induces segmentation.

5.3 External Validation

To demonstrate that this approach works without control of the destination server, we trained our system against 12 censored domains (6 in Kazakhstan and 6 in India). We downloaded CitizenLab's censorship test lists for India and Kazakhstan [18], and tested all the domains to identify which were censored, and then chose 6 randomly for each country. We do not know the type or version of these servers.

Our system successfully identified evasion strategies for every domain we tested. Across these twelve experiments, we discovered 13 unique strategies, 7 of which do not work on any of the other HTTP servers we tested. These experiments demonstrate the generalizability of this technique to new application servers, and underscore the importance of having an automated solution in this space.

Method Mangling Here, we showcase a surprising class of strategies we discovered during this validation phase. This strategy works by simply corrupting the HTTP method and replacing it with another string. Note that this is absolutely not RFC-compliant; RFC 7231 (Section 4) specifically mentions that any non-conforming method should be denied [32]. However, we find that some HTTP servers, when confronted with an HTTP method they do not recognize, choose to default to an HTTP `GET` request and respond as normal. We found this behavior only on a subset of HTTP servers that hosted censored domains outside our control, and we identified that nginx 1.10.3 responds to this query. The Apache and Nginx server versions we controlled did not respond to these requests with invalid methods.

None of the censors we tested could censor this strategy, including for both China's Host-based and keyword-based censorship. This suggests that the censors validate or require a valid HTTP Method before processing the rest of the request.

		Apache 2.4.X				Nginx 1.X.X				Country			
Family	**Strategy**	**6**	**18**	**29**	**43**	**13.4**	**14.1**	**16.1**	**19.0**	**CN-H**	**CN-K**	**IN**	**KZ**
Case Sensitivity	[HTTP:host:*]-changecase{lower}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-changecase{upper}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
Four Element Request Line	[HTTP:version:*]-insert{%09:middle:value:14}-|	✓	✓	-	-	-	-	-	-	✓	✓	✓	-
	[HTTP:path:*]-insert{%09:end:value:1434}-| [HTTP:path:*]-insert{1:start:value:507}-|	✓	✓	-	-	-	-	-	-	-	✓	✓	-
	[HTTP:path:*]-insert{%20:end:value:1}-| [HTTP:path:*]-insert{g:end:value:1013}-|	✓	✓	-	-	✓	✓	✓	✓	-	✓	✓	-
Host Header Shield	[HTTP:path:*]-insert{%20:start:value:1}-| [HTTP:host:*]-duplicate(replace{/:name:64} (replace{/?ultrasurf:value},),)-|	✓	✓	-	-	✓	✓	✓	✓	✓	✓	-	-
	[HTTP:host:*]-duplicate(replace{a:name:64},)-|	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	-	-
	[HTTP:method:*]-insert{%09:end:value}-| [HTTP:host:*]-duplicate(replace{a:name:64},)-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:method:*]-insert{%0A:start:value:1}-| [HTTP:host:*]-duplicate(replace{%2F:name:64},)-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:method:*]-insert{%20:end:value:1}-| [HTTP:host:*]-duplicate(replace{%2F:name:64},)-|	✓	✓	-	-	✓	✓	✓	✓	✓	✓	-	-
	[HTTP:path:*]-insert{%20:start:value:1}-| [HTTP:host:*]-duplicate(replace{%C2%B0:name:32},)-|	✓	✓	-	-	✓	✓	✓	✓	✓	✓	-	-
Host Header Whitespace	[HTTP:host:*]-duplicate(insert{%0A:end:value:1},)-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-duplicate(insert{%0A:random:name:1},)-|	-	-	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-duplicate(insert{%20%0A:end:name:1},)-|	-	-	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-insert{%09:end:name}-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:host:*]-insert{%09:end:value:1}-|	✓	✓	✓	✓	-	-	-	-	-	-	-	✓
	[HTTP:host:*]-insert{%09:start:value:1}-|	✓	✓	✓	✓	-	-	-	-	-	-	-	✓
	***[HTTP:host:*]-insert{%0A%0A:start:value:1}-|	-	-	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:host:*]-insert{%0A%20:start:value:1}-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:host:*]-insert{%0A:end:value:1}-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-insert{%20%0A:start:name:1}-|	-	-	-	-	✓	✓	✓	✓	✓	-	✓	✓
	[HTTP:host:*]-insert{%20:end:name:1}-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:host:*]-insert{%20:end:value:1}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	-	✓
	***[HTTP:host:*]-insert{%20:start:name:1}-|	-	-	-	-	-	-	-	-	✓	-	✓	✓
	***[HTTP:host:*]-insert{%20:start:value:2}-|	-	-	-	-	-	-	-	-	-	-	-	-
Long Request	[HTTP:path:*]-replace{/:value:1434}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-insert{%20:start:value:1413}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-insert{%20:start:value:1434}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:method:*]-duplicate(,replace{a:name:1407})-|	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	-
	[HTTP:method:*]-insert{%09:end:value:2568}-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:method:*]-insert{%0A:start:value:4336}-|	-	-	-	-	✓	✓	✓	✓	✓	✓	✓	✓
	[HTTP:method:*]-insert{%20:end:value:1413}-|	✓	✓	-	-	✓	✓	✓	✓	✓	-	✓	-
	[HTTP:method:*]-insert{%20:end:value:1720}-|	✓	✓	-	-	✓	✓	✓	✓	✓	-	✓	✓
	[HTTP:path:*]-duplicate(replace{a:name:1} (insert{a:start:value:1408},),)-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
	[HTTP:path:*]-insert{%0D:end:value:1434}-|	✓	✓	-	-	-	-	-	-	✓	✓	✓	-
	[HTTP:path:*]-insert{%20:end:value:1413}-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:path:*]-insert{%20:start:value:1}-| [HTTP:path:*]-replace{3:value:511} (insert{&:start:value},)-|	✓	✓	-	-	✓	✓	✓	✓	✓	✓	-	-
	[HTTP:path:*]-insert{%23:end:value:1413}-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:path:*]-insert{%23:end:value:1} (insert{%C3:end:value:470},)-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:path:*]-insert{%3F:end:value:1413}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
	[HTTP:path:*]-insert{%3F:start:value:1413}-|	✓	✓	✓	✓	-	-	-	-	✓	-	✓	-
	[HTTP:path:*]-replace{/:value:1414}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
	[HTTP:version:*]-insert{%20:end:value:1434}-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:version:*]-insert{%20:start:value:1434}-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:version:*]-insert{%25:middle:value:1434}-|	✓	✓	-	-	-	-	-	-	✓	✓	✓	✓
	[HTTP:version:*]-insert{%C2%81:end:value:773}-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:version:*]-insert{%C3%8B:middle:value:717}-|	✓	✓	-	-	-	-	-	-	✓	✓	✓	✓

Table 2: HTTP evasion strategies and where they succeed. A strategy is successful against a nation if it evades that nation's censor. A strategy is successful to a server if it evades in at least one country and is accepted by the server. CN-H and CN-K stand for the China Headers and China Keyword modes respectively. "***" denotes a strategy found against a live server we did not control; though these evade in some of our tested countries, but do not receive responses from the servers we tested. This table is continued in the Appendix in Table 4.

Strategy Family	Example Strategy	CF	OD	CB	CS	DW	Q9	V	G
Elevated Count	[DNSQR:qname:*]-tamper{DNS:qdcount:replace:2}-|	✓	-	-	-	-	-	-	-
	[DNSQR:qname:*]-tamper{DNS:ancount:replace:1}-|	✓	✓	-	-	-	-	-	-
Elevated Count w/ Reserved- and Truncated-bits	[DNS:*:*]-tamper{DNS:nscount:replace:1} (tamper{DNS:z:replace:1} (tamper{DNS:tc:replace:1},),)-|	✓	✓	✓	-	-	-	-	-
DNS Compression	[DNS:*:*]-tamper{DNS:qd:compress} (tamper{DNS:qdcount:replace:2},)-|	✓	-	-	-	-	-	-	✓
Multibyte Query Injection	[DNSQR:qname:*]-duplicate(,replace{%C2%91:name:957})-|	-	-	-	✓	✓	✓	✓	-
Multibyte Query Injection w/ Elevated Count	[DNSQR:qclass:]-tamper{DNS:ancount:replace:98}-| [DNSQR:qtype:]-replace{%C3%95:name:262}-|	✓	✓	-	-	-	-	-	-

Table 3: Summary of the five DNS strategy families we discover that defeat all three DNS injectors simultaneously, and which DNS resolvers respond to them: Cloudflare (CF), OpenDNS (OD), CleanBrowsing (CB), ComodoSecure (CS), DNS.Watch (DW), Quad9 (Q9), Verisign (V), and Google (G). Our system successfully identified strategies for every DNS resolver, and also identified four more unique variants to these strategies that only disabled a subset of the injectors.

6 DNS Results

We trained our system against all three of China's DNS injectors by using a domain that is on all three blocklists ("google.sm") to eight different open resolvers (see Table 1). In prior work, researchers identified that these different DNS injectors could be differentiated based on the fields set in the DNS responses. To avoid ambiguity, we will refer each of the three injectors using the same terminology as Anonymous et al. [8] and identify them by idiosyncratic fields they set in their response headers: Injector #1 (TTL=60, AA=1, DF=0), Injector #2 (AA=0, DF=1), and Injector #3 (AA=0, DF=0, IPID=0).

In total, we discovered 9 unique strategy types, 5 of which defeat all three injectors simultaneously. After our training runs, we performed manual analysis of the strategies to understand why they worked against each DNS injector. For each of the success rates below, we test each strategy 1000 times. See Table 3 for the full breakdown of results. Note that these strategies only apply to unencrypted DNS, as the header fields of encrypted DNS would not be visible to the adversary.

Elevated Count The simplest family of strategy types we discovered works by simply increasing the values of any combination of the count fields in the DNS request: `qdcount` (number of questions; default 1), `ancount` (number of answers; default 0), `arcount` (number of additional records; default 0), or `nscount` (number of name server resource records; default 0). Table 3 shows an example strategy in which the `qdcount` is set to 2, despite there being only a single query in the request, and another example that elevates the answer count to 1. All of these strategies are in violation of the RFC. Surprisingly, each of the GFW's injectors and open resolvers respond differently depending on which field we modify.

Elevating the `qdcount` field evades all three GFW injectors with 100% success rate, but only Cloudflare will respond to the query. Elevating the `ancount`, `arcount`, or `nscount` evade only DNS injectors 2 and 3. Cloudflare responds to all of these queries, OpenDNS responds only to elevated `ancount` and `nscount`, and none of the other resolvers responded to any of them.

Elevated Count with Reserved- and Truncated-bits The next strategy we discover works by increasing the `nscount` to 1 (which evades GFW injector #2 and #3), setting the reserved `z` field to 1, and setting the `tc` (truncated) bit to 1. The combination of the truncated field and reserved field both being set to 1 evades injector #1 with approximately 50% success rate. Therefore, if this strategy is used with a domain blocked by injector #2 or #3, it will evade with 100% reliability, but if the domain is also included on injector #1's blocklist, it will only evade with 50% reliability. Frankly, we do not understand the cause of why this strategy works only 50% of the time against injector #1.

DNS Compression The next strategy we discover works by performing DNS compression on the DNS query and then increasing the `qdcount` field to 2. DNS compression (defined by RFC 1035 [50]) works by splitting the DNS query across multiple records at the separator. This strategy is related to the Elevated Count strategies, but uses DNS compression to increase the number of DNS Question Records in the packet to actually be 2. Technically, since the domain is compressed across multiple DNS question records, the request has two DNS Question Records attached to it, even though they only comprise one DNS Question. This strategy evades all three DNS injectors with 100% reliability, but is only supported by Google and Cloudflare. We note that DNS compression alone does not evade censorship, it must be paired with the elevated `qdcount`.

Multibyte Query Injection The next strategy type we discover relies on injecting new text into the requests; specifically, it creates a second DNS Question Record *after* the forbidden query containing a request for a domain filled with 2-byte-wide multibyte UTF-8 characters. Surprisingly, all three of the GFW's injectors have problems handling requests that contain multibyte characters, but a different number of multibyte characters is required to evade each injector. Evading injector #1 requires at least 241 2-byte-wide multibyte characters; evading injector #3 requires at least 482 (precisely twice as many). Injector #2 can be evaded with a 36% success rate with 721 2-byte-wide multibyte characters; any fewer fails to evade. This success rate can be increased to 97% with at least 1,334 multibyte characters.

Interestingly, not all multibyte characters work: for all three injectors, only the characters within the range of `%C[2-F]%[80-BF]` succeed, and only 2-byte-wide characters work; 3-byte-wide characters do not.

Note that none of these requests are RFC compliant. According to RFC 1035 (Section 2.3.4), the limit to names is 255 bytes; in all the above cases, the DNS Question Record contains many more bytes than this. Different DNS resolvers have different policies as to if they respond to these queries. Quad9, Comodo, and DNS.Watch all respond to these queries normally, while Verisign responds only to 25% of the queries (we suspect this is due to load balancing between resolvers that may or may not be able to handle the queries). None of the other resolvers respond to these requests.

Multibyte Query Injection with Elevated Count Our system also identified a combination strategy of the above multibyte strategy and elevated `arcount`. This strategy creates a second DNS Question Record that contains 242 multibyte characters and sets the `arcount` field to 1. This strategy exemplifies how the different injectors can be defeated individually; by setting the `arcount` field, the strategy bypasses injector #2 and injector #3, and using 242 multibyte characters bypasses injector #1. Because this strategy injects fewer characters than the Multibyte Query Injection family, Cloudflare and OpenDNS now respond to the query, but Quad9, Comodo, and DNS.Watch will not respond, due to the elevated `arcount`.

Collectively, these results show that there is a large space of censorship evasion strategies possible through DNS query manipulation. The simplicity of some of these evasion strategies also indicates that this space has been largely unexplored; the fact that merely setting an incorrect `qdcount` works is surprising. On the other hand, the strange complexities of other strategies (such as requiring no less than 721 multibyte characters to evade Injector #2) justifies our approach of using automated tools to explore this space. Finally, taken in conjunction with our HTTP results, we see once again that servers that are *less* RFC-compliant than censoring middleboxes can lead to evasion opportunities.

7 Discussion

How can censors defend against these attacks? Censors could read this work and try to patch each individual issue we identify; however, we do not think censors will be able to easily (or cheaply) defend against all these attacks. Our results point to a broader trend about protocol compliance in censoring middleboxes. In order to effectively defend against these attacks, censors must always be more permissive in inputs they tolerate than servers on the other side of the connection. In cases where the censor was significantly more RFC-compliant (such as in India), our system had the easiest time discovering ways to evade censorship.

Even beyond censors needing to be more permissive than servers, to effectively censor, the censor must also maintain *at least* as much state as servers on the other side of the connection. If a server buffers more bytes than the censor does, a client can simply make the request longer until the forbidden keyword or header is outside the censors buffer, as we've seen in China. This is good news for evaders, as addressing this issue completely will likely require the censors to buffer vastly more data than they do currently. These trends hold across both HTTP and DNS.

What HTTP strategies work most often, and what do censors most commonly do wrong? The most common strategy we find by far is various forms of injecting whitespace, in both the headers and the request line. In fact, 53 of our 77 strategies work by inserting some form of whitespace, and 38 of which require no further modifications. The HTTP RFCs have many rules about where whitespace should be allowed, ignored, or disallowed, and we identified many cases in which the censor processes whitespace where it should not, or fails to process it where it should. Another common failure mode we observed from the censor was being unable to process a large request from a client, though each censor we studied was affected for a different reason.

What class of strategies are most broadly applicable across server versions and resolvers? For HTTP, we again find that inserting whitespace in different places around the request line or header value. The RFCs mention that certain types of whitespace should be ignored for robustness, so strategies that inject whitespace in these locations are most commonly versatile across server versions. We find that many of the server versions we tested often accept *too much* whitespace for robustness's sake, despite what the RFC says.

For DNS, we found little overlap between the queries accepted between the different resolvers. Our most broadly applicable strategies only worked on half of the resolvers we tested, and most worked across even less. In general, lack of generalizability for DNS strategies does not affect usability the same way for HTTP; if a user wishes to use our strategies to perform forbidden DNS lookups, the user can do all of those lookups to the same resolver. Over HTTP, by contrast, the evasion strategy must be compatible with the server on the other end of the connection, and every site the user visits may be using a different server version.

Is any one location in the HTTP or DNS header more prone to having viable evasion strategies? Overall, we found strategies for every major component of the HTTP request: 31 strategies acted on the Host header, 16 acted on the Method, 22 acted on the Path, and 13 acted on the Version.

Note that these numbers do not add to 77, as there is overlap in strategies that act on multiple parts of the request. In DNS, our strategies were also fairly well distributed throughout the DNS header, and only a few fields were never co-opted by a strategy for evasion.

How does China's Host header censorship compare to keyword censorship? In general, we find that almost all the strategies that evade keyword-based censorship in China also evade host-based censorship (17 out of 22). This interesting finding suggests that in order to correctly censor keywords, the GFW must be able to read the Host header, or read all the headers without problems and find no host header. Our results also suggest that the reverse is not true: no strategies that affected only the Host header were able to evade keyword-based censorship. We also find that more strategies can evade host-based censorship by simply injecting whitespace, compared to keyword censorship.

How do China's three DNS injectors compare to one another? We find differences between all three injectors that affects how well our strategies work. Injector #1 was the most permissive to fields being incorrect in the DNS header, and therefore had fewer strategies work; for example, Injector #1 still correctly processed forbidden DNS queries if the `arcount`, `ancount`, or the `nscount` fields were non-zero. Injector #2 had the most idiosyncratic responses to multibyte UTF characters: injecting between 721 and 1,333 multibyte characters caused Injector #2 to fail at least 33% of the time (and the failure rate increased as the number of inserted characters increased); after 1,334 characters, Injector #2 fails 100% of the time. Every strategy that evaded Injector #2 also evaded Injector #3, though we discover that Injector #3 has different limits to the number of multibyte characters it will tolerate in the DNS Query Records (a limit of 482). Overall, our results further emphasize that these injectors are truly separate, each with their own block list and weaknesses.

How generalizable is this technique to the future? We believe this technique should generalize well to other protocols. Many application-layer protocols fit the abstraction we defined for this paper (with smaller, discrete components that compose within a larger message). For example, TLS records are comprised of fixed static fields, and dynamic TLS Messages and TLS Extensions. We leave the implementation of this to future work.

8 Ethical Considerations

We designed our experiments to limit the potential impact to other hosts and the risk to real users. This work does not involve human subjects, and therefore falls outside the purview of our Institutional Review Board; still, we follow best practices laid out by prior censorship studies [14, 56].

We performed all of our training exclusively from vantage points we control, and our work does not require recruiting users (unwitting or not [16]). Our system does not spoof IP addresses or impersonate other machines, and our interactions with the censors should have had no impact on any other users. To limit the effect of our training on the network, we evaluate strategies serially (and with a small sleep for DNS), which limits how quickly our system can generate traffic. This is important, as some of our training runs involve hosts outside our control (such as with open DNS resolvers), and we believe our impact to these hosts is minimal. For example, our DNS training had a network load of approximately 11kbps, which should be a negligible volume of traffic for the size of the networks we test with. In training to hosts outside our control with HTTP, we set up our experiments to minimize potential harm to those hosts. We ran few experiments, spaced out in time, with slow query limit, and limited generations. We did not believe our fuzzing would cause a crash failure, as we had not observed any crashes in any of our prior experiments or in prior work that crafted strategies manually [44, 67].

Finally, we ask: does releasing this work help censors? We believe that, on balance, this work helps evaders more. Although individual bugs can be patched, the broader takeaways of this work (such as that application-layer censorship evasion can be automated or that RFC non-compliance can be leveraged for evasion) are still applicable. There is also strong precedent for developing automated techniques to evade censorship [14, 52, 60].

9 Conclusion

The censorship arms race has entered a fascinating new era of automated evasion. In this paper, we extend this to the application-layer space by presenting the first techniques to automate discovery of new censorship evasion strategies that require modifications *only* to application-layer requests. Training against China, India, and Kazakhstan, we discovered 77 unique strategies to evade HTTP censorship and 9 for DNS. We thoroughly analyzed each of these strategies and discovered that many of them are successful because censors often adhere *more* to protocol requirements than application servers do. Our tool—a modification of our prior work, Geneva [14]—exploits this discrepancies to alter queries in ways that censors reject but more-permissible servers accept. We believe this represents an interesting and important new domain for censorship evasion research. To assist in these efforts, we have made our code publicly available.

Acknowledgments We thank the anonymous reviewers and our shepherd, Paul Pearce, for their helpful feedback. This work was supported in part by NSF awards CNS-1901325 and CNS-1943240.

References

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

[2] Usage statistics of web servers, 2020. https://w3techs.com/technologies/overview/web_server.

[3] Web Server Usage Distribution in the Top 1 Million Sites, 2020. https://trends.builtwith.com/web-server.

[4] C. Agosti and G. Pellerano. SniffJoke: transparent TCP connection scrambler. https://github.com/vecna/sniffjoke, 2011.

[5] american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

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

[7] 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.

[8] Anonymous, A. A. Niaki, N. P. Hoang, P. Gill, and A. Houmansadr. Triplet Censors: Demystifying Great Firewall's DNS Censorship Behavior. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2020.

[9] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, 2005. https://www.rfc-editor.org/rfc/rfc3986.

[10] K. Bock, A. Alaraj, Y. Fax, K. Hurley, E. Wustrow, and D. Levin. Weaponizing Middleboxes for TCP Reflected Amplification. In *USENIX Annual Technical Conference*, 2021.

[11] K. Bock, P. Bharadwaj, J. Singh, and D. Levin. Your Censor is My Censor: Weaponizing Censorship Infrastructure for Availability Attacks. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2021.

[12] K. Bock, Y. Fax, K. Reese, J. Singh, and D. Levin. Detecting and Evading Censorship-in-Depth: A Case Study of Iran's Protocol Whitelister. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2020.

[13] K. Bock, G. Hughey, L.-H. Merino, T. Arya, D. Liscinsky, R. Pogosian, and D. Levin. Come as You Are: Helping Unmodified Clients Bypass Censorship with Server-Side Evasion. In *ACM SIGCOMM*, 2020.

[14] K. Bock, G. Hughey, X. Qiang, and D. Levin. Geneva: Evolving Censorship Evasion Strategies. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[15] K. Bock, iyouport, Anonymous, L.-H. Merino, D. Fifield, A. Houmansadr, and D. Levin. Exposing and Circumventing China's Censorship of ESNI. https://geneva.cs.umd.edu/posts/china-censors-esni/esni/, 2020.

[16] S. Burnett and N. Feamster. Encore: Lightweight Measurement of Web Censorship with Cross-Origin Requests. In *ACM SIGCOMM*, 2015.

[17] Chromium Development Team. A safer default for navigation: HTTPS. https://blog.chromium.org/2021/03/a-safer-default-for-navigation-https.html, 2020.

[18] CitizenLab. CitizenLab Test Lists. https://github.com/citizenlab/test-lists, 2020.

[19] CitizenLab. URL testing lists intended for discovering website censorship. https://github.com/citizenlab/test-lists/, 2022.

[20] R. Clayton, S. J. Murdoch, and R. N. M. Watson. Ignoring the Great Firewall of China. In *Privacy Enhancing Technologies Symposium (PETS)*, 2006.

[21] Cloudflare. Cloudflare Radar: Up to date Internet trends and insight. https://radar.cloudflare.com/cn?date_filter=last_30_days, 2022.

[22] Congressional Research Service. Social Media: Misinformation and Content Moderation Issues for Congress, 2021. https://crsreports.congress.gov/product/pdf/R/R46662.

[23] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, 2004.

[24] D. Ellard, C. Jones, V. Manfredi, W. T. Strayer, B. Thapa, M. V. Welie, and A. Jackson. Rebound: Decoy routing on asymmetric routes via error messages. In *IEEE Conference on Local Computer Networks (LCN)*, 2015.

[25] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz. Measuring HTTPS Adoption on the Web. In *USENIX Security Symposium*, 2017.

[26] R. Fielding, J. Gettys, J. Mogul, H. Frysyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, 1999. https://datatracker.ietf.org/doc/html/rfc2616.

[27] R. Fielding, Y. Lafon, and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Range Requests. RFC 7233, 2014. https://www.rfc-editor.org/rfc/rfc7233.html.

[28] R. Fielding, M. Nottingham, and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Caching. RFC 7234, 2014. https://www.rfc-editor.org/rfc/rfc7234.html.

[29] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Authentication. RFC 7235, 2014. https://www.rfc-editor.org/rfc/rfc7235.html.

[30] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests. RFC 7232, 2014. https://www.rfc-editor.org/rfc/rfc7232.html.

[31] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, 2014. https://www.rfc-editor.org/rfc/rfc7230.html.

[32] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, 2014. https://www.rfc-editor.org/rfc/rfc7231.html.

[33] D. Fifield. Threat modeling and circumvention of Internet censorship. In *PhD thesis*, 2017.

[34] D. Fifield, N. Hardison, J. Ellithorpe, E. Stark, D. Boneh, R. Dingledine, and P. Porras. Evading Censorship with Browser-Based Proxies. In *Privacy Enhancing Technologies Symposium (PETS)*, 2012.

[35] D. Fifield, C. Lan, R. Hynes, P. Wegmann, and V. Paxson. Blocking-resistant communication through domain fronting. In *Privacy Enhancing Technologies Symposium (PETS)*, 2015.

[36] P. Foremski. Tracking the DNS Stars: The DNS Observatory, 2019. https://www.farsightsecurity.com/blog/txt-record/dnsstars-20190610/.

[37] S. García, K. Hynek, D. Vekshin, T. Čejka, and A. Wasicek. Large Scale Measurement on the Adoption of Encrypted DNS. In *Passive and Active Network Measurement Workshop (PAM)*, 2021.

[38] Gitlab. Gitlab Protocol Fuzzer Community Edition, 2021. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce.

[39] L. Haifeng, W. Shaolei, Z. Bin, S. Bo, and T. Chaojing. Network protocol security testing based on fuzz. In *International Conference on Computer Science and Network Technology (ICCSNT)*, 2015.

[40] C. Hornig. A Standard for the Transmission of IP Datagrams over Ethernet Networks. RFC 894, 1984. https://datatracker.ietf.org/doc/html/rfc894.

[41] A. Houmansadr, G. T. K. Nguyen, M. Caesar, and N. Borisov. Cirripede: Circumvention Infrastructure using Router Redirection with Plausible Deniability. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[42] A. Houmansadr, T. Riedl, N. Borisov, and A. Singer. IP over Voice-over-IP for censorship circumvention. In *arXiv preprint arXiv:1207.2683*, 2012.

[43] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda. T-Reqs: HTTP Request Smuggling with Differential Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[44] S. Khattak, M. Javed, P. D. Anderson, and V. Paxson. Towards Illuminating a Censorship Monitor's Model to Facilitate Evasion. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[45] G. T. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[46] D. Levin, Y. Lee, L. Valenta, Z. Li, V. Lai, C. Lumenzanu, N. Spring, and B. Bhattacharjee. Alibi Routing. In *ACM SIGCOMM*, 2015.

[47] F. Li, A. Razaghpanah, A. M. Kakhki, A. A. Niaki, D. Choffnes, P. Gill, and A. Mislove. lib.erate, (n): A library for exposing (traffic-classification) rules and avoiding them efficiently. In *ACM Internet Measurement Conference (IMC)*, 2017.

[48] Z. Li, S. Herwig, and D. Levin. DeTor: Provably Avoiding Geographic Regions in Tor. In *USENIX Security Symposium*, 2017.

[49] X. Mendez. WFuzz: The Web Fuzzer, 2020. wfuzz.io.

[50] P. Mockapetris. Domain names - implementation and specification. RFC 1035, 1987. https://datatracker.ietf.org/doc/html/rfc1035.

[51] H. M. Moghaddam, B. Li, M. Derakhshani, and I. Goldberg. SkypeMorph: Protocol Obfuscation for Tor Bridges. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[52] S.-J. Moon, J. Helt, Y. Yuan, Y. Bieri, S. Banerjee, V. Sekar, W. Wu, M. Yannakakis, and Y. Zhang. Alembic: Automated Model Inference for Stateful Network Functions. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[53] R. S. Raman, L. Evdokimov, E. Wustrow, A. Halderman, and R. Ensafi. Kazakhstan's HTTPS Interception. https://censoredplanet.org/kazakhstan, 2019.

[54] R. S. Raman, L. Evdokimov, E. Wustrow, A. Halderman, and R. Ensafi. Investigating Large Scale HTTPS Interception in Kazakhstan. In *ACM Internet Measurement Conference (IMC)*, 2020.

[55] S. M. Seal. Optimizing Web Application Fuzzing with Genetic Algorithms and Language Theory. In *Master of Science Thesis*, 2016.

[56] B. VanderSloot, A. McDonald, W. Scott, J. A. Halderman, and R. Ensafi. Quack: Scalable Remote Measurement of Application-Layer Censorship. In *USENIX Security Symposium*, 2018.

[57] P. Vines and T. Kohno. Rook: Using Video Games as a Low-Bandwidth Censorship Resistant Communication Platform. In *Workshop on Privacy in the Electronic Society (WPES)*, 2015.

[58] Q. Wang, X. Gong, G. T. Nguyen, A. Houmansadr, and N. Borisov. CensorSpoofer: Asymmetric communication using IP Spoofing for Censorship-resistant Web Browsing. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[59] Z. Wang, Y. Cao, Z. Qian, C. Song, and S. V. Krishnamurthy. Your State is Not Mine: A Closer Look at Evading Stateful Internet Censorship. In *ACM Internet Measurement Conference (IMC)*, 2017.

[60] Z. Wang, S. Zhu, Y. Cao, Z. Qian, C. Song, S. V. Krishnamurthy, K. S. Chan, and T. D. Braun. SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[61] Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, S. Cheung, F. Wang, and D. Boneh. StegoTorus: A Camouflage Proxy for the Tor Anonymity System. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[62] B. Wiley. Dust: A Blocking-Resistant Internet Transport Protocol. http://blanu.net/Dust.pdf.

[63] P. Winter. brdgrd (Bridge Guard). https://github.com/NullHypothesis/brdgrd, 2012.

[64] wkrp. HTTPS MITM of various GitHub IP addresses in China. https://github.com/net4people/bbs/issues/27, 2020.

[65] E. Wustrow, C. M. Swanson, and J. A. Halderman. TapDance: End-to-Middle Anticensorship without Flow Blocking. In *USENIX Annual Technical Conference*, 2014.

[66] E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman. Telex: Anticensorship in the Network Infrastructure. In *USENIX Security Symposium*, 2011.

[67] T. K. Yadav, A. Sinha, D. Gosain, P. K. Sharma, and S. Chakravarty. Where The Light Gets In: Analyzing Web Censorship Mechanisms in India. In *ACM Internet Measurement Conference (IMC)*, 2018.

[68] W. Zhou, A. Houmansadr, M. Caesar, and N. Borisov. SWEET: Serving the Web by Exploiting Email Tunnels. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

A HTTP Geneva Syntax

We give a brief background of Geneva's syntax for strategies. Strategies are comprised of trigger/action tree pairs: the trigger defines which packet component should be modified, and the action-tree specifies how it should be modified. Action-trees are trees composed of simple manipulation actions.

Actions Depending on the protocol (DNS or HTTP), the principle unit of modification is different: HTTP operates over the Headers, and DNS operations over the DNS Question Records. Each action defines specific parameters it accepts; below is an overview of the arguments the actions can take.

1. `<value>` — Any printable characters (URL-encoded).
2. `<string location>` — Where in a string to be inserted: `start`, `middle`, `end`, or `random`. Start means at index 0, middle is at index length/2, end is equal to index length, and random is anywhere except the start or end.
3. `<component>` — (Only used for HTTP Geneva.) Specifies which part of a header to act upon: "name", or "value". Remember a header is broken up into two sides separated by the semi-colon into the header name and header value. Ex: in “Host: www.example.com”, “Host” is the header name and “ www.example.com” is the header value (note the space is included).
4. `<number of actions>` — How many times the action should be run. For example, with the `insert` or `replace` actions, running this action multiple times concatenates the `<value>` `<number of actions>` number of times before doing the action.
5. `<case method>` — Either random (each character is randomly upper or lower case), lower (all characters are lower case) or upper (all characters are upper case).

With these arguments in mind, below are Geneva's modification actions for HTTP and DNS.

1. `insert{<value>:<string location>:`
   `<component>:<number of actions>}`

   Insert the byte(s) specified by `value` into the location of the string specified `<string location>` into the specified `<component>` of the action target `<number of actions>` times. This will insert either once (if the fourth parameter is omitted) or a specified number of times if the fourth parameter is given.
2. `replace{<value>:<component>:`
   `<number of actions>}`

   Replace the field specified by `<component>` with `<value>`. If `<number of actions>` is given, replace it with that many copies of `<value>` (default: one). Note: delete can be simulated here by the random chance to replace the "value" with nothing.
3. `duplicate(,)` — Makes a second action target equal to the first. Duplicate outputs two identical action targets and each side can be modified individually. Duplicating a header will add a new header, not just concatenate the string of the header name or value.
4. `drop(,)` — Remove the action target from the request.
5. `changecase{<case method>}` — Changes the case of the entire action target (ignores non-alphabetical characters). If this is a header, this works on the header name and value.

Action trees can be extended by adding new actions into children (ending parenthesis) of any action. For every action except `duplicate`, there will only be downstream actions in the first half of the parenthesis.

Trigger There is one matching trigger for every action tree and it signifies when that action tree should act on an action target. Each trigger takes three parameters. The first element of the trigger is the relevant protocol: DNS, DNSQR, or HTTP. The second element signifies which field the trigger should check for a match. For DNS, this trigger will look for certain fields like `qclass`, whereas in HTTP, it uses specific header names. The third element specifies what the target field must be for the trigger to fire. A star “*” can be used as a wildcard. An example of a trigger is `[HTTP:Host:*]` to mean a strategy will act on any Host header it sees, or `[DNSQR:qname:*]` to act on every DNS Question Record.

Strategy Syntax Action trees are combined with triggers to create combinations like `<trigger>-<action tree>-|`. It is possible to have multiple combinations in one strategy. When this happens, each action tree will act on their own version of their designated action target, and all the duplicates will be combined in the end to recreate the request. An example of a full strategy with two action trees is:

```
[HTTP:Host:*]-duplicate(
    replace{NewValue:name:1},
        insert{\%20:start:value:500}
            (changecase{random},))-|
[HTTP:Host:*]-insert{\%0A:start:name}-|}}
```

		Apache 2.4.X				Nginx 1.X.X				Country			
Family	**Strategy**	**6**	**18**	**29**	**43**	**13.4**	**14.1**	**16.1**	**19.0**	**CN-H**	**CN-K**	**IN**	**KZ**
Method Mangling	***[HTTP:method:*]-duplicate(,)-|	-	-	-	-	-	-	-	-	-	-	✓	✓
	***[HTTP:method:*]-replace{%3A:value:1}-|	-	-	-	-	-	-	-	-	✓	✓	✓	✓
	***[HTTP:method:*]-replace{HTTP/1.1:value:1}-|	-	-	-	-	-	-	-	-	✓	✓	✓	✓
Path Confusion	[HTTP:path:*]-duplicate(insert{3:middle:value:1004}, replace{&ultrasurf:value})-|	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	-
	[HTTP:path:*]-insert{%3F:start:value:1}-|	✓	✓	✓	✓	-	-	-	-	✓	-	✓	-
Request Line Whitespace	[HTTP:method:*]-insert{%09:end:value:1}-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	***[HTTP:method:*]-insert{%09:start:value:1}-|	-	-	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:method:*]-insert{%0A:start:value:1}-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:method:*]-insert{%0B:end:value:1}-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:method:*]-insert{%0D:end:value:2}-|	✓	✓	-	-	-	-	-	-	✓	✓	✓	✓
	[HTTP:path:*]-insert{%09:end:value:1}-|	✓	✓	-	-	-	-	-	-	-	-	✓	-
	[HTTP:path:*]-insert{%09:start:value:1}-|	✓	✓	-	-	-	-	-	-	✓	-	✓	-
	[HTTP:path:*]-insert{%0C:start:value:1}-|	✓	✓	-	-	-	-	-	-	✓	-	✓	-
	[HTTP:path:*]-insert{%0D:start:value:1}-|	✓	✓	-	-	-	-	-	-	✓	✓	✓	-
	[HTTP:path:*]-insert{%20:end:value:1}-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:path:*]-insert{%20:start:value:1}-|	-	-	-	-	-	-	-	-	✓	-	-	-
	[HTTP:version:*]-insert{%0A%09%0A%09:end:value:1}-|	-	-	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:version:*]-insert{%0A%09:end:value:1}-|	-	-	-	-	✓	✓	✓	✓	-	-	-	✓
	[HTTP:version:*]-insert{%0A%20%0A%20:end:value:1}-|	-	-	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:version:*]-insert{%20%0A%09:end:value:1}-|	-	-	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:version:*]-insert{%20:end:value:1}-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
Sandwich Strategy	[HTTP:host:*]-duplicate(replace{%C3%97:name:596}, insert{%20:end:name:786})-|	✓	✓	-	-	-	-	-	-	✓	✓	✓	✓
	[HTTP:host:*]-replace{%5E:name:926} (duplicate(duplicate(,replace{host:name:1} (insert{%20:start:value:3238},)),),)-|	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	✓
	[HTTP:host:*]-replace{%C3%97:name:1358} (duplicate(duplicate(,replace{host:name:1} (insert{%20:end:value},)),),)-|	✓	✓	-	-	✓	✓	✓	✓	✓	✓	✓	✓
	[HTTP:host:*]-replace{%C3%97:name:1371} (duplicate(duplicate(,replace{host:name:1}),),)-|	✓	✓	-	-	✓	✓	✓	✓	✓	✓	✓	-
	[HTTP:host:*]-insert{%20:end:value:4081} (duplicate(duplicate(,replace{a:name:1}), insert{%09:start:name:3238}),)-|	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	-	✓
	[HTTP:host:*]-insert{%20:end:value:4081} (duplicate(duplicate(insert{%09:start:name:3238},), replace{a:name:1}),)-|	✓	✓	-	-	✓	✓	✓	✓	-	✓	-	✓
	[HTTP:host:*]-replace{PUT:name:423} (duplicate(duplicate(,replace{host:name}),),)-|	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-
Version Mangling	[HTTP:version:*]-duplicate-|	✓	✓	-	-	-	-	-	-	-	-	✓	-
	[HTTP:version:*]-replace{OPTIONS:value:1}-|	✓	✓	-	-	-	-	-	-	✓	✓	✓	-

Table 4: Continuation of Table 2. A strategy is successful against a nation if it evades that nation's censor. A strategy is successful to a server if it evades in at least one country and is accepted by the server. CN-H and CN-K stand for the China Headers and China Keyword modes respectively. "***" denotes a strategy found against a live server we did not control; though these evade in some of our tested countries, but do not receive responses from the servers we tested.