ABSTRACT

Title of Dissertation: AUTOMATING THE DISCOVERY OF CENSORSHIP EVASION STRATEGIES

Kevin Bock
Doctor of Philosophy, 2022

Dissertation Directed by: Professor Dave Levin
Department of Computer Science

Censoring nation-states deploy complex network infrastructure to regulate what content citizens can access, and such restrictions to open sharing of information threaten the freedoms of billions of users worldwide, especially marginalized groups. Researchers and censoring regimes have long engaged in a cat-and-mouse game, leading to increasingly sophisticated Internet-scale censorship techniques and methods to evade them. In this dissertation, I study the technology that underpins this Internet censorship: middleboxes (e.g., firewalls). I argue the following thesis: *It is possible to automatically discover packet sequence modifications that render deployed censorship middleboxes ineffective across multiple application-layer protocols.*

To evaluate this thesis, I develop Geneva, a novel genetic algorithm that automatically discovers packet-manipulation-based censorship evasion strategies against nation-state level censors. Training directly against a live adversary, Geneva composes, mutates, and evolves sophisticated strategies out of four basic packet manipulation primitives (drop, tamper, duplicate, and fragment).

I show that Geneva can be effective across different application layer protocols (HTTP, HTTPS+SNI, HTTPS+ESNI, DNS, SMTP, FTP), censoring regimes (China, Iran, India, and Kazakhstan), and deployment contexts (client-side, server-side), even in cases where multiple middleboxes work in parallel to perform censorship. In total, I present 112 client-side strategies (85 of which work by modifying application layer data), and the first ever server-side strategies (11 in total). Finally, I use Geneva to discover two novel attacks that show that censoring middleboxes can be *weaponized* to launch attacks against innocent hosts anywhere on the Internet.

Collectively, my work shows that censorship evasion can be automated and that censorship infrastructures pose a greater threat to Internet availability than previously understood.

AUTOMATING THE DISCOVERY OF CENSORSHIP EVASION STRATEGIES

by

Kevin Bock

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2022

Advisory Committee:
Professor Dave Levin, Chair/Advisor (University of Maryland)
Professor Bobby Bhattacharjee (University of Maryland)
Professor Eric Wustrow (University of Colorado, Boulder)
Professor Michel Cukier (University of Maryland)
Professor John Dickerson (University of Maryland)

© Copyright by
Kevin Bock
2022

Acknowledgments

I first want to thank my advisor, Dave Levin. Dave's endless support extended beyond just research, and I am a better student, writer, presenter, researcher, runner, mentor—and most important of all, a better person—from having worked with him over these years. It is Dave's enthusiasm for research and making a difference that got me into graduate school, and his mentorship that got me through. It is well-known within the lab that Dave's top priority is his students and bringing out the best of those students: Dave supported me through many late nights, early mornings, and everything in between, and I will be forever grateful of his generosity with his time and energy spanning years.

I want to thank the team of students that worked with me over the years in the Breakerspace Lab. George Hughey was the first student to join the project, and I will always appreciate his initial leap of faith and support that got this project off the ground. Louis-Henri Merino worked with me across multiple projects, multiple degrees, and multiple academic institutions, and his unwavering support and dedication to the project was a constant source of energy for me. In roughly chronological order, thank you to all the students that contributed to the many various projects under the Geneva umbrella: George Hughey, Louis-Henri Merino, Tania Arya, Daniel Liscinsky, Regina Pogosian, Gabriel Naval, Kyle Reese, Yair Fax, Pranav Bharadwaj, Jasraj Singh, Nathan Stiff, Sadena Rishindran, Quinton Davidson, Alden Schmidt, Michael Harrity, Kyle Hurley, Freddy Sell, Brendan Mcmahan, Amanda Li, Josephine Chow, Katie Sullivan, Melissa Hoff, Sadia Nourin, Aaron

Ortwein, and the other students who chose not to be named here. I have grown tremendously from having worked with you all, and each of you had a meaningful impact on the project.

I also want to thank my many collaborators and those who have helped me. At Colorado Boulder, Eric Wustrow and Abdul Alaraj were close collaborators multiple projects (including some of the work that comprises this thesis), and Eric served on my proposal and defense committees. Thank you both for all of your time, energy, and insights: I am a better researcher and person from having worked with you both. At Berkeley, I thank Xiao Qiang for lending early support to the Geneva project: without your help, Geneva may never have left the lab. I thank Neil Spring for serving on my proposal committee, and thank John Dickerson for serving on both my proposal and dissertation committee; both of you provided valuable feedback and support during the process that helped my work grow. I thank Bobby Bhattacharjee for serving on my proposal and dissertation committees and for giving me the tough feedback that I needed: your tough love helped me grow as a researcher.

I thank the Open Technology Fund, whose early support and enthusiasm helped get this work off the ground. I also thank the OONI community for their support and for the community they have built. There are many other activists and researchers that contributed their time, networks, and expertise to the project whom I cannot thank by name here: thank you.

I next want to thank Michel Cukier and the ACES staff. Michel Cukier is the director of the Advanced Cybersecurity Exeprience for Students (ACES) program on campus, and welcomed me onto his research team early in my undergraduate

career. Bertrand Sobesto led the research project I was working on, and took me under his wing for multiple semesters of research. I credit Michel and Bertrand for first igniting my interest in research, and giving me the space to explore and grow my research skills as an undergraduate. The entire ACES staff (Michel Cukier, Jan Plane, Liz Rogers, Bertrand Sobesto, and the many other assistants during my time in the program) helped to curate and build a solid foundation for me to launch my academic career: thank you.

I must also thank the CS department for their support throughout my doctorate. First, I thank Tom Hurst in the graduate advising office for his endless patience, kind support, and *legendary* email response time. Tom was available and supportive for hours of questions even before I became a graduate student, and it is his patience and investment in me that helped make me comfortable to first take the plunge into graduate school. Throughout my degree, Tom handled more questions, policy edge cases, form submissions, and other academic concerns than I can possibly count, and did it all with a smile. Thank you, Tom. I thank Sharron McElroy (and the entire Purchasing team) for the tremendous behind the scenes work keeping our infrastructure up, available, and running smoothly: without you, the project would not have been possible. I also thank the broader team of personnel within the department that helped me throughout the process.

I also have a significant support network outside of school that helped me along the way. Ashton Webster, Daven Patel, and Ryan Eckenrod read early drafts and gave feedback for every major research paper of my academic career. Baldwin Mei, Chris Fu, Nick Cataldo, Brian Gross, Caroline Juang, Alex Comerford and

more helped to review and give feedback on multiple papers that comprised this dissertation. Brian Bock helped review papers, articles, patiently listened to dozens of technical discussions, and even contributed graphics to the project website. Thank you all for your help, and for keeping me grounded and sane over the years!

I want to extend my sincere thanks to my family. My parents have always been incredible role models for me, and my siblings and entire huge family has been the most supportive squad I could have asked for. My grandparents, Nana and Papa, were also important role models and an amazing component of my support network. Thank you all for your endless support and enthusiasm!

Lastly, I want to thank my wife and life-long supporter Sydnee, who has been endlessly supportive of my graduate pursuits, despite many long days and nights. You have been my fiercest defender, strongest supporter, a patient sounding board through more technical discussions than I can count. I love you all with all my heart.

Grants This dissertation was supported in part by the Open Technology Fund and NSF grants CNS-1816802 and CNS-1943240.

Collaborations This dissertation involved collaborative efforts with the following people:

- *Chapter 3*: My co-authors are George Hughey, Xiao Qiang, and Dave Levin, and this work appeared in ACM CCS in 2019 [1]. I would also like to thank Ramakrishna Padmanabhan, Neil Spring, the Breakerspace lab, and the anonymous reviewers for their helpful feedback.

- *Chapter 4*: My co-authors are George Hughey, Louis-Henri Merino, Tania Arya, Daniel Liscinsky, Regina Pogosian, Dave Levin, and this work appeared in ACM SIGCOMM in 2020 [2]. I would also like to thank my collaborators from the OTF and OONI communities, who have contributed insights and resources that made this work possible, and the anonymous reviewers for their helpful feedback.
- *Chapter 5*: My co-authors are Michael Harrity, Freddy Sell, and Dave Levin, and this work appeared in USENIX Security 2022. I would also like to thank our shepherd Paul Pierce, the anonymous reviewers, David Fifield, my collaborators from the OTF and OONI communities, as well as the University of Maryland UMIACS IT Staff, who contributed insights and resources that made this work possible.
- *Chapter 6*: My co-authors are Yair Fax, Kyle Reese, Jasraj Singh, and Dave Levin, and this work appeared in USENIX FOCI in 2020 [3]. I would also like to thank our shepherd David Fifield and the anonymous reviewers for their helpful feedback. I also thank the OTF and OONI communities who have contributed insights and resources that made this work possible.
- *Chapter 7*: My co-authors are Gabriel Naval, Kyle Reese, and Dave Levin, and this work appeared in SIGCOMM FOCI in 2021 [4]. I would also like to thank the anonymous reviewers and our shepherd, Rob Jansen, for their helpful feedback.
- *Chapter 8*: My co-authors are Abdulrahman Alaraj, Yair Fax, Kyle Hurley, Eric Wustrow, and Dave Levin, and this work appeared in USENIX Security in 2021 [5]. I would also like to thank network infrastructure team at the University of Col-

orado Boulder for supporting our scanning efforts and providing the resources that made this work possible. I also thank the anonymous reviewers for their helpful feedback. Finally, I thank our collaborators from the OTF and OONI communities for contributing resources that enabled this work.

- *Chapter 9*: My co-authors are Pranav Bharadwaj, Jasraj Singh, Dave Levin, and this work appeared in USENIX WOOT in 2021 [6]. I would also like to thank our shepherd Kevin Borgolte and the anonymous reviewers for their helpful feedback; Will Scott for his support with SP^3; and our collaborators from the OTF and OONI communities for contributing insights and resources that made this work possible. Also, I thank the anonymous Artifact Evaluators for their diligent efforts.

To acknowledge the many collaborators and supporters that contributed to this work, I will use the word “we” within many chapters.

Table of Contents

Acknowledgements **ii**

Table of Contents **viii**

List of Tables **xii**

List of Figures **xv**

1 Introduction **1**
- 1.1 Thesis . . . 2
- 1.2 Contributions . . . 4
- 1.3 Ethical Considerations . . . 6
- 1.4 Roadmap . . . 6

2 Background and Threat Model **10**
- 2.1 Nation-state Censors: Threat Model . . . 10
- 2.2 Related Work: Measuring Censors . . . 13
- 2.3 Evasion via Packet Manipulation . . . 14
- 2.4 Automating Censorship Evasion . . . 17
- 2.5 Fuzzing . . . 19

3 Discovering Client-side Evasion Strategies with Geneva **21**
- 3.1 Geneva Design . . . 21
  - 3.1.1 Overview and Challenges . . . 22
  - 3.1.2 Geneva's Genetic Building Blocks . . . 23
  - 3.1.3 Evolution . . . 27
  - 3.1.4 Implementation . . . 32
- 3.2 Validation . . . 33
- 3.3 Evaluation against real censors . . . 37
  - 3.3.1 Experiment Setup . . . 37
  - 3.3.2 China: The Great Firewall . . . 39
  - 3.3.3 Other Countries . . . 49
  - 3.3.4 Training Defunct Strategies . . . 52
- 3.4 Discussion . . . 53
- 3.5 Conclusion . . . 56

4 Server-side Evasion **60**
4.1 Client-Side Strategies do not Generalize . . . 63
4.2 Server-side Methodology . . . 65
4.2.1 Geneva Extensions . . . 66
4.2.2 Data Collection Methodology . . . 67
4.3 Server-Side Results . . . 70
4.3.1 Server-side Evasion in China . . . 70
4.3.2 Server-side Evasion in India & Iran . . . 81
4.3.3 Server-side Evasion in Kazakhstan . . . 82
4.4 Multiple Censorship Boxes . . . 87
4.5 Client Compatibility . . . 89
4.6 Deployment Considerations . . . 91
4.7 Ethical Considerations . . . 92
4.8 Conclusion . . . 93

5 Application-Layer Evasion **95**
5.1 Application-Layer Censorship Background . . . 98
5.2 Fuzzer Design . . . 100
5.2.1 Grammars . . . 103
5.2.2 Manipulations . . . 104
5.2.3 Fitness Function . . . 106
5.2.4 Using Strategies . . . 109
5.3 Methodology . . . 109
5.4 HTTP Results . . . 115
5.4.1 Summary Results . . . 115
5.4.2 Evasion Strategies . . . 116
5.4.3 External Validation . . . 124
5.5 DNS Results . . . 125
5.6 Discussion . . . 131
5.7 Ethical Considerations . . . 135
5.8 Conclusion . . . 135

6 Censorship-in-Depth: Iran **137**
6.1 Iranian Censorship Background . . . 139
6.2 Methodology . . . 140
6.3 Protocol Filter . . . 140
6.3.1 How Iran's Protocol Filter Works . . . 141
6.3.2 Whom the Filter Is Applied To . . . 144
6.3.3 Protocol Fingerprints . . . 147
6.4 Evading the Protocol Filter . . . 150
6.4.1 Old Strategies Do Not Apply . . . 150
6.4.2 Evolving New Strategies . . . 151
6.4.3 Discovered Evasion Strategies . . . 153
6.5 Conclusion . . . 156

7 Censorship-in-Depth: China's SNI Censorship 157
7.1 Methodology . . . 160
7.2 Evasion . . . 165
7.2.1 MB-RA Evasion Strategies . . . 165
7.2.2 Evading MB-RA and MB-R . . . 168
7.3 How does MB-R work? . . . 169
7.4 Ethical Considerations . . . 174
7.5 Conclusion . . . 174

8 Weaponizing Censors for Amplification Attacks 176
8.1 Background . . . 180
8.2 Discovering TCP-based Reflection Attacks . . . 184
8.2.1 Automated Discovery of Amplification . . . 184
8.2.2 Training Methodology . . . 186
8.2.3 Discovered Amplification Attacks . . . 187
8.2.3.1 Amplifying Packet Sequences . . . 188
8.2.3.2 Packet Sequence Modifications . . . 191
8.3 Internet Scanning Methodology . . . 195
8.4 Internet Scanning Results . . . 197
8.4.1 Which strategies work best? . . . 198
8.4.2 Are these actually amplifiers? . . . 201
8.4.3 Are these middleboxes? . . . 202
8.4.4 What kind of packets do amplifiers send? . . . 205
8.4.5 Are these national firewalls? . . . 206
8.4.6 Routing Loops . . . 209
8.5 "Mega-amplifiers" . . . 212
8.6 Ethical Considerations . . . 216
8.7 Countermeasures . . . 218
8.7.1 Middleboxes . . . 218
8.7.2 End Hosts . . . 220
8.8 Conclusion . . . 220

9 Weaponizing Censors for Availability Attacks 222
9.1 Background & Related Work . . . 225
9.2 Measurement Methodology . . . 228
9.3 State of Residual Censorship . . . 230
9.4 Residual Censorship Attack . . . 239
9.4.1 Launching the Attack . . . 239
9.4.2 Results . . . 242
9.5 Attack Impact . . . 249
9.6 Mitigations . . . 252
9.6.1 Censors . . . 252
9.6.2 Potential Victims . . . 253
9.7 Ethical Considerations . . . 254
9.8 Conclusion . . . 255

10 Defending Against Geneva **256**
10.1 What would it take to defend against Geneva? 256
10.2 Does Geneva help the censor? 260

11 Conclusion and Future Work **262**
11.1 Immediate Term Challenges 262
11.2 Long Term Challenges 265

Bibliography **268**

List of Tables

3.1 Species, subspecies, and variants Geneva found (with success rates) against the GFW. For readability, we omit all "`send`"s from the genetic code (e.g., `duplicate(,)` is equivalent to `duplicate(send,send)`). This is correct, syntactic sugar for Geneva. 57
3.2 Mock censors developed for in-lab training, and strategies Geneva learned to defeat them. 58
3.3 Prior work's effective TCP-based strategies and whether Geneva rederived the strategy in the lab or in the wild, regardless of whether the strategy is still effective. Note that Geneva had no knowledge of HTTP fields and could not introduce delays into the request. 59

4.1 Client locations and protocols used in our experiments. 67
4.2 Summary of server-side-only strategies and their success rates. All of these strategies manipulate only TCP, and yet, against China's GFW, their success rates are application-dependent. Kazakhstan's HTTPS and Iran's DNS-over-TCP censorship infrastructure are currently inactive. 71

5.1 DNS Open Resolvers we conduct experiments with. All of these open resolvers are accessible from within China. 110
5.2 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 i Table 5.3. 126

5.3 Continuation of Table 5.2. 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. 127
5.4 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), Google (G), Quad9 (Q9), OpenDNS (OD), CleanBrowsing (CB), ComodoSecure (CS), Verisign (V), and DNS.Watch (DW). 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. 128

6.1 Top 10 providers for *affected* IP addresses. 144
6.2 Top 10 providers for *unaffected* IP addresses 145

8.1 TCP-based reflected amplification attacks discovered against 184 Quack servers. Each packet with the `PSH` flag set includes an offending HTTP GET request in the payload. 188
8.2 Total data received (GB) from the top 100,000 IP addresses for each combination of target URL and packet sequence. Bolded is the maximum value *for each target URL*. 198
8.3 Number of IP addresses with amplification factor over 100× for each combination of target URL and packet sequence. Bolded is the maximum value *for each sequence*. 199
8.4 Nation-states with nation-wide censorship infrastructure and the fingerprint they most frequently respond to clients with. Numbers in parentheses denote packet sizes in bytes. 206

9.1 The current state of residual censorship, among the countries and protocols we tested (those that we tested but are not in the table did not residually censor in our tests). We were unable to reproduce SNI censorship in China; in that row, we report prior results [7]. *: Iran's SNI residual censorship sometimes lasts longer than 180s; in a small number of our experiments, we found it to last upwards of 5 minutes. 231

9.2 Success rates in weaponizing each country's censorship infrastructure against each victim vantage point from our attacker in Seattle, WA. (✓ denotes 100%, ✘ denotes 0%, and N/A denotes a location that does not cross the border of the censor.) Note that the success rates are not always consistent, even to victims in the same country, or between censored protocols in each censored regime. Iran is consistent and reliable; Kazakhstan is consistently unreliable for HTTP, but consistently reliable for HTTPS. In China, however, the attack was not always consistent by protocol, victim location, or server location. 239

List of Figures

4.1 Server-side evasion strategies in China. All of the strategies work without modifications to the client, and yet they induce client-side behavior that helps circumvent censorship. (Standard packets at the beginning and the end are grayed out to emphasize the critical differences from normal behavior.) . . . 72
4.2 Server-side evasion strategies that are successful against HTTP in Kazakhstan. . . . 83
4.3 Single versus multiple censorship boxes. A standard assumption is that evasion strategies that work for one application will work for another within a given country. However, our results indicate that China's GFW uses distinct censorship boxes for each protocol, each with their own network stacks (and bugs). . . . 88

5.1 **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`. . . . 101
5.2 **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 [8] allows for multiple DNS Questions to be included with no separator between them. . . . 101
5.3 **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). 117

6.1 Iran's layered censorship system, employing defense in depth. Note that the order of censorship systems is unknown; this is simply a graphical depiction. . . . 142

7.1 A waterfall diagram of the TCP 3-way handshake and the TLS handshake, denoting where the already known MB-RA and newly discovered MB-R middleboxes act during the connection. Note that MB-R does not act until deeper in the handshake than MB-RA (and *only* if MB-RA does not act), seemingly acting as a backup middlebox for China's HTTPS (SNI) censorship. 162

8.1 The maximum amplification factor we obtained per IPv4 address, based on several Internet-wide scans. (Note: the axes are log-scale.) . 177
8.2 Rank order plot of maximum amplification factor from Quack-identified IP addresses. The maximum amplification factor was 7,455×. 186
8.3 Types of attacks we find. Thick arrows denote amplification; red ones denote packets that trigger amplification. We find that infinite amplification is caused by (d) routing loops that fail to decrement TTLs and (e) victim-sustained reflection. 195
8.4 Rank order plot of the amplification factor received from each IP address for the triggering payloads containing www.youporn.com across all five packet sequences. 198
8.5 Rank order plot of the amplification factor received from each IP address for the ⟨SYN; PSH+ACK⟩ packet sequence across all seven scanning payloads. 200
8.6 The increase factor in the number of bytes we receive between sending 5 probes and sending 1 probe. 46% of IP addresses responded with exactly 5× as much data. 202
8.7 The fraction of the top million hosts that we confirm are middleboxes, using TTL-limited probe. The small gap at $x \approx 100{,}000$ and the large gap in the middle of the plot correspond to networks that block traceroutes at their borders. Accounting for this, we find injected responses from 82.9% of the top million IP addresses are from confirmed middleboxes. 204
8.8 Rank order plot of the amplification factor by country for the www.youporn.com scan with the ⟨SYN; PSH+ACK⟩ packet sequence. 207
8.9 CDF of the increase factor in amplification of candidate looping IP addresses when scanned with a TTL of 255 and 64. Because the increase factor is affected by the number of hops away an IP address is, we expect routing loops to have an increase factor of at least 4. Larger increase factors are further away from our scanner, limiting the overall amplification factor from our perspective. 210
8.10 The /24 prefixes with at least one routing loop, rank-ordered by the fraction of their 256 IP addresses that we observe to loop. Of the 2,763 looping prefixes, 54 (2%) have over 90% of their IP addresses loop, but 1,705 (62%) have only one looping IP address. (Note that the x-axis is log-scale.) 211

8.11 Attack bandwidth received at two vantage points from a self-sustaining amplifying IP address, which (based on its block page) appears to be a component of a Russian ISP's censorship system. The dashed line marks when the packet sequence was sent from the second vantage point. Note how the bandwidth we get from the system is divided evenly between the vantage points. This experiment supports our hypothesis that self-sustaining amplification is caused by an infinite routing loop. 214
8.12 Rank order plot of amplification factor of two scans for the `www.youporn.com` keyword requested with the ⟨`SYN`; `PSH+ACK`⟩ packet sequence: one with outbound `RST` and `RST+ACK` packets being dropped and the other normally. 216

9.1 Vantage points in our experiments. The green dot is our attacker running SP^3 [9]; black dots represent victim vantage points; and the red dots denote the location of the servers inside the censoring regimes we studied: China, Iran, and Kazakhstan (outlined in red). Note that some dots overlap. 229
9.2 The relationship between the number of times censorship is triggered and the reliability of HTTP residual censorship, as measured from our Beijing 2 vantage point. As the number of times residual censorship is triggered increases, the reliability improves. (Error bars represent 95% confidence.) 237

Chapter 1: Introduction

Many nations around the world today engage in country-wide censorship of Internet traffic. Although there are many forms of censorship—including political pressure [10], outright blocking of certain protocols [11, 12], or simply taking large swaths of the internet offline [13]—one of the most pervasive form of online censorship involves *in-network monitoring* and censoring of forbidden keywords. China [14], Pakistan [15], and more [10] deploy on-path middleboxes—similar to network intrusion detection systems (NIDS) [16]—that monitor all the Internet traffic that crosses their borders to detect, tear down, and in some cases outright block network connections that carry a prohibited word, content, or protocol that they view as threatening. These countries regularly block news, information about women's reproductive health, political views that oppose those in power, and recently even credible allegations of sexual assault against top political officials [17, 18]. Such restrictions to open sharing of information threaten the freedoms of billions of users worldwide, especially marginalized groups.

For years, security researchers have engaged in a cat-and-mouse game, developing new schemes to evade [16, 19–26] censors, who in turn have developed increasingly sophisticated countermeasures [12, 27–32]. Unfortunately, censors have

long had an inherent advantage.

Discovering new censorship evasion techniques has, to date, been a laborious, manual process. Details of censors' infrastructures and implementations are generally not made publicly known, and thus researchers typically must first measure and develop an understanding of how a particular censor works before they can develop strategies to evade them [23, 24]. Further complicating matters, many of the middleboxes that power censorship systems operate transparently, adhere to no open standards, and multiple different middleboxes may run in parallel to censor the same content [33]. As a result, when a new censorship technique is deployed or new content is censored, there can often be considerable loss of availability until the new censorship technique is detected by researchers, measured, reverse-engineered, and circumvented [34, 35].

My insight is to *automate* the discovery of censorship evasion techniques. Automated approaches to evasion allow evaders to react quickly to new censorship techniques or deployments. I focus my thesis on studying the core building blocks of censorship infrastructures themselves—middleboxes—and how an attacker can render them ineffective at implementing their network policies. In so doing, I expose problems that are broader than the censors themselves.

1.1 Thesis

It is possible to automatically discover packet sequence modifications that render deployed censorship middleboxes ineffective across multiple

application-layer protocols.

By "deployed censorship middleboxes", I refer to the middleboxes that power censorship infrastructure that are currently in use as of time of writing. Although most of this dissertation will focus on nation-state censorship infrastructure, in Chapter 8, I will also demonstrate attacks on non-nation-state middleboxes. By "ineffective", I specifically mean "not correctly implementing its policy", and I foresee two categories in which this failure can occur: Either a middlebox can fail to correctly censor a connection when it should, or it can incorrectly try to censor an innocuous connection. In this dissertation, I will demonstrate both cases across multiple protocols and across multiple nation-state censorship systems. I will also discuss what my results suggest about the limits of this approach.

To evaluate this broad thesis statement, I decompose it into the following research questions:

- Is it possible to automate the discovery of censorship evasion through client-side manipulation of *IP and TCP headers*?

- Is censorship evasion possible without requiring clients inside of censoring regimes to take any anti-censorship measures whatsoever? Can these server-side evasion strategies be discovered automatically?

- Is it possible to automate the discovery of censorship evasion through client-side manipulation of *application-layer data*?

- Can automatically-discovered evasion strategies allow researchers to gain novel

insights into how censorship infrastructures operate?

- Is it possible for attackers to weaponize censorship infrastructures, and can those attacks be discovered in an automated way?

My work answers each of these questions in the affirmative, thereby collectively proving my thesis. Moreover, I prove my thesis constructively, resulting in various open source tools and other contributions, which I summarize next.

1.2 Contributions

Constructively proving this hypothesis leads to the following contributions:

Geneva, a new open-source tool for automating the discovery of censorship evasion strategies. Geneva demonstrates that it is possible to automate the discovery of censorship evasion strategies, even against a black-box adversary. I developed Geneva and released it open-source, and its extensibility has enabled us to successfully respond quickly to new censorship events [3, 4, 36].

Discovery of the first server-side evasion strategies. Until this work, censorship evasion always required the client to do something in order to evade censorship (such as to install or configure anti-censorship software). This is because it is more difficult to discover strategies that work from the server side because there is little opportunity for the server to influence the state of the connection.

This work presents the first known censorship evasion strategies that work exclusively from the server, enabling servers outside of censoring regimes to subvert

censorship on users' behalf. Server-side strategies can be easier to deploy in real-world settings, as modification of packet headers typically requires elevated privileges that are difficult to attain on mobile devices.

Discovery of the first TCP-based reflected amplification attack. To date, almost all reflected amplification attacks have leveraged UDP. This is because launching non-trivial (going beyond the `SYN`) amplification attacks over TCP had long been thought to be impossible: to go beyond the `SYN` would seem to require an attacker to (1) guess the amplifier's 32-bit initial sequence number (ISN) in their `SYN+ACK` packet and (2) prevent the victim from responding to the amplifier with a `RST` [37]. This work demonstrates that TCP-based reflected amplification attacks are indeed possible: by leveraging TCP non-compliance in middleboxes, an attacker can leverage middleboxes as reflection points. In Chapter 8, I will demonstrate this attack and its evaluation on today's IPv4 Internet.

The first empirical analysis of residual censorship across multiple countries. Residual censorship is a little studied feature of many nation-state censorship systems. After a given TCP connection triggers a censor (e.g., by including a forbidden keyword in a plaintext HTTP GET request), some censors not only tear down the connection, but "residually censor" all *future* communication between the two end-hosts (on particular ports) for some period of time—even if the subsequent traffic is completely innocuous. I perform the first empirical survey of the current state of residual censorship around the world today: what countries employ it, how it operates, how long it lasts, and so on. My results demonstrate a wide variety

in the implementation of residual censorship systems—even within a given country, residual censorship can operate very differently from one protocol to another.

1.3 Ethical Considerations

Ethical considerations were a careful and important piece of this dissertation. As this work is not human subjects research, it falls outside the scope of my university's IRB. Still, many chapters of this dissertation posed unique ethical considerations; for this reason, each chapter will describe its own ethical considerations and responsible disclosure process where appropriate.

1.4 Roadmap

The rest of this dissertation is structured as follows:

Chapter 2: Background and Threat Model

I will start by offering a background that is relevant to all subsequent chapters: on middleboxes and the wider space of censorship research, with a particular focus on censorship evasion of nation-state censorship. I will also discuss packet manipulation for censorship evasion, the foundation for my thesis work, and reason about the threat model described by nation-state censorship infrastructure. Some chapters will require more specific background material, so I will provide background specific to each chapter within that chapter if relevant.

Chapter 3: Discovering Client-side Evasion Strategies with Geneva

Next, I will present the design and results of Geneva, a novel genetic algorithm

that evolves network-level censorship evasion strategies directly against real world censors. Geneva automatically discovers TCP/IP packet manipulation sequences that, when applied only from one side of the connection, confuse censoring middleboxes without impacting the underlying connection. In the lab, Geneva quickly re-derived almost all prior work in the space of packet manipulation strategies. Against real world censors in China, India, Kazakhstan, and Iran, Geneva has discovered dozens of strategies, including previously unknown strategies those that exploit what seem to be bugs in implementation in censors. This chapter demonstrates that it is possible to automatically discover packet manipulation strategies that render nation-state censorship middleboxes ineffective at enforcing their policy.

Chapter 4: Server-side Evasion

Next, I show that using Geneva, server-side censorship evasion is possible, allowing a server to subvert censorship on a client's behalf. This permits unmodified clients to connect directly to forbidden servers without requiring them to install any anti-censorship software. I evaluate this approach across 5 different network protocols (HTTP, HTTPS, DNS, FTP, and SMTP), demonstrating that it is possible to automatically render nation-state censorship middleboxes ineffective at enforcing their policy across multiple network protocols.

Chapter 5: Application-Layer Evasion

In this chapter, I show that it is possible to discover censorship evasion strategies that themselves operate exclusively at the application layer. I design new modification primitives to explore modifications to HTTP and DNS requests, and show that even modifications limited to these application-layer protocols can render mid-

dleboxes ineffective.

Chapter 6: Censorship-in-Depth: Iran

Nation-state middlebox deployments often involve multiple middleboxes deployed in parallel, creating "censorship-in-depth". These deployments make finding censorship evasion strategies and studying the censorship systems more difficult. To evade censorship, we would need to find an overlap in evasion strategies that defeats both systems, and to study either system individually, we would need to be able to disentangle the effects of both systems. In this chapter, I study a novel example of censorship-in-depth in Iran, and show that even in these cases, it is possible to *individually* render middleboxes ineffective at properly enforcing their policies.

Chapter 7: Censorship-in-Depth: China's SNI Censorship

Next, I present a second example of rendering middleboxes ineffective in a "censorship-in-depth" deployment. I study China's deployment of a secondary, backup censorship system to their existing HTTPS (SNI) middleboxes. Unlike in Iran, this is a system in which two different middleboxes operate on the same set of packets with the same goal. This chapter, as with the previous, supports my thesis in the context of more complex, real-world middlebox deployments.

Chapter 8: Weaponizing Censors for Amplification Attacks

In this chapter, I present a new attack that shows that middleboxes can be coerced into (trying to) enforce their policy when they should not. The new attack works by eliciting censorship responses from middleboxes to launch volumetric reflected denial of service attacks.

Chapter 9: Weaponizing Censors for Availability Attacks

In this chapter, I present a second attack that demonstrates that nation-state censors can be coerced into blocking arbitrary IP pairs from communicating across their borders across multiple protocols. This attack makes use of a relatively little studied feature of many nation-state censorship systems: residual censorship.

Chapter 10: Defending Against Geneva

Before concluding, in this chapter I take a step back and reason about what it would take to defend against the myriad attacks I present in this dissertation. What are the limits of this work, and should we expect it to work forever?

Chapter 11: Conclusion and Future Work

Finally, I conclude by revisiting the contributions of this work. I discuss immediate next steps for this work, and comment on future challenges in the censorship evasion space.

Chapter 2: Background and Threat Model

In this chapter, I provide a background relevant to all chapters of this thesis: on nation-state censorship and middleboxes. I will also define the threat model that this work operates within. Some chapters will require additional background material specific to that chapter; where appropriate, individual chapters will provide additional background material.

2.1 Nation-state Censors: Threat Model

Much of this dissertation studies *nation-state censors*. These are powerful entities who are able to inspect [16], inject [38], and sometimes also drop [39] traffic throughout their countries. Nation-state censors operate in two broad ways: *on-path* (man-on-the-side) or *in-path* (man-in-the-middle) [24, 40], and my experiments span both kinds. In this section, I will also discuss other relevant properties of nation-state censors: failing open or closed, the eavesdropper's dilemma, and more.

On-path Censors On-path (man-on-the-side) censors can obtain copies of packets, allowing them to overhear all communication on a connection. To determine whether to censor, these attackers perform deep-packet inspection (DPI) and typ-

ically look for keywords they wish to censor, such as DNS queries [28, 38, 41] or resources in HTTP GETs [23, 24, 42].

On-path censors are also able to inject packets to both ends of the connection. Because they are able to view all traffic on the connection, they can trivially inject packets that the end-hosts will accept—unlike traditional *off-path* attackers who must guess sequence numbers, query IDs, or port numbers [43, 44]. On-path censors have been observed to inject TCP `RST`s to tear down connections [16, 23, 24, 40, 42, 45, 46] and DNS lemon responses to thwart address lookup [38, 41].

To reconstruct application-layer messages and track sequence numbers, on-path censors maintain a Transmission Control Block (TCB) for *each* flow. A TCB comprises sequence numbers, received packets, and other information about the connection. A considerable amount of work has gone into modeling and understanding how censors *synchronize* and *re-synchronize* their TCB state with the ongoing connection's [23, 24]. Understanding this can enable researchers to craft a packet sequence that causes the censor to synchronize on incorrect data.

In-path Censors In-path (man-in-the-middle) censors also perform DPI to determine whether to block a connection, but they can do more than just inject a `RST` or lemon response. For example, an in-path censor is able to simply drop a connection's packets altogether. Alternatively, an in-path censor can also hijack a connection entirely, inject a block-page, and prevent the client's packets from reaching the server. Evading an in-path censor requires tricking the censor into believing that a connection should not be censored, for instance by hiding the true identity of

the server [20, 21, 47], obfuscating the protocol [11, 48, 49], or modifying the packets in such a way that the censor no longer recognizes the forbidden query as a target.

The Eavesdropper's Dilemma Almost all on-path and in-path middleboxes must contend with the *eavesdropper's dilemma*, which states that it is difficult to accurately model the state of a connection from the middle of that connection [50]. The reason for this is that unless a middlebox ensures that every packet is delivered, accepted, and processed by the end-server in the same way as the middlebox, an attacker may be able to tamper with the middlebox's internal state about the connection. For example, I will demonstrate an attack in Chapter 3 in which the attacker sends a packet with a payload and a reduced `TTL`: the packet will be processed by the middlebox (causing it to advance its internal TCB), but will be dropped before it reaches the end-server. In this case, the middlebox will now be desynchronized from the connection, and may be unable to correctly inspect the rest of the flow. As I will discuss in Chapter 10, for a middlebox to mandate consistent state with end-hosts can be difficult in practice, and may require a significant re-architecture of the censorship infrastructure in the world today. Every nation-state middlebox I study in this dissertation is susceptible to at least some attacks enabled by the eavesdropper's dilemma.

Failing Open Most nation-state censors operate in a *fail-open* capacity: any packet that cannot be processed or matched to internal state is allowed to pass. Failing-open reduces the collateral damage of censorship (e.g., an unknown protocol will not be erroneously targeted), but it presents more opportunities for evaders. It

is difficult for an on-path censor to reliably fail closed, however: if the middlebox requires connection state to disrupt a that connection (as is the case with injected `RST`s), if that state is incorrect, the censor will be unable to correctly censor the connection.

Across all the middleboxes I study in this dissertation, only one approximates a fail-closed system: Iran's Protocol Filter, discussed in Chapter 6. This system operated a strict protocol allow-list, and any protocol or packet sequence that could not be positively identified would be dropped. As we will see in Chapter 6, however, this system is not a perfect fail-closed system, and was still susceptible to attacks.

Throughout this dissertation, I will be explicit about the specific threat model that each individual censorship system falls into. In addition to the above information, I make several common assumptions that hold across all the threat models in this dissertation. I assume that censors cannot break encryption that is considered secure: only publicly known weaknesses are considered. I also assume censors do not have sufficient resources to record indefinite packet captures of all network traffic leaving their borders.

2.2 Related Work: Measuring Censors

There has been a wide range of work measuring how censors work and what they block. This can be broadly broken down into two broad categories:

First are studies into what specific content or destinations censors block [39, 51–54]. My work is largely orthogonal to these prior efforts; the primary goal is not

to discover *who* or *what* is being censored, but to measure and understand *how* it is being censored (and evade it).

Second is the body of work that studies how censors operate [12, 14, 15, 23, 24, 28, 38, 55, 56]. My work is complementary to these prior efforts, in that I am able to lend new insights into how several censors perform on-path censorship, as well as gaps in their logic and bugs in their implementations. For instance, I believe I am the first to observe that censors use different transport-layer techniques depending on the overlying application.

2.3 Evasion via Packet Manipulation

There is a history of evading on-path and in-path censorship through the application of *packet-manipulation strategies*. At a high level, these techniques alter and inject packets at one of the communicating endpoints (typically the client). In so doing, their goal is to either de-synchronize the censor's state (e.g., by injecting TTL-limited `RST`s [57]) or to confuse the censor into not recognizing a forbidden keyword (e.g., by segmenting TCP packets).

Client-side evasion The earliest packet-manipulation strategies to evade on-path censors come from an open-source project from 2011, sniffjoke [46]. sniffjoke introduced a handful of client-side strategies, such as injecting packets with random sequence numbers or injecting packets that shift the sequence number but corrupt the payload. Unfortunately, many of the specific strategies sniffjoke employed have long been defunct, but its broad approaches were later re-discovered by

other work [23, 24].

In 2013, Khattak et al. [16] crafted 17 different evasion strategies to exploit specific implementation weaknesses against the GFW. In 2017, Wang et al. [24] developed a suite of highly effective hand-crafted strategies, and their open-source system INTANG could systematically identify the best evasion strategy from this suite for a given server and network path. They perform empirical tests regarding the behavior of the GFW, and make hypotheses on previously unknown updates to the GFW. Li et al. [23] studied numerous middlebox traffic classifiers in their 2017 work, and pioneered automated work of identifying traffic differentiation. Once traffic differentiation is detected, their system could choose from a library of pre-built evasion techniques to evade the censor. They tested their work on many censorship regimes, including the GFW, and many of the censorship techniques they leverage are still relevant today.

My work is informed by and extends these prior efforts: I will present over 100 censorship evasion schemes discovered by Geneva, including some previously thought impossible. My results also lead me to refine prior work's findings. For instance, Wang et al. [24] showed that the GFW was capable of reassembling TCP streams to detect censored keywords in HTTP requests; my result confirms this for HTTP, but show that the GFW is frequently incapable of doing so over FTP, indicating that censors use different transport-layer techniques depending on the application.

Server-side evasion To the best of my knowledge, all prior censorship evasion systems (including Geneva in Chapter 3) require some degree of client-side eva-

sion software. Even techniques that rely on server-side features, such as domain fronting [58] or decoy routing [21], require client-side changes. However, there are two server-side strategies that are similar in spirit to the novel server-side censorship strategies I will describe in Chapter 4. In 2010, Beardsley and Qian [59] demonstrated that a variant of TCP simultaneous open was able to bypass some intrusion detection systems; these do not appear to work against censors, but we show in §4.3 that Geneva discovered multiple simultaneous open-based strategies that work against China's GFW. brdgrd [45] intercepted packets sent by a Tor bridge to the Tor client, and employed a relatively simple strategy—it lowered the TCP window size of outbound `SYN+ACK` packets. This caused Tor clients to segment their TLS handshake packets, splitting the set of supported ciphersuites across multiple TCP packets. At that time, the GFW was unable to reassemble TCP segments, and thus this strategy avoided detection and blocking. In 2013, the GFW added the ability to reassemble TCP segments, rendering brdgrd defunct. Since then, we are aware of no other work on this topic: all prior literature in this space has explored only client-side strategies [16, 23, 24, 40].

More Broadly Beyond packet manipulation-based censorship evasion, there is a much wider space of prior work for circumventing censorship. Researchers have explored tunneling traffic over a wide variety of mediums, including email [60], video games [61], VoIP [62], SSH [63], WebRTC [64], HTTP [65], just to name a few. Other systems seek to hide the true destination of traffic, such as with Tor [20], domain fronting [58], Decoy or Refraction Routing [21, 47, 66, 67], or to avoid the censoring

country altogether (Alibi Routing [68], DeTor [25]). Traffic mimicry systems have also been developed to disguise network traffic as another protocol [48,49,69]; though these appear to have inherent limitations [11]. Geneva is orthogonal to all of these systems, and, as demonstrated with INTANG [24], could be used in tandem with them to help bolster their ability to circumvent censors.

2.4 Automating Censorship Evasion

In the next chapter, I will describe the design and implementation of Geneva, the first system to automate the discovery of censorship evasion strategies. Since Geneva's publication, however, there have been two notable works in the space of automating censorship evasion that deserve mention here.

In 2020, Wang et al. released SymTCP [70]: a system to automatically discover discrepancies between how censors and end servers process packets using symbolic execution of the TCP implementation in Linux. SymTCP offers a contrasting approach towards the same end goal as Geneva: while Geneva treats the censor and the end-host as "black boxes" and explores the space of strategies by evolution, SymTCP performs symbolic execution of the end host's TCP stack and explores the strategy space systematically. There are trade-offs between the approach taken by Geneva and by SymTCP. SymTCP's principled exploration of the strategy space offers a more deterministic approach towards censorship evasion. However, by treating censors and end-hosts as black boxes, Geneva can be more easily deployed against previously unknown or new censorship systems. For example, no additional effort

is required to train Geneva with a censored Windows HTTPS server compared to a Linux SMTP server, whereas SymTCP would require the ability to execute that server within its symbolic execution engine. This has allowed Geneva to be highly responsive to new censorship events and systems [36, 71].

In 2019, Moon et al. released Alembic [72], a system to automatically infer state models for middleboxes. Alembic applies symbolic execution and finite-state machines (FSM) to infer the state of a stateful firewall. Alembic takes a contrasting approach to Geneva: Geneva defeats censorship *first*, and then researchers can infer the firewall's model from the strategies it discovers and discards, whereas Alembic first discovers the firewall's model, and then researchers can use that model to determine evasion strategies. Like SymTCP, Alembic offers a more principled approach towards identifying evasion opportunities, and knowing a firewall's model can be useful beyond just identifying evasion strategies, such as to improve the accuracy of network testing tools. With nation-state censorship, Geneva offers other advantages over Alembic. For firewall model inference, Alembic requires an offline training stage that can last for tens of hours, which may not scale to real nation-state censors. Further, Geneva supports a much larger "alphabet" of potential actions (and the ability for researchers to add new actions), making Geneva more expressive. However, Alembic and Geneva have not trained against the same systems, so it is difficult to compare their effectiveness directly.

2.5 Fuzzing

Fuzz testers [73] mutate inputs non-deterministically in an effort to evaluate the correctness, security, and coverage of programs. Most relevant to my work is the space of grammar-based fuzzers, which define an input grammar for 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 [74], including web applications [75] and other popular protocols [76]. 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 [77]. WFuzz is another powerful fuzzer for HTTP web servers, but it has no support for other protocols or extending its grammar [78].

My work differs from existing fuzzers in two subtle but important ways: First, Geneva 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, my 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 its manipulation primitives) that are in turn applied to inputs (user traffic). Therefore, we search over the space of manipulation actions,

not over the input space itself.

Genetic algorithms have been used for fuzzing, including in the well known American Fuzzy Lop (AFL) [74] and iFuzzer [79]. Genetic algorithm fuzzing techniques have been applied to web applications [75] and other popular protocols [76]. To my knowledge, I am the first to apply such techniques to censorship evasion.

Chapter 3: Discovering Client-side Evasion Strategies with Geneva

I begin by demonstrating that it is possible to automate the discovery of evasion strategies through client-side manipulation of IP and TCP packets. To achieve this, I have designed and implemented Geneva, a novel genetic algorithm that *discovers* how to evade censorship against a live adversary. I trained Geneva against real-world censorship infrastructure in China, India, Iran, and Kazakhstan, and present a total of 27 strategies (including strategies that prior work posited should be impossible). I will detail how these strategies work, and what these new evasion strategies teach us about how Chinese censorship works. This chapter will demonstrate that censors can be rendered ineffective.

3.1 Geneva Design

In this section, I describe its genetic algorithm-based design in terms of its building blocks and how it composes and evolves them over time. I begin by providing a high-level overview of the approach.

3.1.1 Overview and Challenges

Genetic algorithms [80] are a biologically-inspired approach to automate algorithm design. They require three core components: (1) *genetic building blocks* that provide a way to programmatically represent different algorithms, (2) a *fitness function* to capture how well a given algorithm performs, and (3) methods for performing *mutation* and *crossover* to generate new algorithms. Iteratively, over successive *generations* (rounds), genetic algorithms simulate evolutionary natural selection: Given a set of *individuals* (candidate algorithms), it runs each one to compute their fitness, allows only some of the fittest to survive, and mutates or crosses-over the surviving ones to generate new individuals for the next generation.

One primary challenge faced in applying genetic algorithms to censorship evasion lies in how many degrees of freedom we permit in its genetic building blocks. On the one hand, we could allow virtually unlimited degrees of freedom by, say, treating all packets merely as bit strings and allowing the genetic algorithm to construct strategies out of bit flips, bit removals, and bit insertions. Such an approach would *eventually* learn virtually any possible strategy, but would require an inordinate amount of time to do so. On the other extreme, we could use existing evasion strategies from prior work as building blocks; this would learn more quickly, but risks "over-fitting" to the strategies that are already known. Therefore, Geneva needs genetic building blocks that balance between finding new strategies and finding them efficiently.

3.1.2 Geneva's Genetic Building Blocks

Strategies in Geneva comprise a set of (*trigger*, *action tree*) pairs. Packets that match a given trigger (for instance, all TCP packets with the `ACK` flag set) are modified using the corresponding sequence of actions in an action tree. We permit Geneva to evolve the triggers, the structure of the action trees, and the properties of the individual actions themselves.

Here, we present the design of triggers, actions, and action trees, as well as a *syntax* that comprises the genetic code of individuals to unambiguously describe Geneva strategies.

Triggers *Triggers* represent fields in a packet header that, when matched, cause packet manipulation actions to be applied. In this work, we have restricted triggers to span only TCP and IP, though adding support for additional protocols is straightforward in our implementation. Triggers are expressed with the following syntax: `[PROTOCOL:FIELD:VALUE]`. For example, `[TCP:flags:R]` is a trigger that fires when the `TCP` field `flags` is set to `RST`. Geneva requires *exact* matches: for instance, a packet with only the `TCP RST` flag set would not match a trigger for `[TCP:flags:RA]`.

Actions To balance expressiveness with efficiency, we permit four distinct packet-level actions:

1. `duplicate(`A_1`,` A_2`)` copies a packet and applies action sequence A_1 to the original packet and A_2 to the duplicate.

2. `fragment{protocol:offset:inOrder}`(A_1, A_2) fragments or segments the packet (depending on if the protocol is set to IP or TCP) at a specific byte `offset`, applies A_1 to the first fragment, A_2 to the second, and optionally returns them `inOrder`.

3. `tamper{protocol:field:mode[:newValue]}`(A_1) alters the given `field` of a packet and then applies action sequence A_1 to it. `tamper` always tries to keep the packet in a valid state unless otherwise directed, and will recompute the headers' checksums and/or lengths if needed (unless `field` is a checksum or length). Note that if the specified `field` is optional and not present, such as a TCP option, it will be added to the packet. `tamper` has two `mode`s of operation: `replace` and `corrupt`. `replace:newValue` sets the given `field` of the packet to `newValue`. `corrupt` replaces the given `field` of the packet with a random value of the same bitsize (a new random value is selected each time the action is invoked).

4. `drop` causes a given packet to be dropped.

Action Trees Geneva's actions are composed to form a binary tree: `duplicate` and `fragment` both have two children; `tamper` has one child; and `drop` has no children. An action tree encapsulates a packet modification scheme—each packet that matches the associated trigger enters at the root of the tree and is passed down via in-order traversal to the actions of the tree. Packets that emerge at the leaves are sent on or accepted from the wire. We refer to an ordered list of (trigger, action tree) pairs as a *forest*, and forests can be combined to represent a strategy. Triggers

need not be unique within a forest—if multiple action-trees have the same trigger, each action-tree is given its own fresh copy of the original packet, and runs serially, in isolation, in the order the trees exist in the forest. Note that action-trees are stateless, and operate only on singular packet inputs (though they may result in sending multiple packets). An interesting area of future work would be to extend Geneva to operate over packet *streams*.

Outbound vs. Inbound We allow Geneva to evolve action-trees for both inbound and outbound packets. A strategy in Geneva is thus two components: an inbound and outbound forest of triggers and action-trees. This lets Geneva independently alter outgoing packets and alter (or ignore) incoming packets. Due to limitations of NFQueue, branching actions (`duplicate` and `fragment`) are disallowed in inbound forests. We represent the overall strategy syntactically as `outbound-forest \/ inbound-forest`.

Example To demonstrate Geneva's syntax, consider the following:

Strategy 1: TCB Turnaround / `RST` Drop

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

This example strategy has one outbound and one inbound tree. The first (outbound) action-tree duplicates outgoing `SYN` packets; it replaces the first copy's TCP flags with `SYN/ACK` before sending it. It then sends the second copy of the

SYN packet unmodified. On the inbound forest, the only action-tree triggers on RST packets and drops them. Collectively, this strategy implements a hybrid of two previously known strategies: *TCB-Reversal* [24] (characterized by sending a SYN/ACK before the three-way handshake) and *RST-Drop* [42]. (Unfortunately, as we will see in Section 3.3, both halves of this hybrid species are now extinct against the GFW.)

Expressiveness Note that Geneva's genetic building blocks reflect the set of packet manipulations that can occur at the IP layer: as a result, we posit that they can be composed to generate *any* packet stream. To evaluate this hypothesis, we tested whether it was possible to express all prior work's strategies [16, 23, 24] through combinations of `duplicate`, `fragment`, `tamper`, `drop`, and `send` alone. Indeed, we were able to express 30 (83.3%) of the 36 previously published strategies—the only exceptions were strategies that (1) manipulated HTTP packets, as was done by Khattak et al. [16], and those that (2) paused for 40–240 seconds, as was done by lib·erate [23]. These are not fundamental limitations: one could easily extend Geneva to support HTTP manipulation or sleeping through `tamper` actions. For this chapter, we chose to limit Geneva to only manipulate IPv4 and TCP (as this was the central focus of most prior work), and not to include pauses: including pauses would significantly slow down training time. As we will show in §3.2, Geneva was able to independently discover all of these 30 strategies in in-lab experiments, and it discovered many more strategies when trained against a live censor: China's GFW. Geneva automatically derives these strategies through the process of evolution, which

we describe next.

3.1.3 Evolution

Geneva automatically derives censorship evasion strategies through *evolution*, which takes place over a series of discrete *generations*. Each generation comprises multiple individuals (strategies, represented as inbound and outbound forests of action-trees), and includes three broad steps: (1) mutation and crossover, (2) evaluation of individuals' fitness, and (3) selection of individuals to survive to the next generation.

Population Initialization We explored two ways to initialize Geneva's population. For most of our experiments, we *randomly generated* an initial population of individuals. We generated 200 individuals, each with random but valid action-trees with precisely 3 actions each. Additionally, we explored *seeding* the population with "extinct" strategies. With a population seed, the initial population is comprised of duplicates of the seed: this allows the algorithm to focus evolution on improving a given strategy.

Mutation As in biological systems, Geneva's genetic building blocks can be altered through random mutations. Mutations can occur at the level of actions, action-trees, and entire individuals. Each action mutates in the following ways:

- `duplicate` mutations swap the order of the children (i.e., `duplicate(`A_1`, `A_2`)` $\rightarrow$ `duplicate(`A_2`, `A_1`)`).
- `fragment` mutations change the protocol (fragmentation or segmentation), the

order of the packet fragments, or the fragmentation index.

- `tamper` mutations depend on the mode it is in: `replace` mode mutations can alter the field they replace or the new value it changes it to, whereas `corrupt` mode mutations can alter the field it corrupts. Both modes can mutate to the other mode.
- `drop` does not support mutations.

Triggers can also be mutated similarly to the tamper action: the protocol, field, or value to trigger on may be changed.

To mutate an action tree, one of four primitives is applied with some configurable probability[1]: a new action can be chosen at random and added to the tree in a random location (20% probability in our implementation), an existing action can be removed from the tree (20%), the trigger can be mutated (20%), or one of the actions can be mutated (40%).

An individual (which in turn comprises outbound and inbound action-forests) can be mutated in one of four ways, also with configurable probability: a new random action tree can be added to one of its forests (10%); an existing action tree can be removed from one of its forests (10%); trees in its forests can be reordered (5%); or specific trees within each forest can be mutated (25%). In each generation, each individual is mutated with a configurable probability (90%).

As actions and triggers must operate on real-world packet data, it is challenging to mutate the actions or triggers in such a way that it results in packet values that

[1]We verified that Geneva was still effective when each option was chosen with equal probability. We chose our specific values based on our intuition during in-lab experimentation, and leave a full parameter sweep optimization for future work.

are seen in the real world. For example, if the algorithm was to mutate the TCP flags header field to a valid random value (any value from 0–65535) it would very rarely choose a valid combination of TCP flags. Therefore, during mutation, actions and triggers are given access to a packet capture of their previous run against a censor. The triggers (and tamper action) can draw from the values contained in real packets to mutate.

Drawing from real packet captures also confers a second advantage to the evading system. If the censor interacts with the strategy (e.g., by forging `RST` packets), these injected packets will be available in the packet capture for the action system to draw from and use for mutation. This allows action trees to find triggers that apply only to injected packets.

Crossover Unlike mutations—which are random perturbations of singular strategies or actions—*crossovers* serve as a form of "breeding" between two different individuals. To perform crossover, two individuals are chosen at random from the population pool, and one of the following occurs. Trees in each action forest are randomly *swapped*, or a randomly chosen tree in each forest is *mated* with a randomly chosen tree from the other. To mate two trees, an action is chosen from each tree, and the subtrees of that action are swapped between each tree. If each action forest for a specific direction only has one tree, crossover will be applied using the second mechanism.

In each generation, crossover is applied between every other individual in the pool with a configurable probability (40% by default). In our implementation,

crossover is applied before mutation.

Fitness At the end of each generation, all individuals are evaluated for their *fitness*. Genetic algorithms rely on some domain-specific fitness function when determining which individuals should be allowed to survive to the next generation. Geneva evaluates fitness by *running directly against the censor*. This way, Geneva evolves in the presence of the real deployment, and can therefore adapt to the details and idiosyncrasies of a particular censor's implementation.

To evaluate a given strategy, a Geneva client simply tries to make a forbidden GET request through an actual censor (or a simulated censor, for in-lab testing), while the strategy runs on the client side. The specific request depends on the censor: against the GFW, Geneva makes an HTTP GET request with a forbidden word, against India's Airtel ISP, we make an HTTP GET request to a blocked URL; against Kazakhstan's HTTPS MITM, we make an HTTPS request. Geneva assigns a positive numerical fitness metric if the connection can properly finish; if the connection is censored (is reset, blocked, or gets the injected certificate respectively), a large negative value is added to the fitness. As we will see in §3.3, some censors may not work 100% of the time. To prevent false positives in strategy evaluation, Geneva evaluates each strategy twice and records the lower of the two fitness scores.

Three additional adjustments are made to the fitness measure to help refine and optimize successful strategies: First, the fitness is punished if any *vestigial* action-trees are present—action-trees whose triggers which are never fired during an evaluation. Punishing for vestigial actions kills off strategies without effective

triggers early in the evolution process, allowing the framework to evolve good triggers before it discovers fully functional action-trees, and encourages pruning unused action-trees. Second, the fitness is punished for *strategy overhead*—the number of additional packets that a strategy adds to the data-stream. Punishing for strategy overhead encourages precise triggers (such as triggering only on `PSH/ACK` packets, instead of every packet). Finally, the strategy is punished for *strategy complexity*—a count of the number of actions across all of the action-trees in the strategy to encourage succinct strategies. Critically, punishments for strategy overhead and complexity are applied only when the fitness of an individual is positive to encourage the algorithm to explore the strategy space as much as necessary in the early stages of evolution.

Selection In the final step of a generation, Geneva runs a *selection tournament* [81]. Some individuals are drawn at random (with replacement) from the population; the highest-fitness individual among them is added to the *offspring pool*. This process repeats until the offspring pool is the same size as the population pool; then, the offspring pool becomes the population for the next generation.

Selection tournaments have several benefits. High-fitness individuals have a greater probability of being selected for the next generation—and because they are chosen with replacement, multiple copies of them are likely to be selected. This allows Geneva to focus on improving promising strategies. While low-fitness individuals decrease in number, they have non-zero probability of surviving to the next generation. This has the benefit of promoting genetic diversity, thereby steering

Geneva away from local maxima.

As the evolutionary framework will run for many generations, it is possible to find a successful strategy, but mutate away from it or break it in ensuing generations. To prevent the loss of successful strategies as the algorithm progresses, the system maintains a "Hall of Fame": a global sorted collection of every individual the algorithm has evaluated during a run. At the end of each generation, the Hall of Fame is updated with the highest performing individuals.

Strategy Coverage The evolutionary process we have described thus far does not, by itself, promote a broad exploration or coverage of the strategy space. As we will see in Section 3.3, when running in a real environment, some header fields have a higher probability of contributing to a successful strategy. As a result, Geneva tends to find them first, and there is no evolutionary pressure to deviate from those individuals to find new strategies. To broaden coverage, we add an optional meta layer on top of normal evolution: if, across multiple consecutive experiments a particular header field is repeated across all of the successful strategies, Geneva can preclude it from future training sessions. This encourages broader exploration in other portions of the space of potential strategies.

3.1.4 Implementation

We implemented Geneva in approximately 6,000 lines of Python. Geneva runs strictly at the client, and uses NetfilterQueue [82] to interpose on (and possibly alter) all of the client's outbound and inbound packets. As a result, Geneva does

not require any modifications to the applications. To demonstrate this, we deployed an *unmodified* Google Chrome browser on a client running Geneva in China, and, using the strategies we present in §3.3, verified that we were able to browse free of keyword censorship.

In its current implementation, Geneva requires root access—as with all prior work on packet-manipulation-based censorship evasion [16, 23, 24, 45, 46], root privilege is necessary for most of their packet manipulations. However, we demonstrate in §3.3 that Geneva is also able to find strategies that operate strictly through TCP segmentation. Strategies such as these could be deployed without root privilege. Recall that Geneva currently only supports modifications of IP and TCP packets; it would be straightforward to also add application-layer modifications, in the form of new `tamper` primitives for HTTP, DNS, and so on. These would not require root privilege, and given prior successes at application-layer manipulations [16, 23], we speculate that Geneva would also fare well, but this is beyond the scope of this chapter.

3.2 Validation

In this section, we validate Geneva's design by investigating whether it can re-derive strategies found from prior work [23, 24]. Unfortunately, the techniques employed by censors are not guaranteed to be the same today as when these prior studies were performed. To achieve a fair comparison, we have implemented mock censors that exhibit the behavior reported in prior work, and validate against them

in a controlled environment.

Mock Censors We first developed a suite of mock censors (11 in total) to mimic specific aspects of nation-state censor behavior as hypothesized by previous researchers [15, 23, 24, 55]. This includes on-path censors injecting TCP `RST` packets to disrupt a connection (China), varied TCB synchronization/teardown behavior (China, Iran), in-path censors dropping packets (India, China), TCB resynchronization behavior (China), and so on. A full list of the censors we developed is included below in this section.

We implemented a Dockerized [83] evaluation system for Geneva to train against these censors. We ran each strategy in an isolated environment with three containers (a client, a mock censor, and server). We isolated each training session from the others, with a starting population pool of 1,000 individuals, capped at 50 generations. In the lab setting, Geneva evaluated 3–5 strategies per second, and each generation took 4.4 minutes on average to complete.

Validation Results Geneva found successful strategies against every mock censor. We analyzed the strategies that Geneva discovered and found that, of the 36 strategies suggested by previous work [16,23,24], Geneva automatically re-derived 30 (83%) of them. The strategies that Geneva did not find are not possible to create with our genetic building blocks (drop, tamper headers, duplicate, and fragment). Specifically, Geneva did not rediscover the ability to delay packet transmissions [23, 24], perform state exhaustion [16,24], or perform HTTP-specific tweaks [16] (Geneva was not given the HTTP protocol structure to perform specific minor modifications).

In addition to learning simple behavior against weak censors, Geneva finds strategies in the *TCB Creation*, *Data Reassembly*, and *TCB Teardown* species, and learned more complex behavior. For example, prior work theorized that the GFW would enter a "resynchronization state" after a RST or RST/ACK, and that the GFW updates its TCB with the next packet in the stream. Such a feature would allow it to recover to continue censoring a connection, even after an injected insertion RST [24]. Against a similar censor in the lab, Geneva evolved a strategy that injects an insertion RST packet after the connection is established, then injects an insertion packet with an invalid sequence number. Geneva also evolved strategy variants with additional behavior, such as TCB Turnarounds, various fragmentation attacks, and different forms of TCB teardown [23, 24, 84]. While training in the lab, Geneva identified 9 now-patched bugs in scapy [85], a bug in Docker for Mac [83], and a bug in NetfilterQueue [82].

All the discovered strategies require only 1–2 action trees in the outbound forest to express; besides the initial strategy of dropping inbound RSTs, none of the strategies relied on the inbound forest at all (Geneva typically pruned them quickly).

Why does Geneva work? At first glance, it seems counter-intuitive that Geneva would be effective at searching the space of strategies: after all, there is no continuous cost function against which it can gradient descent (changing one TCP flag can cause the entire connection to terminate). Yet, Geneva finds a working strategy in all of its experiments (which comprise at most 10,000 individuals). By comparison, when we run a strawman scheme that simply generates random strategies, it found no

working strategies until we manually assisted it by handing it working triggers, and even then it only found *one* working strategy after 100,000 individuals. Why is Geneva so much more effective?

Observing Geneva's strategies throughout the duration of its experiments, we can broadly classify four major "development phases" that Geneva naturally goes through. First, Geneva learns which triggers are relevant; in early generations, individuals try a highly variable number of triggers, but those who randomly generate relevant triggers receive higher fitness, and the selection tournament converges on a set of workable triggers. Second, Geneva learns how not to kill the ongoing TCP connection; action trees that have at the root `tamper{TCP:chksum:corrupt}` are likely to be doomed—such action trees get very low fitness and are thus likely to be weeded out in the selection tournament. Third, with working TCP connections, Geneva tends to tweak its action trees through mutation, crossover, and mating to iterate on various modifications that ultimately trick the censor. Finally, with working strategies, Geneva's fitness function punishes strategies with more actions; thus mutations drive it towards smaller strategies until a local minimum is reached.

We emphasize that we did not encode these various "stages" into Geneva: these emerge naturally from its genetic algorithm and fitness function.

These in-lab validation experiments demonstrate that Geneva's genetic building blocks are expressive enough to span a wide range of strategies, and that our evolutionary process is effective at finding successful ones. Next, we evaluate against real world censors.

3.3 Evaluation against real censors

We have three high-level questions in evaluating Geneva: (1) Can Geneva find successful circumvention strategies *efficiently* when training against a real censor? (2) What *novel* strategies can Geneva find against a real censor? and (3) Does Geneva *generalize* to multiple censoring regimes?

To answer these questions, we ran Geneva against three nation-state censors: China's Great Firewall, India's ISP-based censorship (Airtel), and Kazakhstan's recent HTTPS MITM infrastructure. Table 3.1 lists the success rates, descriptions, and taxonomy of all strategies and strategy variants Geneva found against these censors.

3.3.1 Experiment Setup

Vantage points We used VPSes in Mainland China from four vantage points (Shanghai, Zhengzhou, Shenzen, and Beijing); in India, we used VPSes in Bangalore; and in Kazakhstan, VPSes in Almaty and Qaraghandy. Censorship strategies can vary based on ISP, routing path, or egress points [24, 86], but we observed no significant difference in the success rate between any two of our vantage points in any of the countries we tested. Nonetheless, it is possible that running Geneva from more locations would result in more varied success rates, or different strategies entirely.

Initialization In each evolution experiment we performed, we initialized Geneva with a set of individuals generated at random, each with three actions and one

trigger (all selected and parameterized with random values), and disallowed it from accessing results from previous runs. We configured each training session with a starting pool of 200 individuals, and capped it at 50 generations, or until population convergence occurred (whichever came first). On average, each generation generated approximately 500KB in outbound traffic and 2MB in inbound traffic. Each generation took 5–10 minutes to complete; overall, training sessions took 4–8 hours.

Triage Recall that during training, Geneva evaluates each strategy in the population by making real connections to censored resources as a part of the fitness function. To compute a success rate for a given strategy in a given country, we repeatedly evaluated the strategy from each of our vantage points within the country and averaged the success rates of each.

After Geneva completed its experiments, we then manually analyzed the set of successful strategies it found. To verify that all of the actions in each strategy were strictly necessary, we manually removed individual actions and verified that the strategy was no longer successful as a result. To better understand *why* the strategies were successful, we manually altered, removed, added, and swapped actions. We emphasize that all manual changes were only done as a post hoc analysis, and all strategies and strategy variants presented herein were independently discovered by Geneva.

3.3.2 China: The Great Firewall

We focus specifically on GFW's HTTP censorship. The GFW injects `RST` packets if a forbidden word is included in the URL of an HTTP GET request. The GFW also employs "residual censorship" [24]: after a client makes a censored request to a given website, the GFW forbids new connections between the client's IP address and the website's IP:port pair for approximately 90 seconds.

To avoid residual censorship, we compiled a pool of destination servers to train against by querying all sites from the Alexa Top 10,000 that are initially reachable with an HTTP GET but censored when the request includes a forbidden word. This allows us to test whether Geneva can be effective at evading keyword censorship of real, popular websites. It also filters servers that are in the GFW's IP blacklist (e.g., Facebook or Google); those blocked by DNS; and those hosted in-country (in which case the GFW may not necessarily be in-between our machine and the server). We find 7,917 sites out of the above 10,000 that were outside the GFW and not immediately censored. This is similar to GreatFire's census, which found that 147 of the top 1,000 Alexa sites are blocked in China [87]. While evaluating Geneva, we chose sites at random, limited to only those that were both accessible and not subject to residual censorship.

As previously shown [16, 23, 24], strategies deployed against the GFW do not succeed or fail consistently; in fact, if no strategy is used whatsoever, we find that it still succeeds 2.8% of the time. Throughout this section and in Table 3.1, we include each strategy's success rate against the GFW.

We allowed Geneva to train against the GFW directly in 27 discrete, isolated experiments over 16 days. Geneva discovered successful strategies in 23 of the 27 training sessions, across four different species of strategy. Geneva failed to discover strategies only when we heavily restricted its access to header fields, in an effort to explore a broader set of strategies (e.g., it failed to identify strategies when disallowed from accessing the entire TCP header). Below, we detail several successful strategies from each of the four species Geneva was able to discover against the GFW.

Species 1: TCB Desynchronization This species' strategies inject an insertion packet with a payload. The GFW treats the packet as legitimate, so the GFW advances the associated TCB, desynchronizing from the connection. Geneva quickly discovered this species; every subspecies emerged within the first three generations.

The most common way Geneva exploits this weakness is with a single outbound action-tree, triggered on `PSH/ACK` packets (which contain the censored keyword). For instance, Strategy 2 creates an insertion packet by duplicating the offensive packet, setting the TCP data offset to 10, and corrupting the checksum.

Strategy 2: TCB Desynchronization **98% (CN)**

```
[TCP:flags:PA]-duplicate(
    tamper{TCP:dataofs:replace:10}(
        tamper{TCP:chksum:corrupt}(send)),
    send)-| \/
```

Interestingly, this strategy sends the forbidden keyword *twice* (in both duplicates' payloads), seemingly increasing the likelihood of detection. Yet, neither request elicits a `RST` from the censor. Why?

The first packet invalidates the checksum, but this only causes the destination web server to ignore it, as the GFW does not verify checksums. The first packet also increases the `dataofs`. This field controls the size of the TCP header; increasing it causes a receiver to interpret the beginning of the payload as additional bytes in the TCP header. This is sufficient for the GFW to no longer identify the payload as an HTTP request, and thus it ignores the keyword, treats it as a legitimate part of the connection, and consequently desynchronizes from the connection. The censor therefore ignores the second packet altogether (the sequence number appears out of window), but the destination server accepts it.

Geneva also identifies seven other unique variants that exploit this issue using different combinations of header fields, operations, and action trees; these are available in Table 3.1.

Species 2: TCB Teardown This species' strategies inject an insertion packet with TCP flags to trigger a teardown of the GFW's associated TCB before sending the censored request. Once the TCB is torn down, the GFW ignores the connection's subsequent packets. Others have identified this species [23, 24], but Geneva has discovered new variants that reveal that the GFW works differently than suggested by prior work.

The most successful TCB Teardown strategy, shown in Strategy 3, has one outbound action-tree, triggered on `ACK` packets. It duplicates the `ACK`; it sends the first one unaltered, and turns the second one into a `RST` with a corrupted checksum before sending it. As with Strategy 2, the server ignores the `RST`, but the GFW does

not verify checksums and accepts the packet.

Strategy 3: TCB Teardown Variant 1	**95% (CN)**

```
[TCP:flags:A]-duplicate(send,
    tamper{TCP:flags:replace:R}(
        tamper{TCP:chksum:corrupt}(send)))-| \/
```

Through mutation, Geneva also found a variant of Strategy 3 that swaps the two packets: the corrupted RST is sent before the original ACK. This swap lowers the success rate to 51%. Through additional mutation, Geneva discovered Strategy 4, which improves this less successful variant by adding a second outbound action tree that corrupts ACK packets. This improves the success rate to 92%.

Strategy 4: TCB Teardown Variant 2	**92% (CN)**

```
[TCP:flags:A]-tamper{TCP:seq:corrupt}-|
[TCP:flags:A]-duplicate(
    tamper{TCP:flags:replace:R}(
        tamper{TCP:chksum:corrupt}(send)),
    send)-| \/
```

To understand why Strategy 4 works, recall that when multiple action trees fire on the same trigger, each is given a fresh copy of the original packet. Thus, the third and final packet sent in this strategy is the original, uncorrupted copy, and the three-way handshake is able to complete. The server ignores the other two, corrupted packets, but the GFW does not.

According to prior work [24], Strategies 3 and 4 *should not work* (at least, not nearly as well as they do). Prior work hypothesized that the GFW may enter a "resynchronization" state upon seeing a RST or RST/ACK packet [24]. In this case, once Strategy 4 sends the RST, the GFW should resynchronize the TCB on the next

packet in the datastream (the original ACK) and resume censoring the connection. If this were the case, then modifying Strategy 4 to move the first action tree (with the corrupted ACK) to the end of the outbound forest should be equally successful. However, this modification causes the strategy's success rate to plummet to 47%. Why?

These results indicate that the GFW is tracking the state of the TCP three-way handshake, and sometimes enters a resynchronization state *only* while the three-way handshake is unfinished. Concretely, we update the resynchronization state hypothesis as follows: upon receiving a RST or RST/ACK packet before the three-way handshake is complete, the GFW may enter the resynchronization state (about 50% of the time) instead of tearing down the TCB. Further, these strategies suggest that the GFW tracks the three-way handshake without paying attention to sequence numbers: the mere presence of an ACK packet is enough to fool the GFW into thinking that the three-way handshake is complete.

Geneva also lends insight into how the GFW processes RST packets. Consider Strategy 5:

Strategy 5: TCB Teardown with Invalid Flags	**96% (CN)**

```
[TCP:flags:A]-duplicate(send,
    tamper{TCP:flags:replace:FRAPUN}(
        tamper{IP:ttl:replace:10}(send))-| \/
```

FRAPUN is a completely invalid combination of TCP flags, and yet the strategy is still highly effective. We hypothesize that the GFW is looking only for the presence of a RST flag to teardown the TCB, and not validating that a legitimate combination

of flags is present in the packet. Table 3.1 shows variants of this strategy with many other invalid combinations of TCP flags.

Species 3: Segmentation This species' strategies take advantage of how the GFW mishandles TCP payloads that are segmented across multiple TCP packets.

The Segmentation species is fundamentally different than the *Data Reassembly* species from prior work [16]. Data Reassembly takes advantage of the censor's inability to differentiate which fragments or which data from fragments should be accepted. For instance, some such strategies extend one segment with junk data and overlap the second segment with the correct data. Prior work theorized that the GFW would accept the first packet to arrive with a specific IP fragment, but the *second* packet to arrive with a particular TCP segment [16]. Other Data Reassembly strategies leveraged this to inject insertion segments or fragments, tricking the GFW into accepting the wrong packet. Conversely, strategies from the Segmentation species exercise no IP fragmentation, no segment overlapping, and no inert packet injection—and can be performed from within an application, without raw sockets. Nonetheless, these are the only strategies Geneva has found to date that are highly successful across all three countries we experimented in.

Geneva has discovered two main Segmentation subspecies that are effective against the GFW. The first subspecies, shown in Strategy 6, segments the HTTP request (triggered on the `PSH/ACK`) at 8 bytes and corrupts packets with only the `ACK` flag set:

Corrupting the sequence number of the `ACK` packet breaks the original three-

Strategy 6: Segmentation with `ACK`	**94% (CN)**

```
[TCP:flags:PA]-fragment{tcp:8:True}(send,send)-|
[TCP:flags:A]-tamper{TCP:seq:corrupt}(send)-| \/
```

way handshake, but the `ACK` flag set in the `PSH/ACK` packet finishes the handshake. Table 3.1 lists additional variants.

One might expect that this strategy simply splits the forbidden word across multiple packets, and that the GFW must not be properly reassembling the segments. However, this is not the case. Our TCP payload is "`GET /?search=ultrasurf`": the first segment is "`GET /?se`" and the censored word appears in its entirety in the second segment. Changing the length of the censored word (e.g., to "`falun-gong`") does not affect the strategy's success rate.

Each component of Strategy 6 is required—for instance, it fails without the corrupted `ACK`—but it works surprisingly well even as many of the individual values vary. *Decreasing* the size of the first segment to anything less than 8 is equally effective, but *increasing* it to larger than 8 renders the strategy completely ineffective. The length of the HTTP parameter does not affect the strategy's success rate. As long as the sequence number is altered and the segmentation index is less than or equal to 8, the GFW seems insensitive to additional changes tried by strategy variants, such as corrupting both the sequence and acknowledgement numbers.

The second subspecies Geneva discovered is even stranger:

This strategy produces three segments, the first of size 8, the second of size 4, and the final containing the remainder of the original packet. Again, this does not segment the keyword: applying Strategy 7 to the original HTTP request results in

Strategy 7: Multi-segmentation	**98% (CN)**

```
[TCP:flags:PA]-
    fragment{tcp:8:True}(send,
        fragment{tcp:4:True}(send, send))-| \/
```

segments (1) "`GET /?se`", (2) "`arch`", and (3) "`=ultrasurf HTTP/1.1\r\nHost...`".

In a post-hoc analysis of this strategy, we explored different values for the segment offsets m and n ($m = 8$ and $n = 4$ in Strategy 7). We found that Strategy 7 works with near identical success rate so long as $0 < m \leq 8$, $m + n \geq 12$, and the second segment does not contain "`HTTP/1`". The strategy's effectiveness is also unaffected by the segment ordering.

Frankly, we do not yet fully understand *why* these strategies work. We hypothesize that this species exploits the GFW's inability to match or identify the packet as HTTP, but it is still unclear why Strategy 6 works; some interplay between how the GFW synchronizes its TCB after the three-way handshake also affects its ability to process segments.

The Segmentation species required significantly more generations to find than the previous two species. Strategy 6 emerged after 23 generations, and it required 4 more generations to achieve population convergence. Strategy 7 required 12 generations to identify. This implies that more nuanced strategies may simply require more generations to find, and there exists an opportunity to identify additional such strategies with a higher generation limit.

Overall, the Segmentation species is a significant departure from previously hand-developed strategies. Unlike almost all strategies from previous work [16,

23, 24, 84], Segmentation strategies do not require insertion packets, and can be deployed without raw sockets (let alone root privilege). Prior work has found that middleboxes can drop certain insertion packets [23, 24], and the requirement of root privilege may be a deployment barrier for some users. Thus, evasion strategies that can be deployed without insertion packets and without root privilege have an advantage of being more reliable and easier to deploy. Moreover, we believe it would be very challenging for a human to develop such a strategy as it exploits multiple instances of previously unknown dynamics with the GFW.

Species 4: Hybrid The final strategy Geneva discovered against the GFW is so distinct from other strategies that we classified it into its own species. The *Hybrid* species (Strategy 8) triggers on the HTTP request (the `PSH/ACK`). Before sending the original request, it sends a corrupted version, with the TCP flags set to `FIN` and the IP length set to `78`.

Strategy 8: Hybrid Species **53% (CN)**

```
[TCP:flags:PA]-
    duplicate(
        tamper{TCP:flags:replace:F}(
            tamper{IP:len:replace:78}(send)),
    send)-| \/
```

This is not a variant of TCB Teardown: injecting a `FIN` packet is not sufficient to trigger a teardown for the GFW [24]. Instead, this strategy actually causes a desynchronization in the GFW. Why?

Recall that checksums are calculated over the entire packet's data, but as the packet propagates, only the bytes within the specified packet length will be sent.

Thus, while the client sends a correct checksum, the subsequent hops will recompute the checksum as being different than what the client sent. In other words, the network assists in constructing a successful insertion packet.

The IP length change cuts the censored GET request at the `Host:` header, after the censored word appears. Like with the Segmentation species, this should be sufficient for the GFW to identify it as a censored HTTP request—indeed, if we remove the `FIN` flag, the strategy immediately fails. We hypothesize that the `FIN` packet carrying a payload induces the GFW to enter the resynchronization state, and causes it to resynchronize *immediately* on the current packet. This resynchronization behavior is unusual. We believe the GFW has made a special case for `FIN` packets with data (after one such packet in a connection, there are usually no further packets to resynchronize on). To test this, we instrumented a client to increase the sequence number of the valid copy of the forbidden request by the length of the injected packet payload (in this case, 38). The GFW tried to tear down this connection, confirming our hypothesis.

Although Geneva discovered this strategy with a fixed IP length (78), we find that any value works so long as only one HTTP header is included in the injected packet. We do not understand why this is the case. Our results suggest that the GFW has a separate processing pipeline when in the resynchronization state which differs from their regular protocol parsing. This allows us to exploit weaknesses in this specific code path. It is this secondary bug exploitation that makes this strategy a unique species.

This strategy also presents an interesting dilemma for the GFW as it pertains

to the resynchronization state. In examining the *TCB Teardown* variants that only succeeded 50% of the time, our results indicated that if the GFW were to enter the resynchronization state more frequently, they would be better protected from TCB attacks. However, this strategy demonstrates that it is not so simple: though increasing the likelihood of resynchronization worsens the performance of some of the *TCB Teardown* variants, it would improve the *Hybrid* variants.

3.3.3 Other Countries

To demonstrate Geneva's generalizability beyond China, we apply it to censors in two other countries: India and Kazakhstan.

India Our vantage points in India are within the Airtel ISP, specifically in Bangalore, which performs HTTP censorship by injecting a block page response if a request is made with a forbidden `Host:` header [86]. In our evaluation, we perform an HTTP GET request to a censored site (e.g., `pornhub.com`) from our vantage points, and consider the strategy to have failed if we receive the Airtel block page instead of the requested site. Airtel does not employ residual censorship, so we do avoid connections to blocked sites. Also, unlike the GFW, all of the strategies we tested either work 0% or 100% of the time against Airtel. Table 3.1 evaluates all strategies found from all of our vantage points against all three censors.

Geneva identified two broad species in India, both of which we believe are previously unknown.

First, Geneva discovered that Airtel is incapable of handling any invalid TCP

options; by adding invalid TCP options to requests, we can evade censorship completely. Geneva identified variants of this strategy using almost every available TCP option. We find that all the end-hosts we test ignore every option we add except `timestamp`, so this strategy does not damage the underlying TCP connection. Geneva also identifies additional subspecies that generate invalid options by controlling the `dataofs` field.

Second, Geneva found that Airtel is incapable of handling TCP segment reassembly; simply segmenting the request is sufficient for the connection to succeed. Similarly, Strategy 9 sends only a portion of the payload before sending the entire payload, thereby rendering the censor unable to identify the connection:

Strategy 9: Stutter Request **100% (IN)**

```
[TCP:flags:PA]-duplicate(
    tamper{IP:len:replace:64}(send),
    send)-|
```

Collectively, we find these evasion strategies to be much simpler than those required to evade China's GFW. Indeed, Geneva did not identify any strategies in India resembling the *TCB Teardown* strategy, and many of the strategies that take advantage of the increased complexity of the GFW do not work against Airtel.

Kazakhstan Starting on July 17, 2019, Kazakhstan began intercepting HTTPS connections to many social media sites using a fake root certificate [88]. Though this interception has fortunately since ended [89], we deployed Geneva against the system while it was active. To perform strategy evaluation, we sent an SNI request with a targeted hostname (such as `facebook.com`) to HTTPS servers hosted in

Kazakhstan within the affected region. We consider the strategy to have failed if our client receives the injected certificate; if we receive the correct certificate, we consider it a success.

Within 4 hours, Geneva discovered three successful species.

Similar to Airtel's censorship, we find that Kazakhstan's HTTPS MITM cannot process TCP segmentation; segmenting the targeted SNI request is sufficient alone to evade the MITM.

Geneva discovered a second species that was originally manually developed (and is now extinct) against the GFW: the *TCB Turnaround* (Strategy 1), which sends a `SYN/ACK` before the `SYN` to make the censor believe the roles of client and server are reversed.

Geneva also identified strategies that resemble *TCB Desynchronization*, though they are simpler than the desynchronization strategies Geneva found against the GFW. As shown in Strategy 10, simply sending a second `SYN` packet with a payload circumvents the MITM with 100% success rate. All of the other desynchronization attacks learned against the GFW also worked (see Table 3.1).

Strategy 10: Simple TCB Desynchronization **100% (KZ)**

```
[TCP:flags:S]-duplicate(send,
  tamper{TCP:load:corrupt}(send,))-|
```

As with India, strategies to evade Kazakhstan's MITM attack are less sophisticated and easier for Geneva to find than the GFW. These results show that Geneva is capable of attacking diverse censorship systems and can apply broadly.

3.3.4 Training Defunct Strategies

Extinct Strategies In addition to deriving new strategies, we also tried multiple strategies in now-extinct species and subspecies suggested by previous works against the GFW. We find the *TCB Creation* species to be extinct; Geneva was unable to find any functional strategies that create a new TCB. In manual testing, we also found that strategies that relied on this species from former work no longer work, and even improved versions of this strategy, such as *TCB Creation + Resync/Desync* [24] do not work against the GFW. This includes related subspecies, such as the *TCB Turnaround* [24].

TCB Teardown using a `FIN` or `FIN/ACK` packet [24] seems to be similarly extinct: the only successful TCB Teardown strategies that Geneva identified required the `RST` flag to be set to successfully function. We also find the *Data Reassembly* (as defined by previous works) species to be largely extinct. This finding also confirms results from previous work [24], which found that IP fragment ordering strategies were no longer effective against the GFW. However, given the nuance of the *Segmentation* species, we hesitate to definitively rule out any species as fully extinct.

Seeded Training We next experimented with how Geneva could cope with changing firewall rules in the real world. For this experiment, we seeded the evolution using the extinct *TCB Creation + Resync/Desync* strategy [24] against the GFW. Seeding the evolution spawns the initial population pool using copies of this strategy instead of a randomly initialized pool. It takes just 4 generations for the first set of

new functional strategies to emerge, and within 15 generations, a sizable population of *TCB Desynchronization* strategies emerged. In a second experiment, it takes just 2 generations to derive various less successful subspecies of *TCB Teardown*, and a further 6 to hone it to a fully reduced, effective strategy. This demonstrates that even if a species has achieved full population saturation and the GFW updates to make them go extinct, Geneva is capable of pivoting to find new successful strategies.

3.4 Discussion

Is Geneva Necessary? Would it be possible to realize Geneva-like functionality with less complexity? One alternative would be to simply enumerate the *entire* space of packet manipulations. Unfortunately, this is infeasible; INTANG [24] presents a strategy ("TCB Creation + Resync/Desync") that would require a Geneva action tree of size nine to represent. However, because Geneva can support modifications to *all* IP and TCP fields (including multiple TCP options), there are a huge number of potential action trees. We conservatively estimate[2] that there are 2^{89} functionally distinct Geneva trees of size nine.

Alternatively, we could ostensibly try to distill down the lessons that Geneva learns and use them to manually craft rules to guide strategy generation. However, this is unnecessary (Geneva learns these lessons by itself), and worse yet, it introduces *bias*: if we were to encode how we *believe* the censor's implementation of TCP works into how Geneva searches the space of solutions, we would not allow Geneva to find

[2]In this under-estimate, we assume that tampering with identifier fields (e.g., `seq`, `chksum`) can only take one of two values: correct, or incorrect, and cardinal fields (e.g. `dataofs`) can take on only one of three values: too-small, too-large, or just-right.

unintuitive strategies or bugs in the censor's implementation.

It is possible that there is another form of machine learning that is more accurate or more efficient than Geneva's use of genetic algorithms. Exploring these alternatives is beyond the scope of this chapter—my primary goal to support my thesis is to show that the problem *can* be automated, and to discover strategies manual efforts have not.

Censor Countermeasures We envision two broad ways in which censors can react to Geneva. First and foremost, they can fix their systems. For implementation bugs, this may be a simple matter—in fact, they may use Geneva themselves to find bugs prior to deployment. More difficult to repair, however, are errors the censors make in their underlying assumptions. For example, the TCB Teardown strategies exploit the GFW's shortcut of tearing down TCBs to save state; fixing this may introduce significant computational overhead.

Second, censors could try to detect and thwart Geneva itself, for instance, by detecting its training packets, and poisoning our datasets by making strategies appear (not) to work. Geneva tampers with packets in random ways, often resulting in strange combinations of flags that would be easy to detect, like `FRAPUN` in Strategy 5. Geneva could be modified to avoid this, for instance by constraining its mutations or by punishing "detectability" in the fitness function.

We see these as logical conclusions to the ongoing censorship arms race: eventually, censors will either have to fully patch their system (which seems costly) or thwart future efforts to probe their systems (which seems infeasible). Geneva's

automation speeds us to these ends. I discuss these countermeasures (and the difficulties in implementing them in practice) in greater depth in Chapter 10.

Limitations of Our Evaluation We did not evaluate our system on as many vantage points in China as some prior work [23, 24] because, since those studies, China has made it significantly more difficult for non-Chinese residents to rent machines in mainland China. Obtaining the vantage points we had required considerable effort. The difficulty with which to run these experiments also limits the ease with which the results can be reproduced, a limitation that unfortunately applies to all work in the space of nation-state censorship evasion. We find this trend concerning, and caution users to fully understand the risks before undertaking similar studies. Nonetheless, by applying Geneva in three fundamentally different censoring regimes, we have shown it generalizes, and expect it would be applicable to other vantage points in these countries, as well.

Ethical Considerations We designed Geneva to have minimal impact on other hosts. To the best of our knowledge, the state of one host's TCP connections does not affect the connections of other hosts. Geneva was designed not to spoof IP addresses or ports, and our interactions with the GFW should have had no impact on any other users. Moreover, we designed Geneva to evaluate strategies serially, which effectively limits the rate at which it creates TCP connections and sends data, mitigating any impact it may have had on other hosts on the same network.

Beyond these traditional concerns of evaluating systems on shared infrastructure, there are also ethical concerns with evaluating in a censoring regime. Similar

to some prior work [16, 23, 24], we evaluated Geneva by running it solely on hosts that we rented and controlled—as opposed to recruiting unwitting users [90]—to mitigate ethical concerns.

3.5 Conclusion

There has long been a cat-and-mouse game between censors and a community of researchers and practitioners who seek to evade them. The current evade-detect cycle requires extensive *manual* measurement, reverse-engineering, and creativity to obtain new means of censorship evasion. In this chapter, I presented Geneva, a genetic algorithm for automatically discovering censorship evasion strategies against network censors. Through evaluation both in-lab and against the GFW, I have demonstrated that Geneva can *efficiently* discover strategies, and that its genetic building blocks allow it to both re-derive all previously published schemes that it can support, and derive altogether new strategies that prior work posited would not be effective. Geneva supports my thesis and shows that middleboxes can automatically be rendered ineffective from the client-side. Geneva represents an important first step towards automating censorship evasion, and to this end, I have made the code publicly available at `https://geneva.cs.umd.edu`.

In the next chapter, I will extend Geneva to support my thesis across multiple protocols and in a brand-new deployment context: evading censorship from the server-side.

<table>
<tr><th rowspan="2">Species</th><th rowspan="2">Subspecies</th><th rowspan="2">Variant</th><th rowspan="2">Genetic Code</th><th colspan="3">Success Rate</th></tr>
<tr><th>CN</th><th>IN</th><th>KZ</th></tr>
<tr><td>None</td><td>None</td><td>None</td><td>\/</td><td>3%</td><td>0%</td><td>0%</td></tr>
<tr><td rowspan="10">TCB Desync</td><td rowspan="5">Inc. Dataofs</td><td>Corrupt Chksum</td><td>[TCP:flags:PA]-duplicate(tamper{TCP:dataofs:replace:10}
(tamper{TCP:chksum:corrupt},),)-|</td><td>98%</td><td>0%</td><td>100%</td></tr>
<tr><td>Small TTL</td><td>[TCP:flags:PA]-duplicate(tamper{TCP:dataofs:replace:10}
(tamper{IP:ttl:replace:10},),)-|</td><td>98%</td><td>0%</td><td>100%</td></tr>
<tr><td>Invalid Flags</td><td>[TCP:flags:PA]-duplicate(tamper{TCP:dataofs:replace:10}
(tamper{TCP:flags:replace:FRAPUN},),)-|</td><td>26%</td><td>0%</td><td>100%</td></tr>
<tr><td>Corrupt Ack</td><td>[TCP:flags:PA]-duplicate(tamper{TCP:dataofs:replace:10}
(tamper{TCP:ack:corrupt},),)-|</td><td>94%</td><td>0%</td><td>100%</td></tr>
<tr><td>Corrupt WScale</td><td>[TCP:flags:PA]-duplicate(tamper{TCP:options-wscale:corrupt}
(tamper{TCP:dataofs:replace:8},),)-|</td><td>98%</td><td>0%</td><td>100%</td></tr>
<tr><td rowspan="3">Inv. Payload</td><td>Corrupt Chksum</td><td>[TCP:flags:PA]-duplicate(tamper{TCP:load:corrupt}
(tamper{TCP:chksum:corrupt},),)-|</td><td>80%</td><td>0%</td><td>100%</td></tr>
<tr><td>Small TTL</td><td>[TCP:flags:PA]-duplicate(tamper{TCP:load:corrupt}
(tamper{IP:ttl:replace:8},),)-|</td><td>98%</td><td>0%</td><td>100%</td></tr>
<tr><td>Corrupt Ack</td><td>[TCP:flags:PA]-duplicate(tamper{TCP:load:corrupt}
(tamper{TCP:ack:corrupt},),)-|</td><td>87%</td><td>0%</td><td>100%</td></tr>
<tr><td>Simple</td><td>Payload SYN</td><td>[TCP:flags:S]-duplicate(,tamper{TCP:load:corrupt})-|</td><td>3%</td><td>0%</td><td>100%</td></tr>
<tr><td>Stutter Request</td><td>Stutter Request</td><td>[TCP:flags:PA]-duplicate(tamper{IP:len:replace:64},)-|</td><td>3%</td><td>100%</td><td>0%</td></tr>
<tr><td rowspan="20">Teardown</td><td rowspan="6">With RST</td><td rowspan="2">Corrupt Chksum</td><td>[TCP:flags:A]-duplicate(,tamper{TCP:flags:replace:R}
(tamper{TCP:chksum:corrupt},))-|</td><td>95%</td><td>0%</td><td>0%</td></tr>
<tr><td>[TCP:flags:A]-duplicate(tamper{TCP:flags:replace:R}
(tamper{TCP:chksum:corrupt},),)-|</td><td>51%</td><td>0%</td><td>0%</td></tr>
<tr><td rowspan="2">Small TTL</td><td>[TCP:flags:A]-duplicate(,tamper{TCP:flags:replace:R}
(tamper{IP:ttl:replace:10},))-|</td><td>87%</td><td>0%</td><td>0%</td></tr>
<tr><td>[TCP:flags:A]-duplicate(tamper{TCP:flags:replace:R}
(tamper{IP:ttl:replace:9},),)-|</td><td>52%</td><td>0%</td><td>0%</td></tr>
<tr><td rowspan="2">Inv. md5Header</td><td>[TCP:flags:A]-duplicate(,tamper{TCP:options-md5header:corrupt}
(tamper{TCP:flags:replace:R},))-|</td><td>86%</td><td>0%</td><td>0%</td></tr>
<tr><td>[TCP:flags:A]-duplicate(tamper{TCP:options-md5header:corrupt}
(tamper{TCP:flags:replace:RA},),)-|</td><td>44%</td><td>0%</td><td>0%</td></tr>
<tr><td rowspan="8">With RST/ACK</td><td rowspan="2">Corrupt Chksum</td><td>[TCP:flags:A]-duplicate(,tamper{TCP:flags:replace:RA}
(tamper{TCP:chksum:corrupt},))-|</td><td>80%</td><td>0%</td><td>0%</td></tr>
<tr><td>[TCP:flags:A]-duplicate(tamper{TCP:flags:replace:RA}
(tamper{TCP:chksum:corrupt},),)-|</td><td>66%</td><td>0%</td><td>0%</td></tr>
<tr><td rowspan="2">Small TTL</td><td>[TCP:flags:A]-duplicate(,tamper{TCP:flags:replace:RA}
(tamper{IP:ttl:replace:10},))-|</td><td>94%</td><td>0%</td><td>0%</td></tr>
<tr><td>[TCP:flags:A]-duplicate(tamper{TCP:flags:replace:RA}
(tamper{IP:ttl:replace:10},),)-|</td><td>57%</td><td>0%</td><td>0%</td></tr>
<tr><td rowspan="2">Inv. md5Header</td><td>[TCP:flags:A]-duplicate(,tamper{TCP:options-md5header:corrupt}
(tamper{TCP:flags:replace:R},))-|</td><td>94%</td><td>0%</td><td>0%</td></tr>
<tr><td>[TCP:flags:A]-duplicate(tamper{TCP:options-md5header:corrupt}
(tamper{TCP:flags:replace:R},),)-|</td><td>48%</td><td>0%</td><td>0%</td></tr>
<tr><td rowspan="2">Corrupt Ack</td><td>[TCP:flags:A]-duplicate(tamper{TCP:flags:replace:RA}
(tamper{TCP:ack:corrupt},),)-|</td><td>43%</td><td>0%</td><td>0%</td></tr>
<tr><td>[TCP:flags:A]-duplicate(,tamper{TCP:flags:replace:RA}
(tamper{TCP:ack:corrupt},))-|</td><td>31%</td><td>0%</td><td>0%</td></tr>
<tr><td rowspan="6">Invalid Flags</td><td rowspan="2">Corrupt Chksum</td><td>[TCP:flags:A]-duplicate(,tamper{TCP:flags:replace:FRAPUEN}
(tamper{TCP:chksum:corrupt},))-|</td><td>89%</td><td>0%</td><td>0%</td></tr>
<tr><td>[TCP:flags:A]-duplicate(tamper{TCP:flags:replace:FRAPUEN}
(tamper{TCP:chksum:corrupt},),)-|</td><td>48%</td><td>0%</td><td>0%</td></tr>
<tr><td rowspan="2">Small TTL</td><td>[TCP:flags:A]-duplicate(,tamper{TCP:flags:replace:FREACN}
(tamper{IP:ttl:replace:10},))-|</td><td>96%</td><td>0%</td><td>0%</td></tr>
<tr><td>[TCP:flags:A]-duplicate(tamper{TCP:flags:replace:FRAPUEN}
(tamper{IP:ttl:replace:10},),)-|</td><td>56%</td><td>0%</td><td>0%</td></tr>
<tr><td rowspan="2">Inv. md5Header</td><td>[TCP:flags:A]-duplicate(,tamper{TCP:flags:replace:FRAPUN}
(tamper{TCP:options-md5header:corrupt},))-|</td><td>94%</td><td>0%</td><td>0%</td></tr>
<tr><td>[TCP:flags:A]-duplicate(tamper{TCP:flags:replace:FRAPUEN}
(tamper{TCP:options-md5header:corrupt},),)-|</td><td>55%</td><td>0%</td><td>0%</td></tr>
<tr><td rowspan="3">Segmentation</td><td>With ACK</td><td>Offsets</td><td>[TCP:flags:PA]-fragment{tcp:8:False}-|
[TCP:flags:A]-tamper{TCP:seq:corrupt}-|</td><td>94%</td><td>100%</td><td>100%</td></tr>
<tr><td>Reassembly</td><td>Offsets</td><td>[TCP:flags:PA]-fragment{tcp:8:True}(,fragment{tcp:4:True})-|</td><td>98%</td><td>100%</td><td>100%</td></tr>
<tr><td>Simple</td><td>In-Order</td><td>[TCP:flags:PA]-fragment{tcp:-1:True}-|</td><td>3%</td><td>100%</td><td>100%</td></tr>
<tr><td>Hybrid</td><td>With FIN</td><td>Cut Header</td><td>[TCP:flags:PA]-duplicate(tamper{TCP:flags:replace:F}
(tamper{IP:len:replace:78},),)-|</td><td>53%</td><td>100%</td><td>0%</td></tr>
<tr><td>TCB Turnaround</td><td>TCB Turnaround</td><td>TCB Turnaround</td><td>[TCP:flags:S]-duplicate(tamper{TCP:flags:replace:SA},)-|</td><td>3%</td><td>0%</td><td>100%</td></tr>
<tr><td>Invalid Options</td><td>Invalid Options</td><td>Corrupt UTO</td><td>[TCP:flags:PA]-tamper{TCP:options-uto:corrupt}-|</td><td>3%</td><td>100%</td><td>0%</td></tr>
</table>

Table 3.1: Species, subspecies, and variants Geneva found (with success rates) against the GFW. For readability, we omit all "`send`"s from the genetic code (e.g., `duplicate(,)` is equivalent to `duplicate(send,send)`). This is correct, syntactic sugar for Geneva.

Censor behavior	Learned strategy to defeat
1. Synchronizes TCB on the first SYN only; sends RSTs only to the client if a censored word appears anywhere in any packet and a matching TCB exists.	Drop inbound RST packets.
2. Synchronizes TCB on the first SYN only; sends RSTs to the client and server if a censored word appears anywhere in any packet and a matching TCB exists.	Inject a SYN packet with a different sequence number.
3. Synchronizes TCB on the first SYN only, drops all future client/server communication if a censored word appears anywhere in any packet and a matching TCB exists.	Inject a SYN packet with a different sequence number.
4. Synchronizes TCB on SYN and ACK packets; sends RSTs to the client and server if a censored word appears anywhere in any packet and a matching TCB exists.	Inject an insertion ACK packet with a different sequence number after the 3-way handshake.
5. Synchronizes TCB on SYN, and resynchronizes periodically every few packets packets; sends RSTs to the client and server if a censored word appears anywhere in any packet and a matching TCB exists.	Inject an insertion ACK packet with a different sequence number after the 3-way handshake.
6. Synchronizes TCB using only IP addresses on SYN and SYN/ACK; sends RSTs to the client and server if a censored word appears anywhere in an HTTP header or packet payload unless TCB is torn down.	Inject an insertion RST packet after the 3-way handshake, or induce the server to send a RST on another port.
7. Synchronizes TCB using only IP/port tuples on SYN and SYN/ACK; sends RSTs only to the client if a censored word appears anywhere in any packet unless TCB is torn down.	Inject an insertion RST packet after the 3-way handshake.
8. Synchronizes TCB on SYN, SYN/ACK, and ACK; sends RSTs only to the client if a censored word appears anywhere in any packet unless TCB is torn down.	Inject an insertion RST packet after the 3-way handshake.
9. Synchronizes TCB on SYN and ACK; sends RSTs only to the client if a censored word appears anywhere in any packet, and enters a resynchronization state on any RST or FIN packet.	Inject an insertion RST or FIN after the 3-way handshake, and then send a followup insertion packet with a different sequence number.
10. Synchronizes TCB on SYN, only processes packets with correct checksums; sends RSTs only to the client if a censored word appears anywhere in any packet, and enters a resynchronization state on any RST or FIN packet.	Inject an insertion RST packet after the 3-way handshake using a non-checksum insertion mechanism (e.g., low TTL), immediately followed by another insertion packet with an incorrect sequence number.
11. Synchronizes TCB on SYN, only processes packets with correct checksums, lengths, and data offsets; sends RSTs only to the client if a censored word appears anywhere in any packet, and enters a resynchronization state on any valid RST or FIN packet.	Inject an insertion RST packet after the 3-way handshake using a low TTL, immediately followed by another insertion packet with an incorrect sequence number.

Table 3.2: Mock censors developed for in-lab training, and strategies Geneva learned to defeat them.

Species	Strategy	Found?			
		[16]	[23]	[24]	Geneva
TCB Creation	w/ low TTL	✓		✓	✓
	w/ corrupt checksum			✓	✓
	(Improved) and Resync/Desync			✓	✓
TCB Teardown	w/ `RST` and low TTL	✓	✓	✓	✓
	w/ `RST` and corrupt checksum		✓	✓	✓
	w/ `RST` and invalid timestamp			✓	✓
	w/ `RST` and invalid MD5 Header			✓	✓
	w/ `RST/ACK` and corrupt checksum			✓	✓
	w/ `RST/ACK` and low TTL	✓	✓	✓	✓
	w/ `RST/ACK` and invalid timestamp			✓	✓
	w/ `RST/ACK` and invalid MD5 Header			✓	✓
	w/ `FIN` and low TTL	✓		✓	✓
	w/ `FIN` and corrupt checksum			✓	✓
	(Improved)			✓	✓
	and TCB Reversal			✓	✓
Reassembly	TCP Segmentation w/ out of order data		✓	✓	✓
	Overlapping fragments	✓		✓	✓
	Overlapping segments	✓		✓	✓
	In-order data w/ low TTL			✓	✓
	In-order data w/ corrupt `ACK`	✓		✓	✓
	In-order data w/ corrupt checksum			✓	✓
	In-order data w/ no TCP flags			✓	✓
	Out-of-order data w/ IP fragments			✓	✓
	Out-of-order data w/ TCP segments			✓	✓
	(Improved) In-order data overlapping			✓	✓
	Payload splitting		✓		✓
	Payload reordering		✓		✓
Traffic Misclassification	Inert Packet Insertion w/ corrupt checksum		✓		✓
	Inert Packet Insertion w/o `ACK` flag		✓		✓
State Exhaustion	Send $>$ 1KB of traffic	✓			
	Classification Flushing – Delay	✓	✓		
HTTP Incompleteness	$>$ 1 space between method and URI	✓			
	Keyword at location $>$ 2048	✓			
	Keyword in 2nd or higher of multiple requests in one segment	✓			
	URL encoding (except %-encoding)	✓			

Table 3.3: Prior work's effective TCP-based strategies and whether Geneva re-derived the strategy in the lab or in the wild, regardless of whether the strategy is still effective. Note that Geneva had no knowledge of HTTP fields and could not introduce delays into the request.

Chapter 4: Server-side Evasion

In the previous chapter, I demonstrated that it is possible to automatically discover censorship evasion strategies that run purely at the client, but this left open a critical question: Do all censorship evasion strategies have to run at the client, or could servers evade censorship on clients' behalves? Indeed, I am aware of no prior censorship evasion that runs purely server-side. In this chapter, I show that server-side evasion is indeed possible, and that it can be used to evade multiple protocols (HTTP, HTTPS, DNS, and more). My results from training against many protocols also exposes new insights into the designs and deployments of censorship infrastructures.

For a client inside a censoring regime to access censored content, it seems quite natural that the client would have to deploy *something*. Indeed, to the best of our knowledge, *all* prior work in censorship evasion has required some degree of deployment at the clients within the censoring regime. Proxies [65, 91], decoy routing [21, 47], VPNs, anonymous communication protocols [20], domain fronting [58], protocol obfuscation [26, 48, 49], and recent advances that confuse censors by manipulating packets [16, 23, 24, 40]—all of these prior solutions require various degrees of active participation on behalf of clients.

Unfortunately, active participation on the part of clients can limit the reach of censorship evasion techniques. In some scenarios, installing anti-censorship software can put users at risk [92]. For users who are willing to take on this risk, it can be difficult to *bootstrap* censorship evasion, as the anti-censorship tools themselves may be censored [93, 94]. Worse yet, there are many users who do not seek out tools to evade censorship because they do not even know they are being censored [95].

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

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

In this chapter, I present the first purely server-side censorship evasion strategies—11 in total, spanning four countries (China, India, Iran, and Kazakhstan). Like a recent string of papers [16, 23, 24, 40], these strategies do not involve a custom protocol, but rather operate by manipulating packets of existing applications, e.g., by inserting, duplicating, tampering, or dropping packets. We verify that each of these strategies (sometimes with small tweaks) work with completely unmodified clients running any major operating system.

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

- Server-side-only circumvention strategies are possible! We succeeded in finding them in every country we tested (China, India, Iran, and Kazakhstan) and for all of the protocols we were able to trigger censorship with (DNS-over-TCP, FTP, HTTP, HTTPS, and SMTP).
- The so-called Great Firewall (GFW) of China has a more nuanced "resynchronization state" than previously reported [24, 40].
- China uses *different network stacks* for each of the protocols that it censors; circumvention strategies that work for one application-layer protocol (e.g., HTTPS)

do not necessarily work for another (e.g., HTTP or SMTP).

The rest of this chapter is organized as follows. §4.1 empirically shows that, unfortunately, client-side techniques do not generalize to server-side. §4.2 presents our experiment methodology. We present 11 new server-side evasion strategies in §4.3, and through further examination, shed new light on the inner workings of censorship in China, India, Iran, and Kazakhstan. §4.4 explores our theory that censors employ different network stacks for each censored application. §4.5 shows that our server-side strategies work for a wide diversity of client OSes. We discuss deployment considerations in §4.6 and ethical considerations in §4.7. Finally, §4.8 concludes this chapter.

4.1 Client-Side Strategies do not Generalize

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

Prior work has identified a wealth of client-side strategies for circumventing censorship. Some of these strategies are tailored specifically to the client; for instance, "Segmentation" strategies split up a client's HTTP GET request across multiple TCP packets, exploiting an apparent bug in some censors' packet reassembly code [40]. However, other client-side strategies appear as if they would work from the server, as well. For example, a seminal circumvention strategy has the client send a TCP `RST` with a TTL large enough to reach the censor but too small to reach the server [16, 23, 24, 40, 57]. As a result of this strategy, the censor believes

the connection has been torn down and thus pays no attention to future packets from that connection, allowing the client to send requests that would have otherwise been censored. *Should such strategies not also work from the server?*

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

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

Unfortunately, *none* of these strategies worked when run server-side. This is surprising: many of the "TCB Teardown" strategies described in the previous chapter involve the client sending tear-down packets (insertion packets with `RST` or `RST+ACK` flags) immediately after receiving the server's `SYN+ACK`; these server-side analogs also send tear-down packets immediately after the `SYN+ACK`, the only difference being that they come from the server. We considered the possibility that network delays were causing the server's tear-down packets to arrive at the censor

after the client's censored query[1]. To account for this, we instrumented our client to delay sending its query until it received the insertion packets, but this was also unsuccessful at evading censorship.

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

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

4.2 Server-side Methodology

In this section, I describe my methodology in deploying Geneva, data collection, and experimentation.

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

4.2.1 Geneva Extensions

New Protocols Geneva's initial design was initially applied only to HTTP. In this chapter, I show that Geneva can be applied to be able to train over a variety of applications across a variety of protocols. Specifically, I added support for DNS-over-TCP, FTP, HTTPS, and SMTP.

Non-additions I also explored applying server-side evasion to Tor Bridges and Telegram MTProxy servers [96, 97]. Although Tor and Telegram are both blocked at the IP and DNS level, as of time of writing, I was unable to trigger active probing to private unpublished Tor bridges or MTProxies. The Tor team is aware that Tor did not trigger active probing as of time of writing, and these findings are consistent with recent reports [24, 40]. We focus our efforts on the protocols that are getting censored now, and we leave a deeper exploration of server-side training over other anti-censorship protocols to later work.

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

We configured Geneva to initialize each population pool with 300 individuals, and allowed evolution to take place for 50 generations, or until population convergence occurs. Although Geneva is capable of evolving not only how it manipulates packets but also *which* packets it triggers on, we observed that for DNS-over-TCP,

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

Table 4.1: Client locations and protocols used in our experiments.

HTTP, HTTPS, and SMTP, the only packet the server could trigger on before a censorship event was the SYN+ACK packet. Thus, as a slight optimization, for these protocols, we restricted Geneva to only be able to trigger on SYN+ACKs.

4.2.2 Data Collection Methodology

Over the span of five months, we ran Geneva server-side in six countries—Australia, Germany, Ireland, Japan, South Korea, and the US—on five protocols: DNS (over TCP), FTP, HTTP, HTTPS, and SMTP (all over IPv4). We used unmodified clients within four nation-state censors—China, India, Iran, and Kazakhstan—to connect to our servers. For each nation-state censor, we trained on each protocol for which we were able to trigger censorship; all four countries censored HTTP, but only China censored all six protocols.[2] Table 4.1 shows the client locations and protocols we used throughout our experiments. Within each censored regime, we find no significant difference in strategy effectiveness across the different vantage points or external servers.

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

[2]Contrary to the findings by Aryan et al. [55], we find that Iran no longer censors DNS-over-TCP at all.

- *DNS-over-TCP (China)*: We make a censored request with an unmodified DNS client to open resolvers (Google and Cloudflare), as well as resolvers we control outside China.
- *FTP (China)*: We sign into FTP servers we control and issue requests for files with sensitive keywords as names (e.g., `ultrasurf`).
- *HTTP (all countries)*: In China, we issue GET requests with a censored keyword in the URL parameters (for instance, `?q=ultrasurf`). In India, Iran, and Kazakhstan, we issue GET requests with a blacklisted website in the `Host:` header.
- *HTTPS (China and Iran)*: We perform a TLS handshake with a forbidden URL (e.g., `youtube.com` in Iran and `www.wikipedia.org` in China) in the Server Name Indication (SNI) field.
- *SMTP (China)*: We connect to SMTP servers we control and, from our unmodified clients, send an email to a forbidden email address, `xiazai@upup.info` [98].

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

Residual Censorship In China, we observe that different protocols are handled differently by the GFW. For example, over HTTP, the GFW has *residual censorship*: for approximately 90 seconds after a forbidden request is censored, all TCP requests to the server IP and port elicit tear-down packets from the GFW immediately following the three-way handshake. Prior work has documented the existence

of residual censorship in some cases for HTTPS; however, we do not observe this behavior from any of our vantage points during our experiments and confirm that as of time of writing, HTTPS residual censorship is not active in China. Further, we do not observe this behavior from any of our vantage points in China for SMTP, DNS-over-TCP, or FTP; after the forbidden request on these protocols is censored, the user is free to make a second follow-up request immediately. I will report on more specific dynamics of residual censorship later in this dissertation; for this chapter, residual censorship is primarily relevant towards informing the methodology.

Evasion Success Rates It has been shown that, somewhat surprisingly, some packet-manipulation strategies succeed only *some* of the time; for instance, in the previous chapter, we found some client-side strategies that work roughly 50% of the time. Throughout this chapter, we present the success rates of the various strategies Geneva has found. For DNS in particular, this requires some special consideration, because, according to RFC 7766 [99] on DNS-over-TCP: *DNS clients SHOULD retry unanswered queries if the connection closes before receiving all outstanding responses. No specific retry algorithm is specified in this document.* Censorship by the GFW qualifies as a premature connection close, and thus results in retries, but the RFC leaves the exact number of retries up to the implementer. This serves to greatly improve the success rates of any server-side strategies for DNS-over-TCP: even if the strategy works only 50% of the time, with just 2 retries (3 total queries), the success rates will improve to 87.5%.

We have found that, in practice, applications choose different numbers of DNS

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

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

4.3 Server-Side Results

Here, we detail newly discovered strategies that defeat censors from the server-side. Table 4.2 summarizes our results across all countries (China, India, Iran, and Kazakhstan) and applications (DNS-over-TCP, FTP, HTTP, HTTPS, and SMTP).

4.3.1 Server-side Evasion in China

We applied Geneva from the server side against the GFW across DNS, FTP, SMTP, HTTP, and HTTPS. Geneva identified 8 distinct server-side only strategies that are successful at least 50% of the time for at least one protocol in China: 4 for

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

Table 4.2: Summary of server-side-only strategies and their success rates. All of these strategies manipulate only TCP, and yet, against China's GFW, their success rates are application-dependent. Kazakhstan's HTTPS and Iran's DNS-over-TCP censorship infrastructure are currently inactive.

DNS, 5 for FTP, 1 for SMTP, 4 for HTTP, and 2 for HTTPS. We provide packet waterfall diagrams in Figure 4.1 which show the resulting server- and client-behaviors when the strategies are run. Although the strategies require *no client-side modifications whatsoever*, they induce client-side behavior that assists in circumventing censorship. In the rest of this subsection, we explore each of these strategies, explain why they work, and describe what they teach us about China's GFW.

Strategy 11: Simultaneous Open, Injected `RST` (China)
DNS (89%), FTP (52%), HTTP (54%), HTTPS (14%), SMTP (70%)

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

Simultaneous Open Strategy 11 triggers on outbound `SYN+ACK` packets. Instead

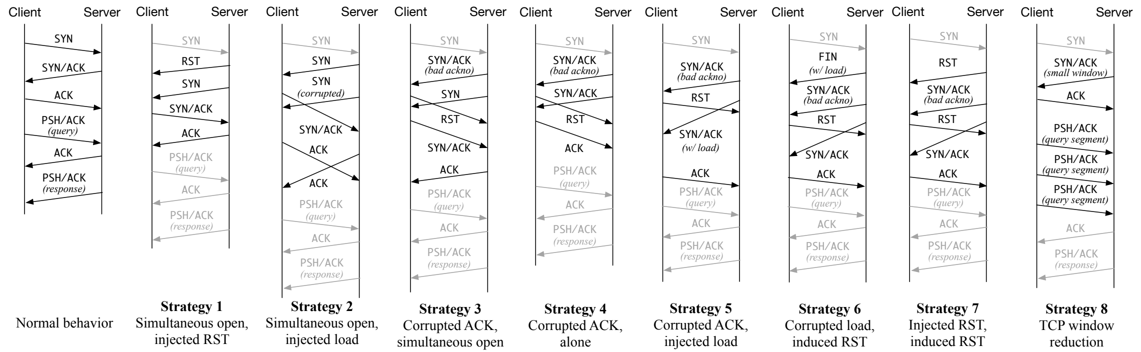


Figure 4.1: Server-side evasion strategies in China. All of the strategies work without modifications to the client, and yet they induce client-side behavior that helps circumvent censorship. (Standard packets at the beginning and the end are grayed out to emphasize the critical differences from normal behavior.)

of sending the SYN+ACK, it replaces it with two packets—a RST and a SYN—and sends them instead. How does an unmodified client respond to this strange sequence of packets?

First, the RST packet is actually ignored by the client, because it does not have the ACK flag set and the TCP connection is not yet in a synchronized state. Despite RFC 793 [100] suggesting that the connection be torn down, we find that in practice, TCP implementations across all modern operating systems ignore this RST. Second, the injected SYN packet serves to initiate *TCP simultaneous open*.

RFC 793 [100] requires TCP implementations to support simultaneous open. Originally, simultaneous open was meant to occur when two hosts attempt to open a connection by sending SYN packets to each other at the same time. However, a server can simulate simultaneous open by responding to a SYN packet from the client with a SYN packet of its own. To the client, this resembles simultaneous open, since the client receives a SYN packet, and therefore must respond with a SYN+ACK packet. This strategy employs simultaneous open by first sending an inert RST packet, then

by setting up the connection with a SYN packet.

When used for HTTP, Strategy 11 has a success rate of 54%. We see similar success rates for FTP and for each single DNS-over-TCP query (recall that DNS will try up to 3 times).

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

Prior work has hypothesized that the presence of a RST packet during the three-way handshake can put the GFW in a resynchronization state with about 50% probability [24, 40]. Therefore, we expect the injected RST packet not to tear down the connection, but instead to put the GFW into the resynchronization state. Wang et al. hypothesized that the only packets sent by the server that the GFW resynchronizes on are SYN+ACK packets, so the next packet for the GFW to resynchronize on is the SYN+ACK packet *sent by the client.* At this point, the GFW should just properly resynchronize onto our connection—but it does not. Why?

When TCP simultaneous open is performed, the sequence number does not

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

To test this theory, we instrumented a client-side request to decrement the sequence number of the forbidden request by 1 while the strategy is run on the server side. If the theory holds, we expect to experience censorship approximately 50% of the time (as this is how frequently China's censors enter the resynchronization state [24]). Indeed, when we perform this experiment, that is exactly the result we see. Note that if we perform this sequence number adjustment experiment without running the server-side strategy, we never experience censorship as expected, because the real query is now desynchronized from the connection.

This experiment suggests that Strategy 11 actually performs a desynchronization attack against the GFW, and that a bug exists in the GFW's resynchronization state handling of simultaneous open. As we will see, this bug is quite powerful, and Geneva identifies it repeatedly in our experiments.

Strangely, Strategy 11 does not work well against HTTPS. We hypothesize

this is because the RST does not cause the GFW to enter the resynchronization state for HTTPS, but does for the other protocols. The rest of this section explores a number of cases in which TCP/IP-level attacks work well for one application-level protocol but not another; §4.4 offers an explanation why this occurs.

Strategy 12: Simultaneous Open, Injected Load (China)
DNS (83%), FTP (36%), HTTP (54%), HTTPS (55%), SMTP (59%)

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

Strategy 12 also relies on simultaneous open, but with a slightly different mechanism. Rather than injecting a RST, it changes the outgoing SYN+ACK packet into two SYN packets: the first SYN is well-formed and the second has a random payload. It has comparable success to Strategy 11, though slightly worse for FTP (36% vs. 52%) and SMTP (59% vs. 70%), and better for HTTPS (55% vs. 14%).

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

Strategy 12 does not damage the TCP connection despite the client being

unmodified. Although it is uncommon for SYN packets to carry a payload, this is permitted by the RFC (this behavior is required by TCP Fast Open), and the payload is ignored by the client (though the client does respond with an ACK to acknowledge the current sequence number).

Strategy 13: Corrupted ACK, Simultaneous Open (China)
DNS (26%), FTP (65%), HTTP (4%), HTTPS (4%), SMTP (23%)

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

Geneva identified one final strategy relying on simultaneous open. Strategy 13 copies the SYN+ACK packet: it corrupts the ack number of the first, and converts the second to a SYN. The SYN+ACK with the corrupted ack number induces the client to send a RST packet, before responding with a SYN+ACK to initiate the TCP simultaneous open. However, unlike Strategies 11 and 12, this strategy is the most successful for FTP.

Wang et al. [24], while studying HTTP censorship, hypothesized that a SYN+ACK from the server with an incorrect ack number is sufficient to trigger the GFW's resynchronization state. We observe that this is no longer true for; however, it *does* work for FTP censorship. Therefore, when the SYN+ACK with the corrupted ack number is sent, the FTP portion of the GFW enters the resynchronization state and resynchronizes on the next packet from the client—the RST induced by the incorrect ack number. Because the RST packet has the incorrect sequence number, the GFW will become desynchronized from the connection. Geneva also identified successful

variants of this species in which the order of the two packets is reversed.

Strategy 14: Corrupt ACK Alone (China)
DNS (7%), FTP (33%), HTTP (5%), HTTPS (5%), SMTP (22%)

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

Strategy 14 is identical to Strategy 13, but without simultaneous open. This shows that, although simultaneous open is not required to evade FTP censorship, it improves the success rate (33% vs. 65%).

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

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

Strategy 15 offers an even greater improvement in success rate. This strategy sends a SYN+ACK with a corrupted ack number, followed by another SYN+ACK with a random payload. As with the previous strategies, the corrupted ack number induces the client to send a RST packet, which the GFW resynchronizes on. This RST is critical to the strategy's success: if we instrument the client to drop this induced RST, the strategy stops being effective.

Strategy 15 is highly successful (97%), but again, largely only applicable to FTP. We do not yet understand the reason for the improvement in success rate with the inclusion of simultaneous open or an inert payload.

We draw special attention here to the specific order that the injected packets

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

Strategy 16: Injected Load, Induced RST (China)
DNS (82%), FTP (55%), HTTP (52%), HTTPS (54%), SMTP (55%)

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

Resynchronization State, Revisited Strategy 16 replaces the outbound SYN+ACK with three packets: (1) A FIN with a random payload, (2) A SYN+ACK with a corrupted ack number, and (3) The original SYN+ACK. Note the apparent similarity with Strategy 15: an inert payload and SYN+ACK with corrupted ack are both sent to the client, but Geneva found that adding the FIN makes the strategy more effective for all but FTP. We also found that this strategy works equally well if an ACK flag is sent instead of FIN.

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

packet with an incorrect ack number.

This presents a stark difference from Strategy 15—once the corrupted ack number caused the GFW to enter the resynchronization state over FTP, the GFW did not resynchronize on the next packet in the connection (which would be a `SYN+ACK` with the correct sequence and ack numbers), but rather on the next packet from the client (the `RST` with an incorrect sequence number). This has a surprising implication: depending on the *reason* the GFW enters the resynchronization state, it *behaves differently*.

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

1. A payload from the server on a non-`SYN+ACK` packet causes the GFW to resynchronize on the next `SYN+ACK` packet from the server or the next packet from the client with the `ACK` flag set for every protocol.

2. A `RST` from the server causes the GFW to resynchronize on the next packet it sees from the client for each protocol except HTTPS.

3. A `SYN+ACK` with a corrupted ack number only causes a resync for FTP, and it resynchronizes on the next packet from the client.

We test this theory with Strategy 17, which begins by copying the `SYN+ACK` packet twice. To the first duplicate, the flags are changed to `RST`, to the second duplicate, the ack number is corrupted, and the third is left unchanged. All three packets are then sent. The first `RST` packet is ignored by the client, the corrupted

ACK induces the client to send a RST, and finally the client responds to the server's SYN+ACK with an ACK to properly finish the handshake.

Strategy 17: Injected RST, Induced RST (China)
DNS (83%), FTP (85%), HTTP (54%), HTTPS (4%), SMTP (66%)

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

If our above new model for the resynchronization state holds true, we expect the first RST packet of Strategy 17 to put the GFW in the resynchronization state for every protocol but HTTPS, and resynchronize *not* on the next packet it sees in the connection or the next SYN+ACK, but on the next packet it sees from the client, which is the induced RST with an incorrect sequence number.

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

Strategy 18: TCP Window Reduction (China)
DNS (3%), FTP (47%), HTTP (2%), HTTPS (3%), SMTP (100%)

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

TCP Window Reduction Strategy 18 works by reducing the TCP window size and removing wscale options from the SYN+ACK packet, inducing the client to seg-

ment the forbidden request. This strategy is almost the exact same strategy identified by brdgrd [45] in 2012. The fact that this strategy works at all is highly surprising—the GFW has had the capacity to reassemble segments since brdgrd became defunct in 2012. It appears that the portion of the GFW responsible for FTP censorship is incapable of reassembling TCP segments. This strategy is also the most effective at evading SMTP censorship in China, and as we show next, it is highly effective in other countries, as well.

4.3.2 Server-side Evasion in India & Iran

Our vantage points in India are all within the Airtel ISP, and we confirm that Airtel only censors over HTTP [28]. Our vantage points in Iran are in Zanjan and Tehran; here, HTTP, HTTPS, and DNS is censored (though DNS-over-TCP is uncensored, so we will focus on HTTP and HTTPS here).

Airtel's censorship injects an HTTP 200 with a block page with a `FIN+PSH+ACK` packet instead of tearing down the connection. Iran's censorship simply "blackholes" the traffic, dropping the offending packet and all future packets from the client in the flow for 1 minute. In India, as reported by Yadav et al., we also observe a follow-up `RST` packet from the middlebox for good measure [28].

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

censorship response.

Given the lack of state tracking for these middleboxes, the problem of server-side evasion becomes even more challenging: there is no censor state to invalidate or teardown, so the only feasible strategies are those that mutate the client's forbidden request in a manner that cannot be processed by the censor. When deployed from the server side, Geneva identifies one such strategy in both countries that we have already seen: TCP Window Reduction (Strategy 18).

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

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

4.3.3 Server-side Evasion in Kazakhstan

Kazakhstan has deployed multiple types of censorship. Previous works have explored weaknesses in their now-defunct HTTPS man-in-the-middle [40]. Here, we focus on their in-network DPI censorship of HTTP. Like the Airtel ISP, the censor steps in when a forbidden URL is specified in the `Host:` header of an HTTP GET

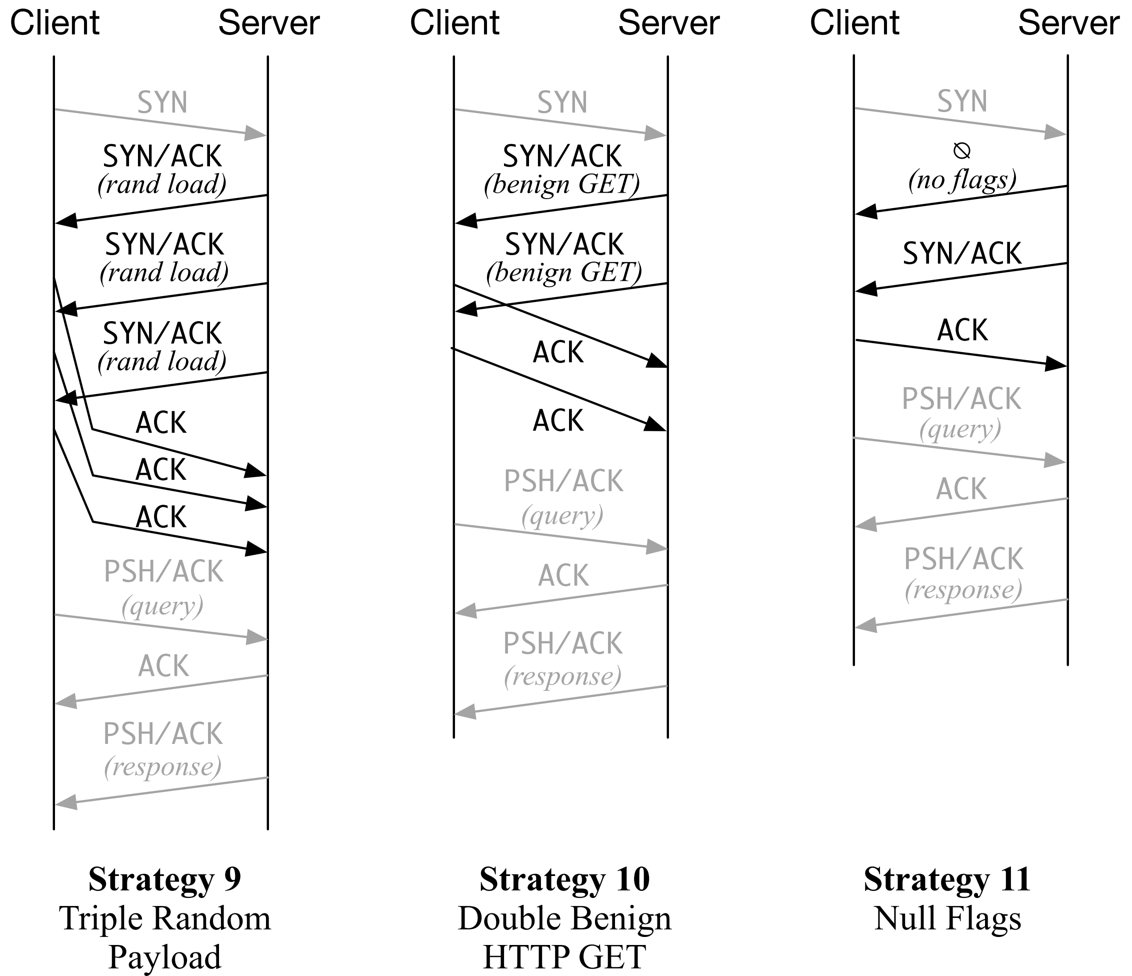


Figure 4.2: Server-side evasion strategies that are successful against HTTP in Kazakhstan.

request. When the censor activates, it first performs a man-in-the-middle, so all packets in the TCP stream (including the forbidden request) for approximately 15 seconds are intercepted by the censor and will not reach the server. The censor then injects a `FIN+PSH+ACK` packet with a block page to inform the user the page is blocked and the connection terminates.

We provide an overview of our successful server-side evasion strategies against Kazakhstan in Figure 4.2.

Strategy 19 takes the outbound `SYN+ACK` packet, adds a random payload, and

Strategy 19: Triple Load (Kazakhstan) *HTTP (100%)*

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

then duplicates it twice, effectively sending three back-to-back SYN+ACK packets with payloads. The payloads and duplicate packets are ignored by the client, and the client completes the 3-way handshake. This strategy works 100% of the time in Kazakhstan.

Strangely, we find that Strategy 19 is effective only if the packet with the load is sent at least three times. Increasing the number of duplicates does not reduce the effectiveness of the strategy, but removing any of them renders the strategy unsuccessful.

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

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

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

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

Strategy 20: Double GET (Kazakhstan) *HTTP (100%)*

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

Strategy 20 duplicates the outbound SYN+ACK packet and sets the load to the first few bytes of a *well-formed, benign* HTTP GET request. Since this payload is on the SYN+ACK, the client ignores it, and the TCP connection is unharmed, but the payload is processed by the censor. The above strategy shows the minimum portion of a HTTP GET request required for the strategy to work (if the "." is removed, the strategy stops working). As long as the GET request is well-formed up to the ".", the strategy works; for example, the strategy works equally well if we specify

the rest of the GET request or use a different or longer path. We also find that the duplicate is required for this strategy to work; if the GET is only sent once, the strategy does not work.

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

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

Strategy 21: Null Flags (Kazakhstan) *HTTP (100%)*

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

Strategy 21 duplicates outbound `SYN+ACK` packet. To the first duplicate, *all* of the TCP flags are cleared before it is sent, and the second duplicate is sent

unchanged. We find this strategy works 100% of the time. Although Geneva first discovered this strategy by clearing the TCP flags, it also identified the strategy works as long as `FIN`, `RST`, `SYN`, and `ACK` are not used. We hypothesize the censor is monitoring for "normal" TCP handshake patterns, and when those patterns are violated, the connection is ignored.

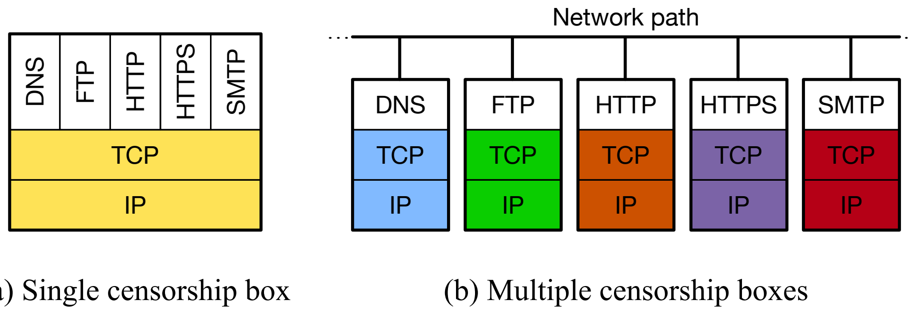
Finally, as expected, Strategy 18 also works in Kazakhstan: inducing client segmentation is sufficient to defeat the censor.

4.4 Multiple Censorship Boxes

The server-side evasion strategies from §4.3 exhibit a surprising property: although they strictly operate at the level of TCP (specifically the 3-way handshake), they have varying success rates depending on the higher-layer application within a given country. This defies expectation: our evasion strategies exploit gaps in censors' logic or implementation at the transport layer, and thus those same gaps *ought* to be exploitable by *all* higher-layer applications. Exceptions to this indicate either a cross-layer violation or a different network stack implementation for each application—two phenomena that are necessarily rare in the layered design of the Internet.

The remaining explanation is that China uses distinct boxes—with distinct network stack implementations—for each of the application protocols they censor. We depict this in Figure 4.3.

This raises an important question: how does the censor know which box to



(a) Single censorship box (b) Multiple censorship boxes

Figure 4.3: Single versus multiple censorship boxes. A standard assumption is that evasion strategies that work for one application will work for another within a given country. However, our results indicate that China's GFW uses distinct censorship boxes for each protocol, each with their own network stacks (and bugs).

apply? This is not as simple as triggering on port numbers; recall that, in our experiments, we randomize the server's port numbers, and yet still experience censorship for each protocol. Indeed, most of the GFW's censorship is *not port-specific*.

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

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

would destroy every connection.

To see if we can detect the presence of multiple boxes, we sought to locate them via TTL-limited censored probes [28]. We instrumented a client to perform 3-way handshakes with servers of various protocols, and then send the query repeatedly with incrementing TTLs until it elicits a response from a censor. We found that, in China, censorship occurred at the same number of hops for each protocol at each vantage point. This indicates that, if there are indeed multiple boxes, then China collocates them.

4.5 Client Compatibility

The evasion strategies presented in §4.3 take advantage of esoteric features of TCP that appear to have faulty implementations in nation-state censors' firewalls. Server-side deployment risks making the server unreachable to any client that also has the same shortcomings. Conversely, strategies that work for a diverse set of clients are readily deployable. Here, we comprehensively evaluate of all of the strategies against a diversity of client operating systems, and we provide some anecdotal evidence across different link types.

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

Standard), **MacOS** (10.15), **iOS** (13.3), **Android** (10), **Ubuntu** (12.04.5, 14.04.3, 16.04.4, 18.04.1), and **CentOS** (6, 7). We tried each protocol and each server-side strategy against each client.

OS Results We found that *all but three strategies* worked on *every* version of every client OS. The only exceptions were Strategies 15, 19, and 20, each of which failed to work on any of the versions of Windows and MacOS. These three strategies all involve sending a `SYN+ACK` with a payload; Linux's TCP stack ignores these, but Windows' and MacOS's do not.

However, we can slightly alter Strategies 15, 19, and 20 to make them work with all clients. The key insight is that these strategies work on Linux precisely because Linux ignores the payload (but censors do not). However, we can modify the strategy in other ways to make the client ignore the packet while the censor still accepts it; this is commonly referred to as an "insertion" packet, and there are other ways to create insertion packets [40]. For instance, we can send the payload packets with a corrupted chksum (so they are processed by the censor but not the client), and send the original `SYN+ACK` packet unmodified afterwards. We re-evaluated these three strategies with this modification, and found that with this small change, the strategies worked for *all* client operating systems. An area of future work is evolving strategies directly against many operating systems to avoid requiring these post-hoc modifications.

Results Can Vary by Network We close this section with an *anecdotal* observation. In addition to the tests on our private network, we also tested all strategies

from a Pixel 3 running Android 10 on wifi and two cellular networks: T-Mobile, and AT&T in a non-censoring country (anonymized for submission). All strategies worked over wifi, and all worked on the two cellular networks *except* Strategies 11 and 13 for T-Mobile and Strategies 11, 12, and 13 (all of the simultaneous open strategies) for AT&T. We speculate that the failures were caused by other in-network middleboxes. This indicates that, while the *client* may not be an issue with some server-side strategies, the client's *network* might.

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

4.6 Deployment Considerations

Where to Deploy? Though we refer to them as "server-side," the strategies we have presented could be deployed at any point in the path between the censor and the server. For instance, a reverse proxy (such as a CDN), a common hosting platform (like Amazon AWS), or even a middlebox along the path (like in TapDance [47]) could run our strategies by manipulating packets in-flight. However, for ease of deployment, we anticipate that our strategies will mainly be run at whichever host is performing the 3-way handshake with the client. Our strategies incur little computation or communication overhead (at most three extra payloads), so we

expect that they could be deployed even in performance-critical settings.

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

4.7 Ethical Considerations

Ethical Experiments We designed our experiments to have minimal impact on other hosts and users. All of our testing and training was done from machines directly under our control. Geneva generates relatively little traffic while training [40] and does not spoof IP addresses or ports. We follow the precedent of evaluating strategies strictly serially, which rate-limits how quickly it creates connections and sends data. We believe this mitigates any potential impact it may have had on other hosts on the same network.

Ethical Considerations of Server-side Evasion In traditional, client-side tools for censorship evasion, the user is directly responsible for attempting to evade the censor, and is taking a deliberate action to do so. As such, the user has the oppor-

tunity to both *decide* and *consent* to the evasion, and (ideally) is knowledgeable of the risk associated with attempting to (and/or failing to) evade censorship.

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

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

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

4.8 Conclusion

In this chapter, I supported my thesis across multiple network protocols and in a novel deployment context: server-side evasion. I have presented eleven server-side packet-manipulation strategies for evading nation-state censors—ten of which are novel and, to my knowledge, the *only* working server-side strategies at time of

writing. My results lend greater insight into how the national censors in China, India, Iran, and Kazakhstan operate: we find, for instance, that the GFW appears to use separate censoring systems for each application it censors, and that each such system has gaps in its logic, bugs in its implementation, and different network stacks—all of which we have shown can be exploited to evade censorship. Such heterogeneity severely complicates the process of evading censorship. Fortunately, we have shown that, by applying automated tools like Geneva, it is possible to *efficiently* evade (across multiple protocols) and understand a threat as nuanced (and buggy) as nation-state censors.

In the next chapter, I will lend additional support to my thesis across multiple protocols. This chapter's results showed that TCP/IP level packet manipulation could render middleboxes ineffective across multiple application layer protocols. Next, I will show that packets can be efficiently manipulated at the application layer itself to render middleboxes ineffective.

Chapter 5: Application-Layer Evasion

The previous two chapters demonstrated that both client- and server-side evasion strategies can be automatically discovered, but were limited for the most part to manipulations of IP and TCP headers. This leads me to ask: Are TCP/IP-level packet manipulations the only way that middleboxes can be rendered ineffective? Can censorship be evaded via manipulating application-layer data, instead? In this chapter, I will explore how middleboxes can be rendered ineffective, even if packet modifications are restricted to the application-layer.

The ability to automatically discover censorship evasion strategies is powerful, but by focusing only on TCP and IP headers, the approach suffers 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 [20]). Ideally, censorship evasion could take place by manipulating only *application-layer* data, which could take place in unprivileged usermode.

Lack of UDP support. Geneva (in addition to other tools published after Geneva's release [70, 72]) only support 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, in this chapter we present what we believe to be the first work that automatically discovers application-layer censorship evasion strategies. We extend Geneva with application-layer fuzzing and new fitness functions. The fuzzing engine we have built is not the primary contribution of this chapter; 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.

As such, we make the following **contributions**:

- We take the first steps toward automating the discovery of application-layer cen-

sorship evasion strategies. These are easier to deploy than their headers-only counterparts.

- We use our extended build of Geneva to perform a wide-scale 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 Tables 5.2 and 5.3.
- 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.

Why study censorship of unencrypted protocols? HTTPS adoption is on the rise for most of the web [101], and browsers have started to request HTTPS by default [102]. Similarly, 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 still dominates the market, and encrypted DNS alternatives are not yet widely adopted anywhere. 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 [103]. Worse yet, many censored websites still do not support HTTPS. We issued HTTPS requests to all the domains in Citizenlab's censorship

test lists [104] 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 [36], and launching HTTPS man-in-the-middle attacks [105–107]. 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.

Roadmap The rest of this chapter is structured as follows: §5.1 presents further background on the specifics of censorship in the countries we study in this chapter. §5.2 describes the design of our extensions to Geneva and the specific application to DNS and HTTP. §5.3 describes our experimental methodology. §5.4 presents our results from training over HTTP and §5.5 presents our results from training over DNS. We discuss these results, and what we can learn about censors in §5.6, and address ethical considerations in §5.7.

5.1 Application-Layer Censorship Background

In this section, we review additional details about the specific nation-state censorship infrastructure studied in this chapter and additional background about application-layer fuzzing techniques.

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 [1, 2, 28]. Censors in India and Kazakhstan examine the

Host header, while the Great Firewall of China 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 Great Firewall of China 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 will commonly inject responses that contain an incorrect IP address. As of time of writing, China has deployed three independent DNS censorship systems running in parallel, each with their own fingerprints and block-lists [33]. Although some DNS and HTTP servers are censored by IP-blocking, the focus of this work will be on the active censorship performed at the application level.

Why extend Geneva? For this work, due to the number of DNS resolvers, HTTP servers, and censoring countries, we will use an automated approach for discovering application layer strategies.

We are familiar with three existing systems to automating censorship evasion: Geneva [1], SymTCP [70], and *Alembic* [72]. 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 teardown a connection (while preserving the connection to the server 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. In our case, we may not have access to the source code of the application servers, and will also run across multiple versions of multiple server types. For this work, we chose

to extend Geneva, and we will detail our design in §5.2.

Application Fuzzing In addition to the relevant fuzzing works described in Chapter 2, most similar to this chapter is a concurrent work T-Reqs [108], a grammar-based differential HTTP fuzzer to identify 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 simply to allow the original request to bypass the censor. 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.

5.2 Fuzzer Design

In this section, for completeness, we discuss 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 [1]) or a symbolic execution approach (SymTCP [70] and *Alembic* [72]) to identify successful modifications to network packets. In this work, we will not always have access to the source code for every application layer server

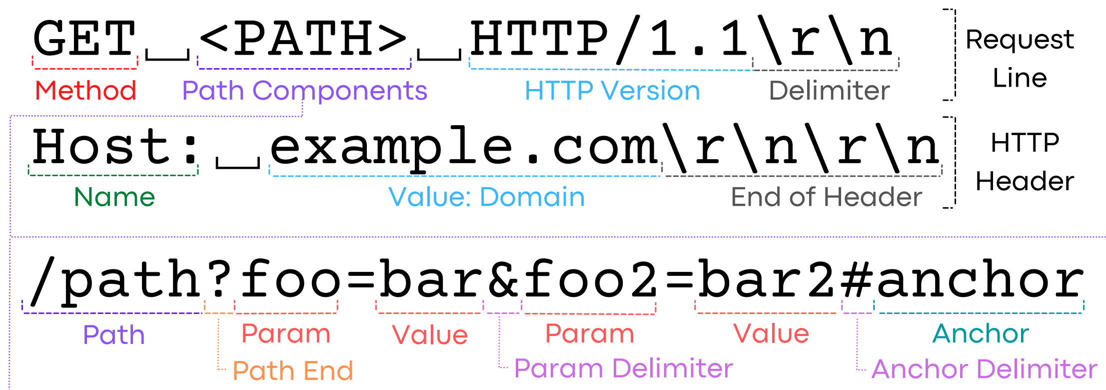


Figure 5.1: **Structure of an HTTP request** for `example.com`. Note that "␣" denotes where whitespace is required by the RFC, typically 1 space. Typically, HTTP Requests contain multiple headers separated by a `\r\n`.

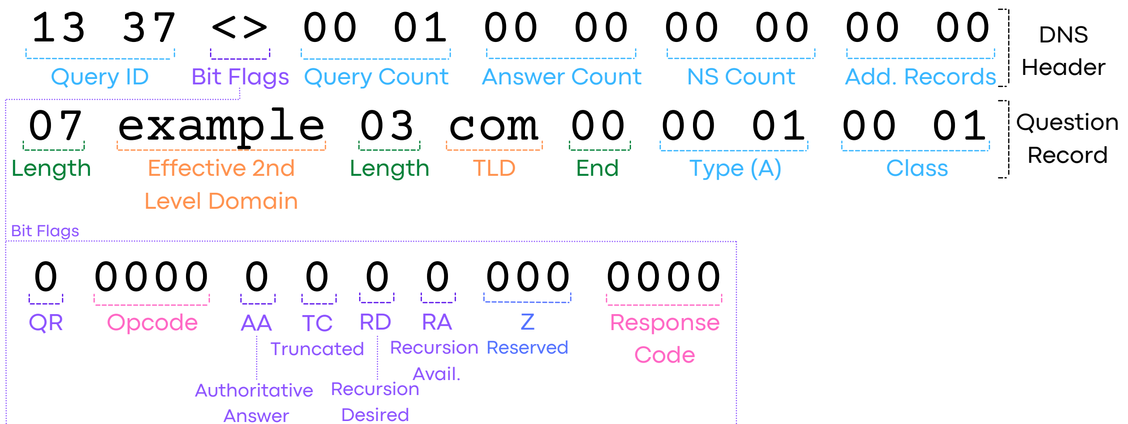


Figure 5.2: **Structure of a DNS request** for `example.com`. Note that the Bit Flags field (detailed in the lower box) is two bytes wide. Although DNS requests typically only contain one Question Record, the RFC [8] allows for multiple DNS Questions to be included with no separator between them.

we need to train with (such as Google's public DNS resolver). Therefore, we will use a fuzzing approach for our design, and specifically will extend Geneva's to the application layer space and re-use its existing genetic algorithm.

What lessons can we learn from the design of Geneva to inform how we should fuzz for application-layer strategies? Geneva built censorship evasion strategies out of small, individual manipulation primitives (called *actions*) that could modify a packet. Each action takes parameter values, which were chosen at random or from packet captures of previous strategies. Since some actions can introduce new packets into the network (such as `duplicate`), these actions compose to form trees that describe how a packet should be modified, and each tree has an associated *trigger* to describe *which* packet to modify. Despite the simplicity of the manipulation actions, by composing them together with associated triggers, Geneva's strategies can be expressive enough such that a strategy can transform any set of packets into any other set of packets.

Each strategy is evaluated with a fitness function, which applies the strategy to modify a request for a forbidden resource and assigns it a numeric fitness value based on its success, overhead, and complexity. The genetic algorithm uses the fitness values to decide which strategies should survive to the proceeding generations and propagate.

How can we apply these ideas 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 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 chapter, other popular application layer protocols follow this pattern; for example, TLS packets usually have many TLS Messages and TLS Extensions.

5.2.1 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 [109–116]. 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 [116]. 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 5.1 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 [8]. 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 5.5, 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 5.2 for the fields in a DNS Query.

5.2.2 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 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.

5.2.3 Fitness Function

In this work, we do not modify Geneva's original genetic algorithm, but we will update its *fitness function* to allow us to evaluate application-layer strategies. We will 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 will initialize it with a clean slate with no access to prior results or knowledge of the censorship system. Our system will execute each training run for a pre-specified number of generations or until population convergence occurs. Be-

tween each training run, we perform post-hoc analysis of the results and strategies the system identified.

HTTP Evaluation To evaluate HTTP strategies, the fitness function makes a request that either contains a forbidden Host header, or a forbidden keyword in the request string. To train for HTTP strategies, we will 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 will allow 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, the fitness function will use a different destination port within a large range of ports for every strategy, and all of these ports will be forwarded to a single port the server runs on. In this regard, we can train without residual censorship affecting the fitness function.

DNS Evaluation To evaluate each DNS strategy, the fitness function 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 [33]. The GFW does not drop the offending query packet, so in addition to the DNS injectors, the in-

tended 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, the fitness function 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 fitness function rewards the strategy's 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 the fitness function punishes the fitness value).

Importantly, as with the HTTP fitness function (and fitness functions from prior work), the fitness function gives 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.

5.2.4 Using Strategies

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.

5.3 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 [117] and BuiltWith [118]. According to both resources, Apache [119] was the most popular (with 36.5% and 35% estimated market share from each respective resource) and Nginx [120] 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. Deploy-

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

ments of Apache and Nginx span many versions; we selected the four most popular versions for each, according to W3Techs [117], 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 [121]. 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 5.1 for a full list of the resolvers 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 uncensored 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 will trigger 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` to redirect all of these ports to a single port that hosts the server. The fitness function ensures that each strategy gets its own port. 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 §5.4.3.

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 [33]. 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 will 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 §5.2.4, 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.

5.4 HTTP Results

In this section, we will detail our results from training our system against HTTP censorship 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.

5.4.1 Summary Results

We only report on strategies for which at least one HTTP server we tested correctly responded. We consider a strategy unique if it defeats censorship, or is accepted by a server, for a unique reason. This means for each strategy, there are often many ways to craft strategy variants that do functionally the same thing, but the total number of strategies we report are only those that work for a unique reason.

In total, we identify 77 unique HTTP strategies, and we manually performed experiments to understand how they work and determine their success rate against each country and HTTP server. We found the most strategies that defeated Airtel's censorship in India: of the 77 strategies we discovered, an incredible 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 if a censor is more RFC-compliant than the destination server, there will be many more opportunities 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.

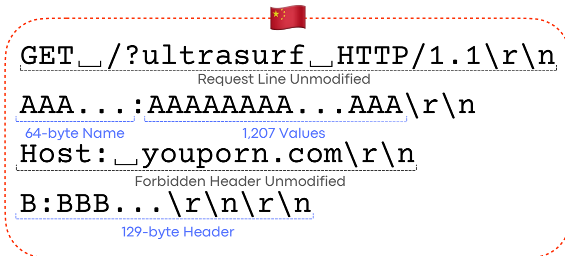
5.4.2 Evasion Strategies

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

(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 [110] permits more.

(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.



(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 5.3: **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).

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 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 [110, 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 5.3a. 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) [110]. 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 5.3b. 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) [122], 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 [2].

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 additional 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 5.3c.

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.

5.4.3 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 [123], 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 [111]. 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.

5.5 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 5.1). 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. 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) [33].

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

Family	Strategy	Apache 2.4.X 6	Apache 2.4.X 18	Apache 2.4.X 29	Apache 2.4.X 43	Nginx 1.X.X 13.4	Nginx 1.X.X 14.1	Nginx 1.X.X 16.1	Nginx 1.X.X 19.0	Country CN-H	Country CN-K	Country IN	Country 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 5.2: 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 i Table 5.3.

		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 5.3: Continuation of Table 5.2. 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.

Strategy Family	Strategy	CF	G	Q9	OD	CB	CS	DW	V
Elevated Count; ZBit truncated	[DNS:*:*]-tamper{DNS:nscount:replace:1}(tamper{DNS:z:replace:1}(tamper{DNS:tc:replace:1},),)-—	✓	-	-	✓	✓	-	-	-
Elevated Count	[DNSQR:qname:*]-tamper{DNS:qdcount:replace:2}-—	✓	-	-	-	-	-	-	-
Long Secondary query; Elevated Count	[DNSQR:qclass:]-tamper{DNS:ancount:replace:98}-— [DNSQR:qtype:]-replace{%C3%95:name:262}-—	✓	-	-	✓	-	-	-	-
Long Secondary Query	[DNSQR:qname:*]-duplicate(,replace{%C2%91:name:957})-—	-	-	✓	-	-	✓	✓	✓
Compression	[DNS:*:*]-tamper{DNS:qd:compress}(tamper{DNS:qdcount:replace:2},)-—	✓	✓	-	-	-	-	-	-

Table 5.4: 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), Google (G), Quad9 (Q9), OpenDNS (OD), CleanBrowsing (CB), ComodoSecure (CS), Verisign (V), and DNS.Watch (DW). 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.

of the success rates below, we test each strategy 1000 times. See Table 5.4 for the full breakdown of results.

Elevated Count Fields The simplest family of strategy types we discovered works by simply increasing one of the count fields (`qdcount`, `ancount`, `arcount`, or `nscount`) by 1. All four of these strategies are in violation of the RFC: the request only contains 1 Question Record and 0 Answer, Name Server, or Additional records. Surprisingly, each of the GFW's injectors and open resolvers respond differently depending on which field we modify.

Elevating the `qdcount` field to 2 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 injector 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.

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 [8]) works by splitting the DNS query across

multiple records at the separator. This strategy is related to the **Elevated Count Fields** 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`.

Truncated-Reserved 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.

Multibyte Long 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 cause trouble for each injector. A strategy will evade injector #1 if it inserts a new DNS Question Record containing at least 241 2-byte-wide multibyte characters. A strategy will also evade injector #3 with at least 482 multibyte characters; any less, and the strategy fails to evade #3. We note that the required number to evade injector #3 is exactly double that required to evade injector #1. Injector #2 can also be evaded with a 36% success rate with 721 2-byte-wide multibyte characters; any less than 721 and the strategy fails to evade #2. 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 all of these requests are not 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 and ARCount 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. The benefit of this re-combination of the above strategies is that it permits different resolvers to respond: by injecting fewer characters, Cloudflare and OpenDNS now respond to the query, but Quad9, Comodo, and DNS.Watch will not respond to the elevated `arcount`.

5.6 Discussion

In this section, we discuss our results, and what we can learn about the nation-state censors.

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. The reason for this is that 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 chapter (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.

5.7 Ethical Considerations

We design 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 [1, 52].

We perform all of our system training exclusively from vantage points we control, and our work does not require recruiting users (unwitting or not [90]). 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 that involved 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.

5.8 Conclusion

In this chapter, we present the first techniques to automate the discovery of new censorship evasion techniques purely in the application layer. The approach is applicable to HTTP and DNS, and we trained our system against three distinct

HTTP and DNS censors across China, India, and Kazakhstan. In total, we discover 9 unique strategies for DNS and 77 unique evasion strategies for HTTP, which exploit differences between how the censor and destination server process a request. All of these evasion strategies require only application-layer modifications, making them easier to incorporate into applications and deploy.

Taken collectively with the Chapter 3 and Chapter 4, I have demonstrated that it is possible to render middleboxes ineffective at implementing their policy (incapable of correctly censoring traffic when they should) from the client-side and server-side and via both TCP/IP and application-layer packet manipulations.

Chapter 6: Censorship-in-Depth: Iran

Through the years of implementing, evaluating, and applying Geneva, I have observed that censoring nation-states have deployed new, more sophisticated censorship infrastructures, with multiple middleboxes running in parallel. This provided a unique opportunity to evaluate whether my thesis applies even as censors evolve: that is, whether Geneva is able to quickly and effectively render new forms of censorship ineffective. In this chapter and the next, I evaluate this in the context of new forms of censorship in Iran and China, respectively.

Censoring nation-states employ defense-in-depth, layering multiple orthogonal censorship mechanisms to make it more difficult to communicate with certain destinations or via certain protocols. Typically, such "censorship-in-depth" involves wholly different systems, such as combining lemon DNS responses [38,41], IP blocking [1, 124], and TLS SNI blocking [2]. As a result, each form of censorship targets different packets, and can often be studied and defeated in isolation.

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

simultaneously.

In early 2020, Iran launched such a form of censorship-in-depth by deploying their protocol filter. A *protocol filter* only allows a small list of protocols to be used, and censors protocols it forbids. A similar system in Iran was first reported on by Aryan et al. [55] in 2013, but to the best of our knowledge was not used for years until it was turned back on in 2020. We are also unfamiliar with any work detailing how Iran's protocol filter works or how to evade it—underscoring the difficulties inherent in measuring and circumventing censorship-in-depth.

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

The rest of the chapter is organized as follows. §6.1 reviews prior work in measuring Iranian censorship. §6.2 describes our methodology and vantage points we use for our experiments. §6.3 presents our analysis of the protocol filter. §6.4 discusses how the protocol filter can be evaded. Finally, §6.5 concludes.

6.1 Iranian Censorship Background

Iranian censorship has been studied in broader efforts to measure global censorship [52, 125–129]. There have been fewer studies specific to how Iran's censorship operates. Notably, Anderson proposed a technique for detecting censorship via throttling in Iran [130].

The most closely related study to this chapter was a 2013 study by Aryan et al. [55]. They observed throttling between two vantage points that affected SSH, custom obfuscated SSH, and custom obfuscated HTTP. Since HTTP and HTTPS were unaffected, the authors hypothesized that Iran had deployed a protocol filter and were throttling connections that did not match HTTP and HTTPS. This behavior disappeared shortly after Iran's June 2013 election, and to the best of our knowledge, there have been no further reports on protocol filtering.

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

6.2 Methodology

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

In our experiments to measure the protocol filter (§6.3), we performed active measurements from these vantage points to servers we controlled outside of Iran, in Amazon EC2, Microsoft Azure, and DigitalOcean (located geographically in Japan, Ireland, the United States, Australia, and India). We find no significant difference in the behavior of the protocol filter across any of our vantage points or external servers, nor did we observe any change in the behavior of the filter during the course of our experiments to the time of writing.

To develop new evasion strategies (§6.4), we used Geneva and trained it from the client-side and the server-side to discover ways to defeat the protocol-filter in isolation.

6.3 Protocol Filter

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

6.3.1 How Iran's Protocol Filter Works

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

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

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

We find that Iran's protocol filter affects only TCP traffic, and only on ports 53 (commonly DNS), 80 (commonly HTTP), and 443 (commonly HTTPS). Traffic

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

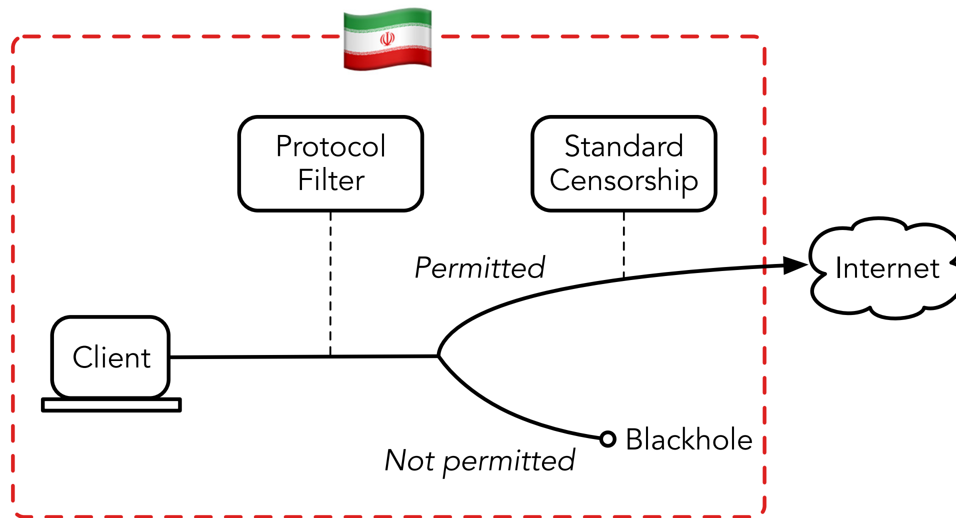


Figure 6.1: Iran's layered censorship system, employing defense in depth. Note that the order of censorship systems is unknown; this is simply a graphical depiction.

sent on any other port is not filtered (and is therefore also not subject to Iran's standard censorship, which only operates over these same ports).

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

How many packets does the filter monitor? To answer this question, we sent multiple packets with non-protocol data (e.g., "`test`") before well-formatted allowed protocol data. We determined that the filter monitors the first two *data-carrying* packets from the client at the start of a connection. If either of those two packets matches a protocol fingerprint, the flow is unharmed; if no packet does, the second packet and rest of the flow are dropped.

How long does the filter censor an offending flow? To test this, we in-

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

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

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

However, we find that the filter is *not* bidirectional: it only affects connections where the client is inside Iran. The server also receives almost no indication censorship has taken place. Recall that packets from the server are unaffected: unlike with the Great Firewall of China, which sends `RST`s in both directions [24], Iran's protocol filter only affects the packets sent by the client. This makes it difficult to identify and study the protocol filter without vantage points within Iran.

Can the protocol filter reassemble TCP segments? We repeatedly made

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

Table 6.1: Top 10 providers for *affected* IP addresses.

valid but segmented DNS, HTTP, and HTTPS requests on filtered ports[2]. We find that segmenting our requests too many times incurs censorship from the protocol filter, indicating that, like Iran's regular censorship infrastructure [1,2,55], the filter is incapable of reassembling TCP segments. We also note that the filter also does not check the checksums of the packets it processes.

6.3.2 Whom the Filter Is Applied To

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

To identify which IP addresses are affected by the filter, we tested the effects of the filter on the Alexa top-20,000 most popular websites. To avoid the effects of

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

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

Table 6.2: Top 10 providers for *unaffected* IP addresses

DNS censorship or requesting IP addresses inside of Iran (as the requests would not cross the filter), we used `dig` outside of Iran to get IP addresses for all 20,000.

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

Over all ten experiments, 3,595 IP addresses (17.9%) tripped the filter at least eight times. Of those, 3,499 were affected all ten times (17.4%), and 278 (1.4%) IP addresses were affected 3–7 times. Tables 6.1 and 6.2 show the number of IP addresses per provider that were affected and unaffected by the protocol filter, respectively. Overall, we find that IP address provider is not correlated with whether the filter affects an IP address or not, but some prefixes are affected significantly

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

more heavily than others.

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

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

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

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

6.3.3 Protocol Fingerprints

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

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

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

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

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

The last requirement appears to be a bug in the implementation of the DNS fingerprint. Recall that the DNS-over-UDP header is slightly different than DNS-over-TCP's; over TCP, the DNS header includes a `length` field [131]. Since the filter is only active over TCP but does not take the extra field into account, it will *never match* a legitimate DNS-over-TCP packet. We believe the reason this oversight has not caused a significant issue is because DNS-over-TCP generally only requires a single data-carrying packet from the client, but Iran's protocol filter only begins dropping packets on the second data-carrying packet.

However, the faulty DNS fingerprint does still pose a problem: clients can reuse DNS-over-TCP connections [132]. In such cases, the filter would allow the first query, but block any subsequent queries made within 60 seconds.

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

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

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

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

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

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

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

Also, the last requirement again appears to be an error in the design of the protocol filter. It allows TLS versions 1.0, 1.1, and 1.2 to be declared, but this version field is not used accurately in practice: TLS servers must accept any two byte value in this field so long as the first byte is `\x03` [133, Appendix E].

The HTTPS fingerprint *does not* filter specific HTTPS connections or applications; it simply enforces that generic TLS is used. As a result, censorship evasion tools that use TLS will likely be unaffected by the protocol filter at this time, as they will fulfill the above fingerprint requirements by default. This also means the protocol filter would spare more secure DNS transport protocols, such as DNS-over-HTTPS and DNS-over-TLS, if those protocols were used over one of the affected ports.

After the first 5 bytes of the packet (the type, version, and the length), the

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

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

Using Fingerprints We find that any of the fingerprints can be used to evade the filter. This presents an opportunity for censorship evasion tool developers: by sending any fingerprint at the start of a connection (or injecting it as an "insertion packet" [1, 24, 134]), we can ensure the filter will permit the rest of the flow, regardless of the actual protocol used. As we will see in the next section, Geneva also independently discovers strategies to inject innocuous fingerprints from the client-side.

6.4 Evading the Protocol Filter

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

6.4.1 Old Strategies Do Not Apply

We first explored whether we could apply the same strategies that work against Iran's regular censorship system (affecting HTTP and HTTPS) to evade the protocol

filter.[5] The only functioning strategy in Iran we are aware of is simple segmentation: simply splitting the censored request into multiple packets to take advantage of the censor's inability to reassemble TCP segments. We find that no other strategies identified by Geneva or prior work defeats Iran's censorship system.

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

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

6.4.2 Evolving New Strategies

To identify new strategies to defeat the protocol filter, we leveraged Geneva, an open-source genetic algorithm designed to evolve packet-manipulation strategies to evade censorship [1]. Unlike most anti-censorship systems, Geneva does not require deployment at both ends of the connection: it runs exclusively at one side (client

[5] Contrary to the 2013 findings by Aryan et al. [55], from our vantage points, we find that Iran's *standard* censorship infrastructure no longer targets DNS-over-TCP at all.

or server) and defeats censorship by manipulating the packet stream to confuse the censor without impacting the underlying connection. Geneva's packet manipulation strategies are expressed in a domain-specific language [1]; we describe each in plain English, but to allow us to unambiguously express strategies, we also present them using Geneva's language.

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

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

6.4.3 Discovered Evasion Strategies

Strategy 22: Innocuous Fingerprint The simplest strategy Geneva identified was to inject a `PSH/ACK` packet with a corrupt checksum and an innocuous HTTP request as the payload immediately following the 3-way handshake. This trivially serves to bypass the filter, as it matches the protocol fingerprints. However, because the checksum is corrupt, the server will not accept this packet. There are other variants of this strategy that ensure that the filter processes the packet but the server does not, such as setting the TTL large enough to reach the censor but too small to reach the server [1].

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

Strategy 22: Innocuous Fingerprint

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

Strategy 23: Double `FIN`s This strategy works by sending two additional packets *before* the 3-way handshake starts: two empty packets with the `FIN` flag set. To the server, the `FIN` packets are ignored, as they are not a part of an active connection, but the filter processes them and causes it to ignore the rest of the connection. We do not understand why this strategy works, though we hypothesize the `FIN` packets trick the filter into thinking it has already missed the relevant data packets, causing

it to ignore the rest of the flow.

Strategy 23: Double FIN

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

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

Strategy 24: Nine ACKs The final client-side strategy we present is stranger than the first two: this strategy works by sending *nine copies* of the ACK packet during the 3-way handshake. This causes the filter to ignore the rest of the flow. This strategy works 100% of the time, and does not affect the underlying TCP connection. We hypothesize this works because the filter has some internal limit on the number of packets it will process for a given flow.

Strategy 24: Nine ACKs

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

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

This strategy presents us with an opportunity to evade the protocol filter *from the server side*. Server-side censorship evasion allows completely unmodified clients to connect directly to a server while the server subverts censorship on behalf of the clients [2].

Since Strategy 24 is effective with any set of TCP flags, if a server can induce the client to send nine non-data-carrying packets before it sends its forbidden request, we can defeat the protocol filter. We can accomplish this using a trick from prior deployments of `Geneva`: by sending multiple `SYN+ACK` packets during the three-way handshake with a corrupted `ack` number, we induce the client to respond with multiple `RST` packets.

Strategy 25: Nine Induced `RST`s, Server Side

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

Strategy 25: Nine Induced `RST`s This strategy sends nine corrupted `SYN+ACK`s, followed by one unaltered `SYN+ACK`. This induces the client to send nine `RST` packets with corrupted sequence numbers before sending its normal `ACK`, thereby evading the protocol filter.

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

6.5 Conclusion

In 2020, Iran took the latest step in censorship-in-depth by deploying a protocol filter alongside their standard censorship infrastructure. In this chapter, I have performed a deep investigation into Iran's protocol filter. Using vantage points within Iran and servers outside, we empirically demonstrated how the protocol filter works, what its fingerprints are, and to a lesser extent whom it filters. Also, using Geneva [1, 2], I identified four ways to bypass the protocol filter—three from client-side and one from server-side. My results collectively show that Iran's two censorship systems can still be studied in isolation, and bypassed together.

Iran has had a greater capacity for censorship than they have exercised in the past, and the protocol filter can pose a threat to existing deployments of censorship-evasion tools (VPNs, Tor, etc.). As the censorship arms race advances, we anticipate censorship-in-depth to become increasingly common. In the next chapter, I will show a second example of a censorship-in-depth censorship deployment, this time in China, and will show that my thesis still holds.

Chapter 7: Censorship-in-Depth: China's SNI Censorship

As shown in the previous chapter, censorship-in-depth deployments can complicate censorship measurements and censorship evasion. In this chapter, I will showcase a second example of censorship-in-depth, this time in China, where I discovered that the GFW was using two independent middleboxes running in parallel to censor HTTPS connections with SNI.

As much of the web transitions to HTTPS, nation-state network censors have less information to base their decisions of whether to block or tear down a connection. Whereas HTTP permitted deep packet inspection (DPI) of keywords, HTTPS hides all request and response data through encryption. However, the server name indication (SNI) field in the TLS handshake reveals the website to which the client wishes to connect. Censors such as China and Iran have thus used the plaintext SNI field to guide their censorship decisions and, in some cases, outright block all traffic that seeks to hide the SNI through encryption (ESNI) [36].

As a result, significant effort has been paid to understanding and evading SNI censorship, with particular attention paid to one of the world's largest censors,

the so-called Great Firewall of China (GFW). In 2019, Chai et al. [7] empirically evaluated how SNI censorship operated in China, and argued for the importance of using ESNI. Unfortunately, China began blocking all ESNI traffic the next year [36]. In 2020, Bock et al. investigated how to evade China's SNI censorship [2] and recently demonstrated how to weaponize it to launch availability attacks [135]. Through all of this work, a mental model emerged that indicated that China uses a single model of middlebox to detect and react to SNI connections.

In this chapter, I show that in fact China's GFW uses *two* distinct censorship mechanisms in parallel to censor HTTPS based on SNI.[1] We first discovered this second HTTPS censorship middlebox while trying to reproduce the censorship evasion results from Chapter 3 for HTTPS. We observed that some censorship evasion strategies could evade the GFW's known HTTPS censorship, but small modifications could cause strategies to fail unexpectedly: via a single `RST` packet deeper in the TLS handshake. Now, we understand and report on the root cause of this strange behavior: the GFW had a second censorship middlebox all along.

In this chapter, I present a detailed analysis of China's secondary HTTPS censorship middlebox: how it works, how it can be triggered, and how it can be defeated. We confirm this behavior is caused by a separate middlebox by identifying unique TCP-layer bugs in each middlebox, suggesting separate TCP stacks [2]. These findings are important in refining our understanding of SNI censorship in China—they resolve some of the confusing behavior previously identified and chart

[1] We only focus on SNI-based censorship of HTTPS, and thus use "HTTPS censorship" and "SNI censorship" interchangeably.

a clearer path forward for how to measure and evade SNI censorship more precisely. This is especially important now, as China has effectively stopped the roll-out of ESNI within its borders [36] and Russia is actively working to do the same [136]. These findings also support my thesis, and show that middleboxes can be rendered ineffective, even in more complex deployment scenarios.

Whereas prior approaches and the previous chapter investigate cooperating mechanisms that aim to censor *different* but complementary protocols, we have identified two distinct mechanisms that both aim to censor the *same exact* protocol (SNI-based HTTPS). As we will demonstrate, this makes it particularly challenging to disentangle the two, as they both operate on the same packets. Our findings demonstrate what we believe to be a novel way in which nation-states employ censorship-in-depth. Although it is tempting to think of them as a single "black box" of censorship, this chapter shows that it is both possible and important to tease them apart into their constituent components, even in this deployment context.

The rest of this chapter is organized as follows. §7.1 discusses the methodology for our experiments. §7.2 shows how we can evade the newly discovered censorship middlebox and how censorship evasion is critical for our measurements of the new middlebox. §7.3 studies the functionality of the new middlebox. Finally, §7.4 discusses ethical considerations and §7.5 concludes.

7.1 Methodology

Measuring two censorship mechanisms that both operate on the same packets is challenging. To understand how they both operate independently and in conjunction with one another, our methodology involves evading one of the boxes to selectively measure the other. In this section, we describe our high-level approach to evasion and measurement.

Admittedly, our approach was somewhat circular: our initial measurements provided insight that allowed us to begin evading, which let us perform more measurements, and so on. Thus, to best understand our methodology, it is useful to also understand at a high level how the two censorship mechanisms work, which we also provide here.

Vantage Points We obtained two censored vantage points inside China (Beijing) and external uncensored vantage points in Japan (Tokyo) and the United States (Iowa, Virginia). Our Chinese vantage points are located within different ISPs, but Xu et al. found that the GFW's actual deployment of certain censoring middleboxes may vary based on the type of ISP [137], so our conclusions are limited by the ISPs we can measure. We use the vantage points in China as our "client," and our vantage points outside as our "server." Throughout our experiments, we only connect to machines we control.

Detecting Evasion of One Mechanism It is straightforward to determine if we have evaded both of the censorship mechanisms—we need only see if we received

the censored content. But how can we determine if we have evaded censorship of only *one* box?

The key insight is that the two mechanisms block censorship in different ways. The GFW's primary (already known) censorship middlebox operates by injecting an idiosyncratic pattern of three `RST+ACK` packets to both the client and the server once it observes a TLS Client Hello with a forbidden Server Name Indication (SNI) field [2, 7, 135]. We will refer to this primary middlebox as `MB-RA` (MiddleBox `RST+ACK`). The GFW's secondary SNI censorship middlebox, by contrast, tears down connections by injecting one single `RST` packet: we will refer to this middlebox as `MB-R` (MiddleBox `RST`).

Unless otherwise specified, we configured our vantage points to drop all outbound `RST` and `RST+ACK` packets. Thus, we expect any `RST` or `RST+ACK` packets received by our client to come from the `MB-R` or `MB-RA` middleboxes, respectively.

Triggering Censorship We trigger censorship by injecting forbidden domain names in the SNI field (though all communication is strictly between the client and server machines we control). However, we have found that it is not always sufficient to stop sending packets at that time.

Unlike `MB-RA`, `MB-R` does not tear down a connection immediately after observing a forbidden SNI. Instead, it waits to inject its `RST` packet until the client sends the next packet in the TLS handshake: the `ClientKeyExchange` or the `ClientChangeCipherSpec`. Note that the forbidden SNI field is not present in either of these messages. `MB-R` is a stateful middlebox that is *triggered* by the for-

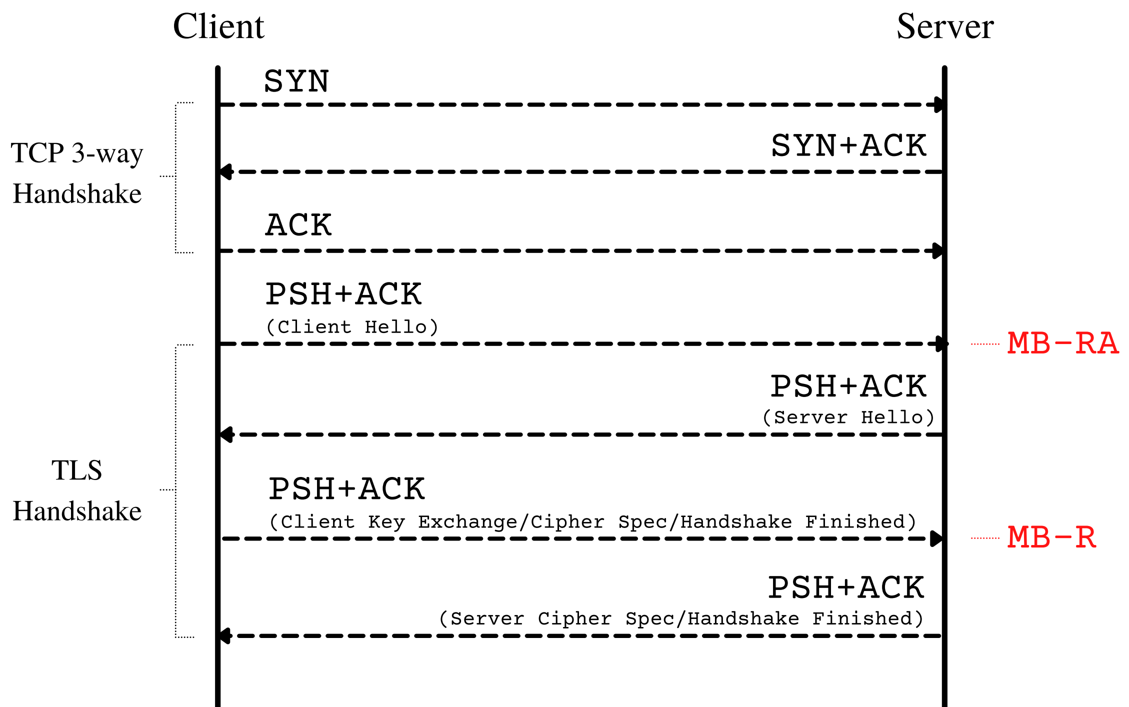


Figure 7.1: A waterfall diagram of the TCP 3-way handshake and the TLS handshake, denoting where the already known `MB-RA` and newly discovered `MB-R` middleboxes act during the connection. Note that `MB-R` does not act until deeper in the handshake than `MB-RA` (and *only* if `MB-RA` does not act), seemingly acting as a backup middlebox for China's HTTPS (SNI) censorship.

bidden SNI field in the Client Hello message but does not *act* until after the client continues the handshake. We believe this is the reason researchers have not reported on this middlebox until now. Figure 7.1 illustrates the TCP 3-way handshake and TLS handshake and where each of the two middleboxes acts.

Isolating the Second Middlebox Studying MB-R is also made more difficult because MB-RA and MB-R seem to interact with one another. Specifically, when MB-RA takes action to tear down a connection, MB-R *does not act* even if MB-RA fails to tear down the connection or the connection continues. We performed an experiment in which we instrumented a vantage point within China and a server outside of China to drop all inbound RST+ACK packets and tried to complete a TLS handshake with a forbidden SNI between them. If MB-R and MB-RA operated independently, after both sides of the connection drop the RST+ACKs injected by MB-RA, we would expect MB-R to inject its RST packet once the client continues the TLS handshake.

Instead, we find that any time MB-RA injects packets, MB-R stops paying attention to the connection entirely. We believe the injected RST+ACK packets from MB-RA are causing MB-R to tear down its TCB (Transmission Control Block) for the connection. This experiment suggests that MB-R is a backup censorship middlebox for MB-RA: it only injects RST packets if MB-RA fails to take action.

This interaction between MB-RA and MB-R also offers MB-R a way to avoid state exhaustion: once a connection is torn down by MB-RA, MB-R does not need to continue tracking it. We believe this interaction also explains why other components of the GFW will stop paying attention to a connection if the client injects a RST+ACK packet

(a TCB Teardown attack). Researchers have wondered why the GFW continues to be vulnerable to TCB Teardown attacks to this day, despite having been reported for years [2,16,23,24,40,70]. If the GFW is architected to internally use the `RST+ACK` packets injected by one middlebox to prevent state exhaustion in other middleboxes, this would explain why TCB Teardown attacks have not been patched.

Unfortunately, this interaction between `MB-R` and `MB-RA` makes studying it in isolation difficult. The only signal we have to measure `MB-R` is its injection of `RST` packets, but it does not inject these packets until deeper in the TLS handshake than `MB-RA`. We could repeatedly make forbidden connections until `MB-RA` fails to inject packets, but to make reliable measurements, instead we leverage packet manipulation evasion strategies to *evade* `MB-RA` without affecting `MB-R`.

Evading Censorship We leveraged an open-source tool called Geneva (*Gen*etic *Eva*sion), a genetic algorithm designed to discover packet manipulation-based censorship evasion strategies. Geneva has been used successfully against the GFW in the past [2, 36, 40], as well as censorship infrastructure in other countries [2, 3].

The output of Geneva is sequences of packet manipulations that confuse or disable a censoring middlebox. Central to Geneva's ability to find evasion strategies is its *fitness function*, which evaluates how successful a strategy is against a given censor. For this work, we made a small modification to Geneva's reward function to optionally ignore inbound `RST` packets on both sides of the connection. This enables us to optionally train Geneva to find strategies that defeat only the `RST+ACK` middlebox (since `MB-RA` injects `RST+ACK` packets, not `RST` packets).

After using Geneva, we performed manual follow-up experiments to understand how each strategy works. To compute reliability for each strategy, we used each strategy 100 times while trying to complete a full TLS handshake with a censored keyword in the SNI field (`wikipedia.org`) between vantage points within China and outside of China.

Because of the interaction between the two middleboxes, we are only able to defeat either `MB-RA` alone or both of them together. Recall that the only signal we have to measure `MB-R`'s reaction is it injecting `RST` packets, but it does not do this injection until later in the TLS handshake after `MB-RA` may act. It is possible that there exist packet sequences that confuse or disable `MB-R` without disabling `MB-RA`, but we are unable to confirm this.

7.2 Evasion

In this section, we will report on client-side strategies we discovered with Geneva that defeat *only* `MB-RA` and both `MB-RA` and `MB-R`. Following precedent from prior work, we will report on the strategies we find both in text and include the Geneva syntax that implements the strategy.

7.2.1 `MB-RA` Evasion Strategies

The most reliable working client-side strategy that we found first sends two `SYN` packets, then splits the TLS Client Hello in half to make two TCP segments, and

sends them out of order[2]. In our testing, this strategy worked with 99% reliability.

Strategy 26: `MB-RA`: Double-`SYN` Segmentation

```
[TCP:flags:S]-duplicate-|
[TCP:flags:PA]-fragment{tcp:-1:False}-|
```

The fact that this strategy works is strange and surprising. The GFW is known to be capable of reassembling TCP segments, even if sent out of order [2]. Indeed, if the second `SYN` packet is removed, the strategy no longer works, as `MB-RA` reassembles the TLS Client Hello and censors the connection. This strategy suggests that `MB-RA` is keeping track of both the TCP handshake and the TLS handshake, but seeing the unexpected `SYN` packet interferes with its ability to reassemble messages. We do not know why this is. Note that this strategy does not evade `MB-R`; this *only* disables `MB-RA`.

The second type of client-side strategy we discovered that defeats `MB-RA` also involves abusing `MB-RA`'s ability to reassemble TCP segments. This strategy involves performing 6 TCP segmentations to create 7 total TCP segments out of the original TLS Client Hello, with each segmentation reversing the order of the segments. In the end, this strategy reverses the order of the segments exactly. This strategy worked with 100% reliability.

This is not the only variant of this strategy that works to defeat `MB-RA`, but it is not sufficient to simply split the TLS Client Hello into any seven segments. Geneva found dozens of strategies with similar number and ordering of segmentation that

[2]Note that Geneva's syntax represents TCP segmentation with the `fragment` action with the `tcp` parameter.

Strategy 27: MB-RA: Segmentation Overload

```
[TCP:flags:PA]-fragment{tcp:-1:False}(
    fragment{tcp:-1:False}(
        ,fragment{tcp:-1:False}),
        fragment{tcp:-1:False}(
            fragment{tcp:-1:False},
            fragment{tcp:-1:False})
)-| \/
```

function, and hundreds more that do not.

Without a second SYN packet, at least 7 segments are required for this strategy to work, and further segmentation does not negatively affect the reliability of the strategy. Exactly reversing the order of the segments too is not a requirement; other variants of this strategy exist that defeat MB-RA without defeating MB-R without this property. Previous researchers found that different parts of the GFW have issues reassembling segments less than 8 bytes long [2,70], but each segment in this example is at least 24 bytes long.

We originally hypothesized that this series of segmentations must simply split up the SNI field across multiple packets, but when this strategy is used, the SNI field is intact and unchanged in a single TCP segment. Leaving the SNI field intact is also not a requirement; other versions of this strategy that split the SNI field across multiple segments and work equally well. Frankly, we do not understand why this strategy defeats MB-RA.

7.2.2 Evading MB-RA and MB-R

Next, we will discuss strategies that can defeat both MB-RA and MB-R. Geneva discovered variants of the aforementioned Segmentation Overload strategy that defeat both MB-RA and MB-R simultaneously, with 99% reliability. Like before, this strategy performs multiple rounds of TCP segmentations on the TLS Client Hello packet to produce 7 individual packets, most of which are out of order. Again, it is not clear why this strategy works.

Strategy 28: MB-R & MB-RA: In-Order Segmentation Overload

```
[TCP:flags:PA]-fragment{tcp:-1:False}(
    fragment{tcp:-1:False}(
        ,fragment{tcp:-1:False})
    ,fragment{tcp:-1:False}(
        fragment{tcp:-1:True},
        fragment{tcp:-1:False})
)-| \/
```

The most salient difference between strategies that defeat MB-R compared to the previously discussed MB-RA-beating strategies is that these strategies contain at least one middle pair of segments that remain in-order. The location of the SNI field does not impact the reliability of this strategy; it can be included in any segment or be split across multiple segments.

Geneva also found that it could combine pieces of the In-Order Segmentation strategy to reduce strategy complexity. This next strategy works by duplicating the SYN packet and performing three TCP segmentations of the TLS Client Hello.

In our follow-up experimentation, we find that in order for this strategy to

Strategy 29: MB-R & MB-RA: Double SYN, Triple Segmentation

```
[TCP:flags:S]-duplicate-|
[TCP:flags:PA]-fragment{tcp:-1:False}(
    ,fragment{tcp:-1:False}(
        fragment{tcp:-1:True},)
)-| \/
```

defeat both MB-RA and MB-R, the segments must be sent in a specific order: the fourth segment must be sent first, then the second segment, then the third, and finally the first segment. Any deviation from this order causes MB-R to detect the sequence, though any order in which the first segment is not sent first is sufficient to evade MB-RA.

We verified that only *the order* in which the segments are sent matters, not the content or size of the segments. We manually tested different strategies that would make a single segment 188 bytes long (making each of the other segments just a single byte long); as long as the correct segment order is maintained, the strategy evades MB-RA and MB-R. We do not understand why these constraints apply.

We also rediscovered several strategies that researchers had found in the past for other components of the GFW [24, 36, 40]: TCB Teardowns (injecting a TTL-limited or checksum corrupted RST) and TCB Desynchronization (injecting a TTL-limited or corrupt checksum with data).

7.3 How does MB-R work?

Now that we have a robust way to trigger MB-R in isolation, we can explore how MB-R works. In this section, we report on MB-R's functionality.

Which packets from the client will MB-R act upon? We performed a series of experiments in which we instrumented a client to send a TLS Client Hello with a forbidden SNI field (such as wikipedia.org), followed by different client handshake messages or packet payloads, including empty packets, garbage messages, and HTTP payloads. We did not observe a response from MB-R for any non-TLS messages nor for ClientHandshakeFinished messages. We find that MB-R will only take action if it sees a ClientKeyExchange or ClientChangeCipherSpec.

Is MB-R bidirectional? Yes, both MB-R and MB-RA track connections that originate from both inside and outside of China. First, we confirmed that MB-RA is still bidirectional: we made requests from vantage points we controlled outside of China to our vantage points inside China, and in the opposite direction; in both cases, we can trigger MB-RA. Next, we tested if MB-R also monitors traffic inbound to China by sending multiple different packet sequences that evade MB-RA (in different ways) but trigger MB-R and confirmed that MB-R is also bidirectional.

What is the reliability of MB-R and MB-RA? Previous researchers have found that the GFW is not 100% reliable in its censorship (usually around 97%) [2,24,40]. To test the reliability of both the primary and secondary middleboxes, we sent 2,000 packet sequences with small sleeps in between for both MB-R and MB-RA from a vantage point outside of China to servers we controlled inside China, each from a fixed source port to a unique destination port. By observing which ports are interfered with, we can estimate the reliability of each middlebox.

We find that MB-R interfered with 87.0% of the connections, and MB-RA inter-

fered with 88.2% of connections. Interestingly, these numbers composed together explain the approximately 97% total reliability found by previous researchers [2]: the likelihood of both middleboxes failing is approximately 1.8%, for a total reliability of 98%.

What ports does `MB-R` monitor? Researchers in the past have reported that the GFW's SNI censorship middlebox (`MB-RA`) monitors all ports 1-65,535 [2]. To test which ports `MB-R` monitors, we conducted an experiment in which we sent the sequences of packets that trigger `MB-R` from our vantage points outside of China to servers we control within China on every destination port. For this experiment, we configured the server within China to drop all outbound `RST` and `RST+ACK` packets, so we expect any `RST` or `RST+ACK` packet received by our vantage points outside of China to originate from the middlebox. We also verified the sequence numbers of inbound `RST` packets to prevent any spurious `RST` packets from interfering with the experiment. To account for `MB-R` not being 100% reliable, for any port that did not elicit censorship, we repeat the packet sequences to confirm whether or not the failure was a fluke. We find that `MB-R`, like the already known `MB-RA`, monitors all ports.

Does `MB-R` monitor ESNI or omit-SNI? In 2020, researchers discovered that China had deployed a new censorship middlebox to censor uses of HTTPS with Encrypted SNI (ESNI) [36]. They found that the new ESNI censorship middlebox does not censor *omit-SNI* (Client Hello messages with the SNI field omitted), although other censorship middleboxes have been observed censoring omit-SNI [138].

They determined that this censorship middlebox was different from the already known `MB-RA` HTTPS (SNI) censorship middlebox and confirmed that `MB-RA` does not monitor or censor uses of ESNI or omit-SNI. Does `MB-R` censor ESNI or omit-SNI?

To test this, we modified the sequence of packets we discovered that trigger `MB-R`. In the first experiment, we replaced the forbidden SNI TLS Client Hello with a TLS 1.3 Client Hello with an ESNI extension. In the second experiment, we replaced the forbidden SNI TLS Client Hello with a TLS Client Hello with no SNI extension at all. We find that `MB-R` does not censor ESNI or omit-SNI connections.

Does `MB-R` middlebox have residual censorship? *Residual censorship* is a feature of some censorship middleboxes in which after a censorship event occurs between a pair of hosts, the censor continues to interfere with benign connections between them for a short amount of time [135]. Some prior work has reported that `MB-RA` has residual censorship [7], but other researchers have reported that this residual censorship may be specific to certain vantage points [135]. From our vantage points in China, we do not observe residual censorship for `MB-RA`: after a censorship event, future benign connections between the same pair of hosts are not affected.

To test if `MB-R` has residual censorship, we issued packet sequences that trigger `MB-R`, and then sent follow-up benign connections. We find the same result as `MB-RA`: we do not observe residual censorship. Unfortunately, like all censorship measurement research, we are limited in what vantage points we can access, and

absence of evidence for residual censorship at both of our vantage points is not evidence of its absence throughout the network. It is possible that MB-R's residual censorship varies by geographic location.

Does MB-R and MB-RA have the same blocklist? To test if MB-R and MB-RA have different blocklists, we downloaded CitizenLab's China (567 domains) and Global (1,435 domains) test lists [104] to see if there were any domains censored by one middlebox that was not censored by the other. For each domain on the test list, we sent trigger packet sequences for both MB-RA and MB-R from the vantage points we controlled in China to a vantage point outside of China with the test domain in the SNI field of the TLS Client Hello. We used a unique source port for each of these connections and our vantage points were configured to drop all outbound RST and RST+ACK packets, so we expect any RST or RST+ACK packets we receive to originate from the GFW. Note that since our vantage points do not experience residual censorship for MB-RA or MB-R, residual censorship is not a concern for this experiment. Since the reliability of MB-R and MB-RA are not 100%, we repeated this experiment 5 times. As long as a test domain triggers a middlebox at least once, we know it is censored.

We find that both middleboxes had the same response to all of the domains we tested; if MB-RA censored it, so did MB-R and vice versa. This experiment supports our theory that MB-R acts as a backup middlebox to MB-RA.

Where is MB-R deployed relative to MB-RA? To test where MB-R and MB-RA are located on the network, we performed an experiment in which we TTL limited

the packet trigger sequences for both MB-R and MB-RA. By repeatedly sending a trigger sequence of packets with increasing TTL values, we can see at what hop each middlebox performs traffic injection. We repeated this experiment from both of our vantage points inside of China destined to multiple vantage points outside the country and then again in the reverse direction. We find that MB-RA and MB-R were the same number of hops away from each test vantage point; this suggests that they are collocated on the network level. This finding aligns with a previous exploration of China's censorship middlebox, which also found that China collocated the censorship infrastructure for other protocols [2].

7.4 Ethical Considerations

We designed our experiments to minimize impact on other hosts and to minimize risk to other users. All of our experiments with MB-R and training with Geneva was done strictly between hosts we controlled and hosts not located in residential networks. Geneva does not spoof IP addresses and generates a fairly small amount of traffic while training [40]. We also followed the original experiment design of Geneva and evaluated strategies serially to limit the volume of data we sent at once.

7.5 Conclusion

In this chapter, I showed that China's SNI-based censorship has continued to evolve, and supported my thesis in the context of more complex middlebox deployments. We discover and report on the existence of a secondary SNI censorship

middlebox and show that the middleboxes can be studied in isolation.

It is somewhat surprising that China continues to invest in its SNI-based censorship, as TLS is evolving to incorporate encrypted versions with Encrypted SNI (ESNI) and Encrypted Client Hello (ECH). Indeed, China continues to do so, and they (as with other countries [136]) are working to block ESNI outright [36]. This indicates that there is not yet enough critical mass behind ESNI/ECH to make the collateral damage of blocking them prohibitively large for China. Until it is, SNI-based censorship will remain a threat.

Our work also uncovers a more fundamental finding: censors are employing censorship-in-depth not just by blocking multiple intersecting protocols but by deploying middleboxes that target the same protocol in slightly different ways. The techniques we presented in this chapter provide a potential path forward for understanding and evading these robust forms of censorship.

Collectively, these results show that automated, packet-manipulation-based censorship evasion can render censoring middleboxes ineffective at censoring, and that this thesis holds even as censors evolve and employ censorship-in-depth.

In the next chapter, I will demonstrate how packet manipulation attacks can be used to render middleboxes ineffective by coercing them to enforce their policy when they should not, to disastrous effect. Whereas Chapter 3-7 demonstrated that packet manipulation strategies can render middleboxes ineffective at censoring, the following chapters demonstrate another class of middlebox policies that can be rendered ineffective. In particular, I will show that automated techniques can discover how to *weaponize* middleboxes to launch attacks against innocent hosts.

Chapter 8: Weaponizing Censors for Amplification Attacks

In the previous chapters, I showed it is possible to trick middleboxes into failing to implement their policy when they should, but it still leaves open the question of the reverse: can middleboxes be coerced into taking action when they should not? In this chapter, I show that this indeed, middleboxes can be rendered ineffective in this way, and that by doing so, middleboxes can actually be leveraged to launch startlingly effective volume-based reflected denial of service attacks.

Volume-based distributed denial of service (DDoS) attacks operate by producing more traffic at a victim's network than its capacity permits, resulting in decreased throughput and limited availability. An important component in the arsenal of a DDoS attacker is the ability to *amplify* its traffic. Instead of sending traffic directly to a victim V, the attacker spoofs V's source address, sends b bytes to some *amplifier* host A, who then "replies" to V with $\alpha \cdot b$ bytes for some $\alpha > 1$. In this manner, the attacker hides its IP address(es) from the victim, making it difficult to simply filter the attack traffic at a firewall, and increases its effective capacity by the *amplification factor* α.

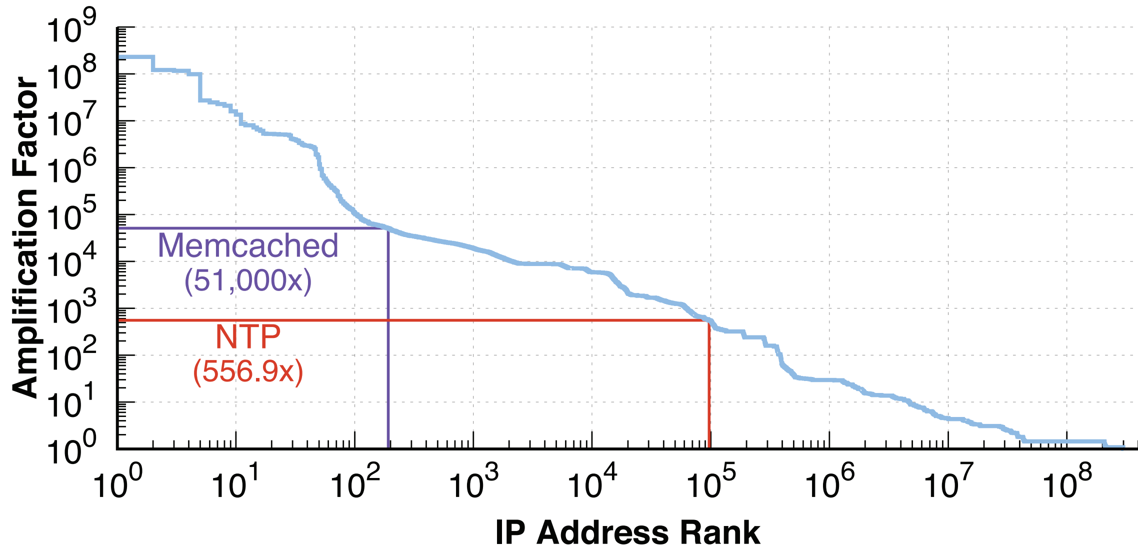


Figure 8.1: The maximum amplification factor we obtained per IPv4 address, based on several Internet-wide scans. (Note: the axes are log-scale.)

Some reflected amplification attacks can elicit impressive amplification factors. Among the most notable, DNS has been shown to have an amplification factor of 54, while NTP offers up to 556.9 [139]. Misconfigured Memcached [140] servers can provide amplifications over 51,000 [141, 142], and were used against Github in 2018 in the largest known DDoS attack to date, achieving 1.35 Tbps at peak [143].

To date, almost all reflected amplification attacks have leveraged UDP. In fact, to the best of our knowledge, there are *no* known TCP-based reflected amplification attacks that send beyond a single `SYN` packet.[1] This is because such attacks appear virtually impossible: to go beyond the `SYN` would seem to require an attacker to (1) guess the amplifier's 32-bit initial sequence number (ISN) in their `SYN+ACK` packet[2] and (2) prevent the victim from responding to the amplifier with a `RST` [37].

In this chapter, we show that it is indeed possible to launch reflected amplifi-

[1]We discuss *non-reflected* TCP-based amplification attacks in Section 8.1.
[2]We will use `+` to denote when a single packet has multiple TCP flags set.

cation attacks with TCP beyond a single `SYN` packet without having to guess initial sequence numbers. The key insight is to not elicit responses from the destination, but rather from *middleboxes* on the path to the destination.

Many middleboxes (especially nation-state censors) inject block pages or other content (such as `RST` packets) [24, 86, 126, 144] into established TCP connections when they detect forbidden requests. Moreover, because middleboxes cannot rely on seeing all packets in a connection [50], they are often designed to operate even when they see only one side of the connection. Our attacks tend to leverage *non-compliant middleboxes* that respond without having to observe both ISNs. Our measurements show that such middleboxes are surprisingly common on today's Internet, and that they can lead to amplification factors surpassing even many of the best UDP-based amplification factors to date.

We introduce a novel application of a recent network-based genetic algorithm [40] that *discovers* sequences of TCP packets that elicit large amplification factors from middleboxes.

We perform a series of IPv4-wide scans of the Internet using ZMap [145], to identify how many hosts can serve as amplifiers and quantify their amplification factor. Figure 8.1 provides an overview of the maximum amplification factor we were able to get from all IP addresses after several Internet-wide scans. We find 386,187 IP addresses that yield an amplification factor of at least 100×; 97,079 IP addresses that elicit a larger amplification factor than the infamous NTP attack [139], and over 192 IP addresses that responded with a higher amplification factor than Memcached [142].

Compared to `SYN`-only reflective amplification attacks, our attack identifies two orders of magnitude more IP addresses [146,147], and we also find amplification factors above 2,500×.

In fact, we find many hosts that effectively have an *infinite* amplification: in response to one or two attack packets, these machines respond at their full capacity indefinitely (barring packet drops) without any additional attacker involvement. Czyz et al. [148] observed similar behavior when studying NTP amplification, and called such hosts "mega-amplifiers." We at last answer the open question of why some hosts provide such abnormally high amplification factors: we show that many are actually sustained *by the victims themselves*, and others are due to routing loops.

Collectively, our results show that there is significant, untapped potential for TCP-based reflective amplification attacks. To enable this new area of study, we have made our code publicly available at `https://geneva.cs.umd.edu/weaponizing`.

Contributions We make the following contributions:

- We introduce a novel application of genetic algorithms to discover and maximize the efficacy of TCP-based reflective amplification attacks, and identify 5 attacks in total.
- We scan the IPv4 Internet to determine how many IP addresses can be used as TCP-based amplifiers, and their amplification factor.
- We confirm that these amplified responses typically come from network middleboxes, including government censorship infrastructure and corporate firewalls.
- We resolve the open question of the root causes of "mega-amplifiers." We attribute

them to infinite routing loops and what we call "victim-sustained amplification", in which victims' default responses (`RST`s) actually induce the reflector to send more data without additional effort from the attacker, leading to virtually infinite amplification.

The rest of this chapter is organized as follows. I provide additional background on DDoS attacks specifically in §8.1. In §8.2, we present novel techniques for discovering new TCP-based amplification attacks, and the results from applying these techniques to live censoring middleboxes. Next, I describe our methodology (§8.3) and results (§8.4) from scanning the entire IPv4 Internet with our newfound attacks. I explore "mega-amplifiers" in §8.5. I discuss ethical considerations and our responsible disclosure in §8.6, potential countermeasures in §8.7, and conclude this chapter in §8.8.

8.1 Background

Here, we define our threat model and review details of TCP and in-network middleboxes that are relevant to our attacks.

Threat Model To maximize the applicability of our attacks, we make very few assumptions about the adversary's capabilities. In particular, we assume a completely *off-path* attacker: it cannot eavesdrop, intercept, drop, or alter any packets other than the ones destined to it. We also assume that the attacker has the ability to source-spoof its victim's IP address. This would not be possible if the attacker's network performs *egress filtering*—that is, if it verified that the packets leaving its

network had IP addresses originating from within its network—but egress filtering is still not yet widely deployed in practice [146, 149, 150].

TCP Basics To ensure in-order delivery of bytes, both ends of a TCP connection assign 32-bit *sequence numbers* to the bytes they send. TCP connections begin with a *three-way handshake*, during which the end-hosts inform one another of their (random) initial sequence number (ISN). In a standard three-way handshake, the client sends a `SYN` packet containing its $\text{ISN}_{\textsf{client}}$, to which the server responds with a `SYN+ACK` that contains both its own $\text{ISN}_{\textsf{server}}$ and $\text{ISN}_{\textsf{client}} + 1$ to acknowledge the client's ISN. Finally, the client acknowledges $\text{ISN}_{\textsf{server}}$ by including it (plus one) in an `ACK` packet. Following this, a typical client sends a `PSH+ACK` packet containing its application-layer data (e.g., an HTTP GET request).

For a TCP connection to complete, the ISNs must be acknowledged with perfect accuracy. If the client were to send an `ACK` acknowledging anything but $\text{ISN}_{\textsf{server}} + 1$, the server would not accept the connection.

TCP-based Reflection Attacks In a *reflection* attack, an adversary sends to a destination r a packet that spoofs the source IP address to be that of victim v. As a result, r will believe v sent the packet, and will send its response to v. Reflection can be useful to hide the attacker's identity from the victim, and is commonly used when the reflector r is also an amplifier, sending more data to v than r received from the attacker.

Note that an adversary within our threat model cannot feasibly complete a three-way handshake in a reflection attack. The adversary would send the `SYN`

while source-spoofing v, and thus the server's SYN+ACK—with $\text{ISN}_{\text{server}}$—would be sent to v, not the attacker. To complete the handshake, the attacker would have to send a source-spoofed ACK, but would only have 2^{-32} chance of guessing the correct $\text{ISN}_{\text{server}}$. Moreover, even if the adversary were to guess $\text{ISN}_{\text{server}}$, the victim (if online) will respond to the server's spurious SYN+ACK with a RST, thereby tearing down the connection at the server.

Given these challenges, prior work assumed that TCP-based reflection attacks were limited to the initial handshake, in which the attacker sends a source-spoofed SYN and does not try to guess the appropriate ACK, let alone send an application-layer PSH+ACK [146,147]. Kührer et al. [147] showed that a single TCP SYN can result in a surprising amount of amplification. Compliant servers amplify a small amount because they retransmit SYN+ACKs a handful of times, until they timeout, receive the appropriate ACK, or receive a RST from the victim. Kührer et al. also found a few non-compliant machines on the Internet that respond to SYNs with many more packets, affording a greater amplification [146, 147].

In this work, we discover that *middleboxes* enable more sophisticated TCP-based reflected attacks beyond a single SYN. Compared to prior work, these new middlebox-enabled attacks yield even higher amplification rates and provide larger numbers of amplifiers that attackers can use.

Why should we think middleboxes might be vulnerable to this attack? Middleboxes often track the content of connections across multiple packets to handle re-ordered or dropped packets. However, middleboxes may not see packets in both

directions. This is because the Internet can exhibit *route asymmetry*, whereby packets between two end-hosts may traverse different paths [151]. Consequently, a middlebox may only see one side of a TCP connection (e.g., the packets from client to server). To handle this asymmetry, middleboxes often implement non-compliant or partial TCP reassembly, allowing them to still block connections even though they don't see all of the packets in a connection.

Middleboxes' resilience to missing packets presents an opportunity to attackers: a reflecting attacker may not need to complete the three-way handshake so long as it can convince the middlebox that the handshake had been completed. Combined with the packets they inject—especially block pages—middleboxes could be attractive targets for reflected amplification. In the remainder of this chapter, we show packet sequences that trick middleboxes into responding, and we show that middleboxes can yield very large amplification factors.

Non-reflective and UDP Amplification Attacks Other amplification attacks abuse TCP but involve directly connecting to the victim. Sherwood et al. [152] showed an attacker can use *optimistic acknowledgments* to induce a server to send a file at higher rates, ultimately DoSing *its own network*. The Great Cannon injects Javascript into Baidu webpages, turning visiting browsers into denial of service bots [153]. Our attack is effectively the reverse: instead of a censor co-opting the bandwidth of users to perform an attack, an attacker can co-opt the bandwidth of the censor.

Reflected UDP attacks have been studied extensively [139,140,154,155]. How-

ever, we are the first to study the use of middleboxes as reflectors.

Victim-sustained Attacks As we will see later in this chapter, we discover a mechanism by which an attacker attacks a victim in such a way that the victim themselves sustains the attack. Sargent et al. [156] identified 79 hosts that respond to a particular IGMP request by repeating the request. Ostensibly, source-spoofing this request could cause an infinite loop between two such hosts, and is thus similar to our victim-sustained attacks in §8.5. Our attacks are more widely applicable, since they rely on standard client behavior (sending RSTs to unsolicited packets); and as a result we identified several orders of magnitude more targets of victim-sustained infinite amplification. However, their findings motivate applying tools like Geneva at the *application layer* to discover application-specific bugs.

8.2 Discovering TCP-based Reflection Attacks

In this section, we present the first non-trivial, TCP-based reflected amplification attacks. We present a novel way to automatically discover new amplification attacks (§8.2.1), train it against a set of censoring middleboxes (§8.2.2), and report on the amplification attacks we discovered (§8.2.3).

8.2.1 Automated Discovery of Amplification

Our goal is to identify sequences of packets that will elicit amplified responses from middleboxes, without requiring us to establish a legitimate TCP connection or guess ISNs. This requires identifying non-compliant TCP behavior. Unlike

UDP [148] or TCP `SYN`-based [147] reflected amplification attacks—which take advantage of weaknesses in protocol designs—we must find weaknesses in TCP *implementations*.

We make two modest changes to Geneva to find new amplification attacks against middleboxes:

Initial Packet Sequence Geneva operates by manipulating an existing packet sequence, such as a real client's packets as it browses the web. To discover new amplification attacks, we use a single `PSH+ACK` packet with a well-formed HTTP GET request with the `Host:` header set to a given URL (we describe which URLs we use in §8.2.2). We chose HTTP as the input traffic because recent work demonstrated both how widely deployed HTTP filtering middleboxes are [126] and that many HTTP censors inject large block pages in response to small web requests [52].

Fitness Function Our goal is to find packet sequences that maximize amplification from middleboxes. The straightforward approach would be to set the fitness function to the amplification factor itself (number of bytes received divided by the number of bytes sent). However, we found that this sometimes encourages Geneva to try to elicit many small (e.g., `SYN+ACK`) packets from the end-host, rather than larger (e.g., block page) packets from middleboxes. To encourage Geneva to elicit responses specifically from middleboxes, our fitness function is the amplification factor, but ignoring all incoming packets that have no application-level payload. This optimization applies only to the fitness function; we report on *all* bytes sent and received in our results.

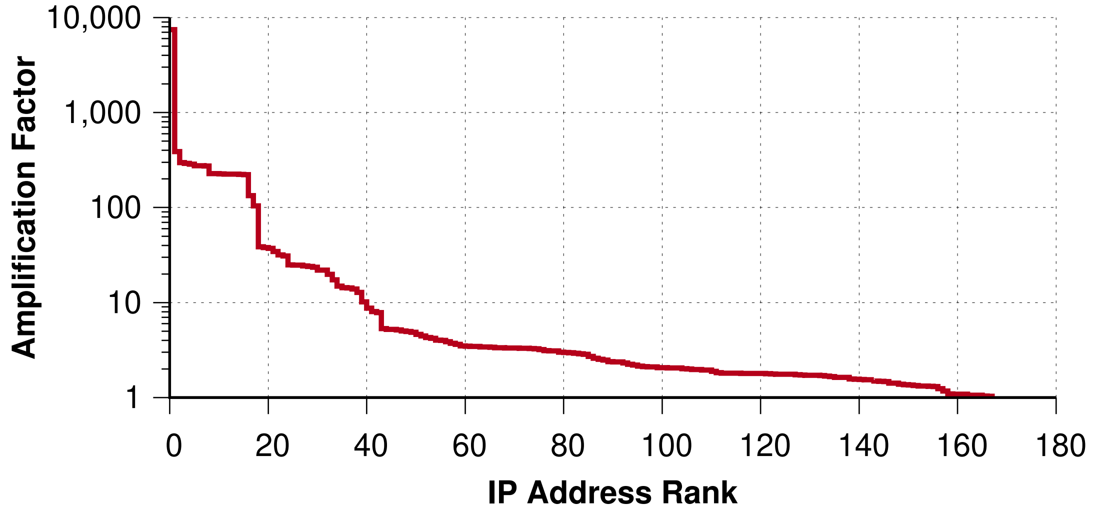


Figure 8.2: Rank order plot of maximum amplification factor from Quack-identified IP addresses. The maximum amplification factor was 7,455×.

8.2.2 Training Methodology

Geneva trains on live networks, and thus requires destination IP addresses to train against. To identify destination IP addresses that are likely to have middleboxes on the path from our measurement machine to them, we use data from Quack [52], a part of the Censored Planet [157] platform that performs active measurements of censorship. Quack regularly sends HTTP GET requests with potentially forbidden URLs in the `Host:` header to echo servers around the world, and detects injected censorship responses from middleboxes.

We use Quack's daily reports [54] to find endpoints that are likely to have middleboxes on the path, and the URLs likely to trigger them. We downloaded Quack's March 28th, 2020 dataset and extracted the IP addresses that experienced HTTP injection interference. This identified 209 IP addresses with active censoring middleboxes on their path, along with the offending URLs. We began training

against them on March 29th.

To train Geneva with an IP address from Quack's data, we set the destination of the generated traffic to the IP address, and set the `Host:` header in the HTTP GET request to one of the URLs that triggered interference to this IP address.

We let Geneva train for 10 generations with an initial population of 1,000 randomly generated strategies[3]. Training took approximately 25 minutes per IP address. To limit our impact on the network, we spaced our experiments out over four days; we sent each end-host just 2.8 Kbps of traffic on average (comparable to Quack's scans).

Before each experiment, we repeated Quack's methodology to the destination IP address to confirm it is still experiencing interference, and we skipped IP addresses that we did not experience interference. During our experiments, 25 of the 209 IP addresses (11.9%) stopped responding or no longer experienced interference, consistent with the churn rates seen in Quack's original experiments [52]. This left 184 IP addresses with active censoring middleboxes that Geneva trained against. Next, we present the packet sequences Geneva discovered.

8.2.3 Discovered Amplification Attacks

For 178 (96.7%) of the 184 IP addresses from the Quack dataset, Geneva found at least one packet sequence that elicited a response, and achieved an amplification factor greater than 1 for 169/178 (94.9%). Figure 8.2 shows the maximum amplification factors we discovered across all of these 169 hosts. Some of the middleboxes

[3]We forgo a full hyperparameter sweep to limit our impact on end hosts.

Strategy	Response %	Max Amplification
⟨SYN; PSH+ACK⟩	69.5%	7,455×
⟨SYN; PSH⟩	65.7%	24×
PSH	44.6%	14×
PSH+ACK	33.1%	21×
SYN (with GET)	11.4%	572×

Table 8.1: TCP-based reflected amplification attacks discovered against 184 Quack servers. Each packet with the PSH flag set includes an offending HTTP GET request in the payload.

provided high amplification factors: 17 (9.5%) had greater than 100×, and the maximum amplification factor was 7,455×.

We identify five unique packet sequences that elicit responses and five additional modifications to improve amplification factor. We summarize them in Table 8.1 and describe them in turn below.

8.2.3.1 Amplifying Packet Sequences

⟨SYN; PSH+ACK⟩ The most successful strategy we discovered sends a SYN packet (with no payload) with sequence number s, followed by a second PSH+ACK packet containing sequence number $s + 1$ and the forbidden GET request. Although this strategy comes at the cost of an entire additional packet, we find it to be highly effective at getting responses from middleboxes. It elicited responses from 128/184 (69.6%) of the middleboxes, with a maximum amplification factor of 7,455×.

From a middlebox's perspective, this packet sequence looks like a traditional TCP connection, missing the server's SYN+ACK and the client's ACK. As with normal TCP connections, the sequence number of the SYN is one less than the sequence number of the PSH+ACK. As discussed in §8.1, middleboxes must be resilient to asym-

metric routes, so it is expected that they would respond while missing the server's SYN+ACK. We note this sequence omits the client's ACK in a typical handshake, though the PSH+ACK may suffice to replace it. Geneva tried adding the client's ACK, but eliminated it during training—in follow-up experiments, we verified that adding the ACK had no effect on how the middleboxes responded.

⟨SYN; PSH⟩ This sequence sends a SYN with sequence number s (and no payload) followed by a PSH with sequence number $s + 1$ and the forbidden GET request as its payload. Note that this is the same as the ⟨SYN; PSH+ACK⟩ strategy, but with the ACK flag cleared in the second packet.

⟨SYN; PSH⟩ elicited responses from 121/184 (65.7%) of middleboxes, with a maximum amplification of 24×. Most (118, or 97.5%) of these also responded to the ⟨SYN; PSH+ACK⟩ sequence with the same amplification factors: those middleboxes appear not to be sensitive to the presence of the ACK flag on the packet containing the request. However, 10 middleboxes responded only when the ACK flag was set and 3 middleboxes responded only when it was not. We explore these differences more deeply with full IPv4 scans in §8.4.

We also explored if an additional ACK packet between the SYN packet and the PSH packet would improve response rate. Like with the ⟨SYN; PSH+ACK⟩ sequence, we found it had no effect on the middleboxes' responses.

PSH This sequence sends only a single packet: a PSH with the forbidden GET request. It elicited responses from 82 (44.6%) of middleboxes, with a maximum amplification factor of 14×. Note that this is the same as the ⟨SYN; PSH⟩ sequence,

without the SYN. All but one (98.8%) of the middleboxes that responded to just the PSH also responded to ⟨SYN; PSH⟩, indicating that the SYN was not necessary. For those hosts, avoiding the SYN resulted in an increase in amplification factor.

PSH+ACK This also sends a single packet: a PSH+ACK with a forbidden GET request. No TCP-compliant host should respond to this packet with anything besides an empty RST, as there is no three-way handshake. Still, 61 (33.2%) middleboxes responded with injected responses, with a maximum amplification factor of 21×.

This strategy is identical to the ⟨SYN; PSH+ACK⟩ sequence, minus the SYN packet. We find that all of the middleboxes that responded to a lone PSH+ACK also responded to the ⟨SYN; PSH+ACK⟩, with the responses of the same size. For those hosts, sending the additional SYN strictly decreases the amplification factor.

Most (51, or 83.6%) of the middleboxes that responded to PSH+ACK also responded to PSH; these middleboxes' responses were the same for both strategies, indicating no change in amplification. 10 middleboxes responded to PSH+ACK but not to PSH; these gave PSH+ACK its greatest amplification factor. However, 31 middleboxes responded to PSH but not PSH+ACK. Overall, PSH elicited more responses, but PSH+ACK elicited larger ones.

SYN with Payload This strategy sends the forbidden GET request as the payload of a single SYN packet. This elicited the fewest responses—21 (11.4%) of the middleboxes—but one of the largest amplification factors: 527×.

It is not common to send payloads in SYN packets[4], which led us to hypothesize that the middleboxes that responded to this might *only* be looking at the payloads.

[4] This is generally reserved for TCP Fast Open, which is rare in practice.

But this appears not to be the case: only 3 (14.3%) of the middleboxes that responded to `SYN` also responded to `PSH+ACK`, and only 6 (28.6%) also responded to `PSH`.

8.2.3.2 Packet Sequence Modifications

Geneva identified five additional modifications to the above packet sequences that improve the amplification factor for some middleboxes. One of these (increasing `TTL`s) never resulted in lower amplifications, and appear to be worth doing against all middleboxes. Four improve amplification for some middleboxes but lower it for others; to use such modifications in a practical setting, an attacker would ideally identify the middleboxes it uses ahead of time.

Increased `TTL`s Every IP header includes a time-to-live (`TTL`) field to limit the number of hops a packet should take; routers are supposed to decrement this at each hop, and drop the packet if the `TTL` reaches zero. Against one middlebox, Geneva learned to increase the `TTL` of both packets in the ⟨`SYN`; `PSH+ACK`⟩ sequence to its maximum value (255) to improve the amplification factor. It is very surprising that the `TTL` would have any impact on the amplification factor; the default `TTL` was already large enough to reach the destination.

To understand its root cause, we sent packet sequences to this middlebox with `TTL`s ranging from 0 to 255, and counted the number of responses for each. We find a perfectly linear relationship between `TTL` and amplification factor: we received $t - 13$ block pages for all `TTL` values $t \geq 13$. At the maximum `TTL` value (255), it

sent 242 copies of its block page!

This behavior can be explained by *routing loops* in the network of the censoring middlebox. Each time the packet sequence circles the routing loop, it re-crosses the censoring middlebox, causing it to re-inject its block page. That this only works for TTLs greater than 13 indicates that the routing loop is 13 hops from our measurement host. We show in §8.4 that routing loops are surprisingly common on the Internet at large, and they can be exploited by attackers for significant improvements to the amplification factor.

We found that setting a high TTL on packets has no effect on the response rate of any of the other packet sequences, so this modification can be made at no cost to freely exploit routing loops for maximum amplification.

Increased wscale Window scaling (or wscale) is a TCP option that controls how large the TCP window can grow. Geneva discovered an optimization that gets 7 (3.8%) more middleboxes to respond to the ⟨SYN; PSH+ACK⟩ sequence: setting the wscale TCP option in the SYN packet to an integer greater than 12. Based on the block page these middleboxes injected, we believe they are instances of Symantec's Web Gateway (SWG).

To understand this behavior, we sent the modified packet sequence 1,000 times to the candidate middleboxes in Quack's dataset, and repeated this experiment five times. Strangely, in each case, the middleboxes responded only ∼25% of the time. We could successfully ping the end-hosts behind each SWG with innocuous requests, suggesting that packet drops are not the root cause of the reduced response rate.

Varying the time between each packet sequence had no effect on the response rate, indicating we were not overloading the SWGs. The behavior is also not affected by packets sent by the end-host: if we limit the TTL of all of our packets such that they reach the middlebox but not the end-host, the middlebox still injects content to 25% of requests. Finally, altering the actual value of wscale had no effect on response rate. We do not understand why SWG is sensitive to this option.

Like with increased TTLs, increasing wscale had no adverse effect on response rates or sizes. However, because wscale is a TCP option, it requires additional bytes, thereby potentially lowering the amplification factor.

TCP Segmentation One modification Geneva identified for some middleboxes is to simply segment the forbidden GET request across multiple packets, either by adding an additional packet to single-packet sequences, or across the two packets in the ⟨SYN; PSH⟩ or ⟨SYN; PSH+ACK⟩ sequences. Geneva discovered that 5/184 (2%) middleboxes would send the block page a second time, once for each packet segment. For these middleboxes, this serves as an optimization for the amplification factor: although it comes at the cost of an additional packet with some payload, the payoff is a doubling in traffic elicited from the middleboxes. Strangely, this modification only works for two segments: any further segmentation causes two of the middleboxes to not respond, and the other three only send a maximum of two block pages.

Although this optimization can improve the amplification from middleboxes with this behavior, 26 others (14%) are unable to perform packet reassembly and stop responding entirely. Worse, for the middleboxes that do perform reassembly

and still respond, segmenting the request across multiple packets lowers the amplification factor.

`FIN+CWR` Another modification Geneva identified against four (2%) middleboxes was to change the TCP flags of the `PSH+ACK` packet in the ⟨`SYN`; `PSH+ACK`⟩ sequence to `FIN+CWR`. The `CWR` flag—"Congestion Window Reduced"—is used for TCP's Explicit Congestion Notification (ECN), and generally should not be combined with a `FIN` flag. The modified packet sequence elicits 12 copies of the middleboxes' block pages, each sent 0.4 seconds apart. The block page duplication increases the amplification factor of these middleboxes to 301×. If the `CWR` flag is not present on the packet, no response is sent. According to the injected block pages, these middleboxes appear to be instances of Fortinet Application Guard; this modification appears to only improve amplification factor for these middleboxes.

Shorter HTTP Geneva discovered an optimization against one middlebox: cutting off the four bytes in the HTTP GET request that immediately follow the forbidden URL (`\r\n\r\n`). Although this slightly improves the amplification factor for one middlebox, none of the other 183 middleboxes responded. This suggests that it is important for the HTTP GET request to be well-formed.

Failed Approaches We expected that changing the TCP `window` in our packet sequences might have an impact on amplification. Recall that TCP `window` size determines how much data the other endpoint can send before expecting an acknowledgement. However, we found that none of the middleboxes respected this TCP feature. Similarly, though TCP mandates that data sent should not exceed

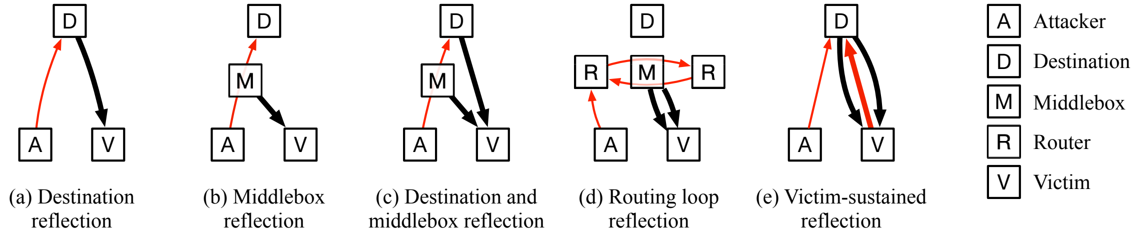


Figure 8.3: Types of attacks we find. Thick arrows denote amplification; red ones denote packets that trigger amplification. We find that infinite amplification is caused by (d) routing loops that fail to decrement TTLs and (e) victim-sustained reflection.

the maximum segment size (MSS) TCP option, every middlebox ignored this option.

8.3 Internet Scanning Methodology

We perform ZMap [145] scans of the IPv4 Internet to measure the effectiveness each of the attack packet sequences from §8.2.

Modifications to ZMap ZMap allows us to create arbitrary probe packets with the "probe modules"; we wrote a custom probe module for the packet sequences identified by Geneva. ZMap does not natively have the ability to send multiple distinct packets in each probe (e.g., `SYN` followed by `PSH+ACK`), so we modified ZMap to add this capability.

Selecting Forbidden URLs Quack's dataset contains 1,052 URLs that triggered censorship. Ideally, we could perform full Internet-wide scans for *each* URL and determine which ones produce the highest amplification. Unfortunately, this would take over 6 weeks of scanning at full 1 Gbps line rate per Geneva strategy, and would likely have diminishing returns.

Instead, we chose to estimate the smallest combination of URLs that collec-

tively elicit responses from the largest number of IP addresses. To do this, we construct every set of size $1 \leq N \leq 7$ of the 1,052 URLs from the Quack dataset, and for each set compute the number of Quack IP addresses it would have triggered.

We find the ideal set to be of size $N = 5$, each coincidentally from a different website category as identified by the Citizen Lab Block List [158]: www.youporn.com (pornography), plus.google.com (social networking), www.bittorrent.com (file sharing), www.roxypalace.com (online gambling), and www.survive.org.uk (sexual health services). These five keywords collectively elicit responses from 83% of the Quack IP addresses, after which there are diminishing returns (adding a sixth keyword only increased the response rate by 3.6%).

We acknowledge that the Quack dataset may not be representative of the entire Internet. Moreover, coverage of IP addresses is not necessarily the same as coverage of middleboxes; however, few IP addresses (4%) in the Quack dataset share the same /24 prefix, so we expect little middlebox overlap. It is possible that other keywords will elicit broader coverage or greater amplification; we leave this to future work.

Data Collection From April 9th to April 26th, 2020, we performed 5 sets of Internet scans, one for each mutually exclusive packet configuration (§8.2.3). For each set, we performed 7 Internet-wide scans: one for each of the 5 domains and our two control scans ("`example.com`", and no payload at all). To avoid saturating our link, we scanned at 350 Mbps; and each scan took approximately 2–4 hours. After each scan, we aggregated the number of bytes and packets we received from each IP

address that responded to our probes. Following convention, we include the size of the Ethernet header in the size of our probes and response packets when computing amplification factors.

8.4 Internet Scanning Results

This section presents the results of sending our attack packet sequences from §8.2 to the entire IPv4 Internet. We make two notes upfront that are important in understanding our results:

Responder variation Our packet sequences elicit a wide range of behaviors. We broadly classify them in Figure 8.3; for some destinations and packet sequences, we get response packets directly from destinations, from middleboxes (pretending to be the destination), or some combination of the two. We confirm in §8.4.3 that over 82% of the largest responses we receive come from middleboxes, but unfortunately it is difficult to perform this analysis for *every* destination IP address we send to. Thus, for consistency (and because middlebox de-aliasing is difficult and error-prone), we report on the number of destination IP addresses from which we can elicit responses throughout this chapter. We explore clustering and identifying middleboxes by their responses in §8.4.4.

Infinite amplification We discover many IP addresses that continue to respond, seemingly indefinitely, to our probes. The amplification factors for these IP addresses are technically *infinite*, but we report the (finite) amplification we obtained during our scans. These tend to be orders of magnitude larger than other hosts. We explore

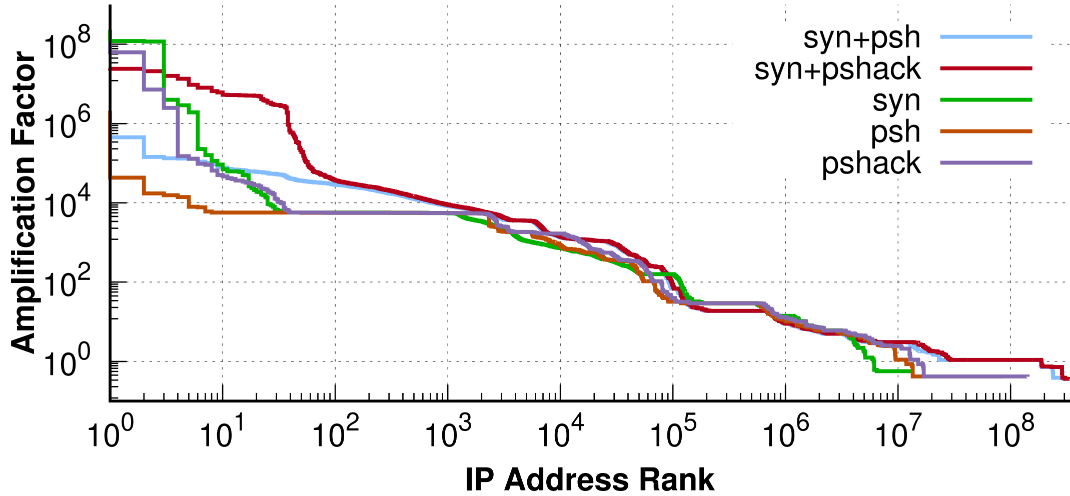


Figure 8.4: Rank order plot of the amplification factor received from each IP address for the triggering payloads containing `www.youporn.com` across all five packet sequences.

URL	SYN	PSH	PSH+ACK	⟨SYN; PSH⟩	⟨SYN; PSH+ACK⟩
www.youporn.com	49.4	4.4	23.2	13.9	**52.0**
roxypalace.com	5.8	4.4	16.5	13.6	**31.3**
plus.google.com	7.4	7.0	5.9	13.4	**14.9**
bittorrent.com	3.7	3.2	3.8	10.6	**13.7**
survive.org.uk	4.4	2.8	2.4	11.0	**11.2**
example.com	3.4	2.9	2.8	**11.2**	8.4
empty	**0.06**	0.01	0.02	0.05	0.06

Table 8.2: Total data received (GB) from the top 100,000 IP addresses for each combination of target URL and packet sequence. Bolded is the maximum value *for each target URL*.

infinite amplifiers in §8.5.

8.4.1 Which strategies work best?

We begin by measuring the impact that packet sequence and keyword have on response rate and amplification factor.

Figure 8.4 compares the amplification factors for each of the 5 packet sequences with the URL `www.youporn.com`. We immediately observe that each of

URL	SYN	PSH	PSH+ACK	⟨SYN; PSH⟩	⟨SYN; PSH+ACK⟩
www.youporn.com	116,120	**67,503**	**78,830**	**92,765**	97,689
roxypalace.com	**128,843**	52,168	63,080	86,010	97,213
plus.google.com	39,177	27,815	24,827	54,916	63,090
bittorrent.com	33,187	19,171	24,682	47,348	**193,754**
survive.org.uk	98,038	14,600	13,060	45,953	43,927
example.com	28,909	15,669	15,911	46,469	27,962
empty	65	27	49	42	59

Table 8.3: Number of IP addresses with amplification factor over 100× for each combination of target URL and packet sequence. Bolded is the maximum value *for each sequence.*

these strategies elicits responses from over 5M destination IP addresses with amplification greater than one. Moreover, we find that all of them elicit very large amplification factors; for each packet sequence, there are over 50,000 destination IP addresses that yield over 100×.

To focus on the heaviest hitters, Table 8.2 compares the total volume of traffic generated from the top 100,000 IP addresses for each scan, and Table 8.3 shows the number of IP addresses with amplification factor greater than 100×. ⟨SYN; PSH⟩ and ⟨SYN; PSH+ACK⟩ get responses from the largest number of unique IP addresses: 29× more than the SYN scan. Despite requiring an additional packet, they also yield higher amplification factors for most of the top 1,000 IP addresses, and elicited the highest total amount of traffic across every URL. Sending a SYN packet with a forbidden HTTP GET was surprisingly effective at eliciting responses: for half of the URLs, it had the most IP addresses with an amplification factor greater than 100×.

The choice of URL has a strong impact on how well a given packet sequence amplifies. Figure 8.5 shows the amplification factors from using each of the key-

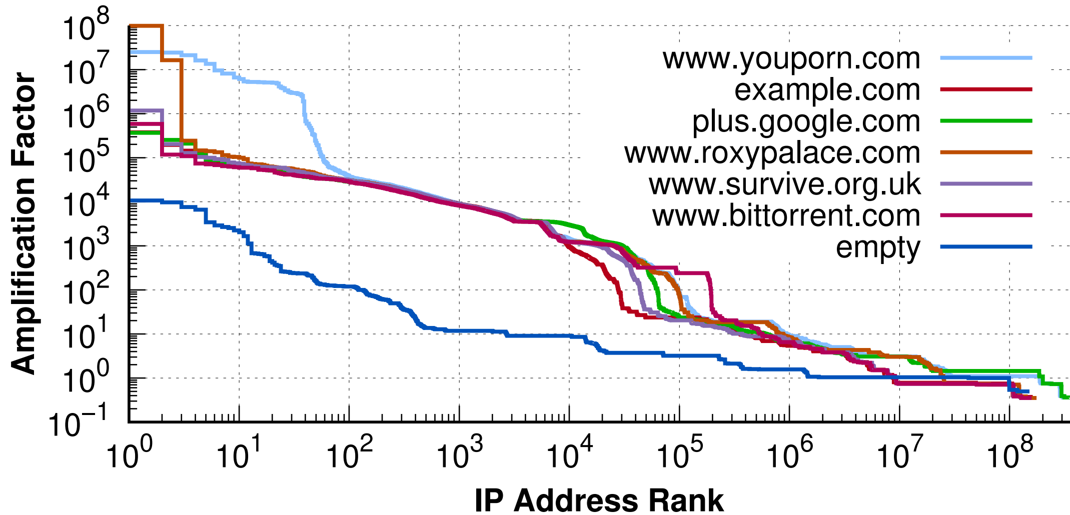


Figure 8.5: Rank order plot of the amplification factor received from each IP address for the ⟨SYN; PSH+ACK⟩ packet sequence across all seven scanning payloads.

word/strategy combination.

Overall, `www.youporn.com` was the most effective for eliciting the most responses, with two notable exceptions. First, `www.bittorrent.com` elicited double the number of IP addresses with amplification factor greater than 100×. The source of this is highly amplifying censorship of two networks with /16 prefixes: one run by the University of Ghent; the other, the City of Jacksonville, Florida. Second, `roxypalace.com` on `SYN` packets similarly elicited responses from more IP addresses than any other URL, and this is largely due to triggering the border firewall at Brigham Young University, which runs a /16 prefix.

Surprisingly, scans for the control keyword `example.com` trigger many amplifiers. It under-performed every other keyword in number of IP addresses and amount of data elicited, but thousands of IP addresses still responded with 20× amplification. It is possible the middleboxes who respond to this do so as a means of access control. Scans with an empty payload received the fewest amplifiers, smallest total

data elicited, and smallest total amplification: the ⟨SYN; PSH+ACK⟩ scan elicited *three orders of magnitude* more data than an empty SYN scan.

Summary The ⟨SYN; PSH+ACK⟩ packet sequence with www.youporn.com is overall the most effective at eliciting amplification, but other URLs and sequences are needed to trigger specific, large networks.

8.4.2 Are these actually amplifiers?

We next explore if these IP addresses can be (ab)used for real-world attacks. In a real attack, an attacker would not send just one trigger packet sequence; instead, she would repeatedly send trigger packet sequences to these IP addresses to amplify the response traffic. To test if the IP addresses we identify are true amplifiers, we perform an experiment with the top 1 million IP addresses with the highest amplification factor from the ⟨SYN; PSH+ACK⟩ scan with www.youporn.com keyword. Using ZMap, we perform two independent scans to these IP addresses: first, by sending 5 trigger packet sequences to each IP address, and second (as a control), just one trigger packet sequence[5].

Figure 8.6 presents the *increase factor*: the ratio of bytes we received from each IP address when sending 5 probes to the bytes received from 1 probe. Perfect amplifiers have an increase factor of 5×. Our results suggest that the majority of the top 1 million IP addresses are true amplifiers. Over 46% of IP addresses responded with exactly 5× as much data, and another 30% responded with between 2× and

[5]When sending multiple probes, we modify ZMap so that each probe is sent from a different source port, so the packets are not identical.

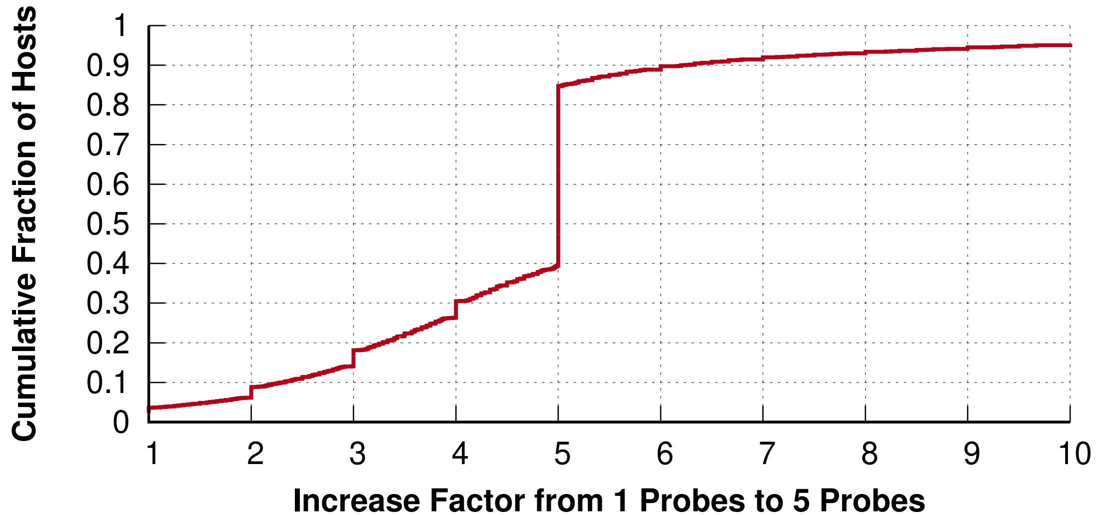


Figure 8.6: The increase factor in the number of bytes we receive between sending 5 probes and sending 1 probe. 46% of IP addresses responded with exactly 5× as much data.

5× as much data, likely representing amplifiers that missed or dropped one or more of our packets. Notably, many of the IP addresses that sent the most data do not increase by the same rate. Of the top 100 amplifiers, none of them increased by exactly a factor of 5×, and only 10 increased by 4–6×.

8.4.3 Are these middleboxes?

Next, we determine if the responses we receive are truly coming from middleboxes. We performed a traceroute using a custom ZMap probe module on the top million IP addresses by bytes received in our ⟨`SYN`; `PSH+ACK`⟩ `www.youporn.com` scan. Our ZMap module sent three `TTL`-limited TCP `SYN` packets for each `TTL` between 10 and 25 to each of the million hosts, and recorded the resulting ICMP `TTL`-exceeded messages. This allowed us to construct a (partial) traceroute for each target for hops 10–25. Out of the million targets, 99.5% provided at least one router hop, with an

average of at least 6 hops per traceroute.

For each target, we extracted the last hop that we received a `TTL`-exceeded message for (i.e., the last hop we learned on the traceroute to the target). We then sent a follow up ⟨`SYN`; `PSH+ACK`⟩ sequence with `www.youporn.com` to the target, but `TTL`-limited to the last known hop. This probe is certain to not reach the target, as it should generate a `TTL`-exceeded message by the last-hop router. Therefore, if we still receive a response from the endpoint, we can tell the response is coming from a middlebox along the path to the target, and not the target itself.

If we do not receive a response, we cannot conclude that responses normally come from the target endpoint, as it could be that our traceroute was incomplete: there may be a middlebox further along the path but still before the endpoint. However, we can interpret the presence of a response to our `TTL`-limited probe as confirmation that it was produced by a middlebox.

Figure 8.7 shows the results of this scan, binning IP addresses into bins of size 1,000 and plotting the fraction of the IPs in the bin that we identified as middleboxes. Overall, 36.8% of the 1M targets responded to our `TTL`-limited probe, positively confirming their responses were produced by a middlebox. Notably present, however, are two gaps in the graph in which almost no responses were received:

The small gap has ∼10,000 IP addresses (104,000 $\leq x \leq$ 114,000). All of these IPs are in three /20-sized subnets that belong to the Texas State Technical College Harlingen (TSTCH). Their responses correspond to block pages generated by a SonicWall network security appliance, a common middlebox we see in our data. It appears that TSTCH blocks traceroutes at its border, meaning that our

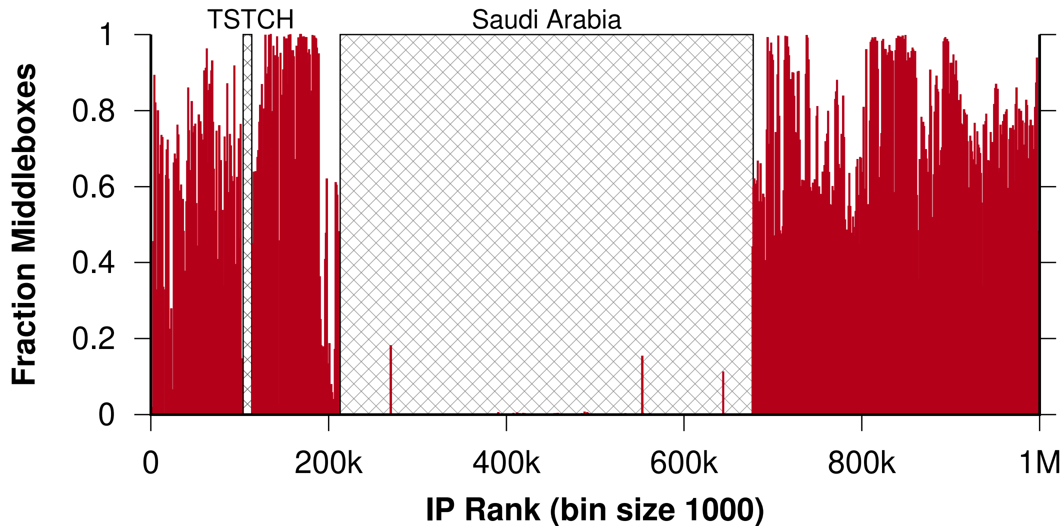


Figure 8.7: The fraction of the top million hosts that we confirm are middleboxes, using TTL-limited probe. The small gap at $x \approx 100{,}000$ and the large gap in the middle of the plot correspond to networks that block traceroutes at their borders. Accounting for this, we find injected responses from 82.9% of the top million IP addresses are from confirmed middleboxes.

last-observed traceroute hop occurs before the SonicWall appliance.

The larger gap has ∼465,000 IP addresses ($213{,}000 \leq x \leq 678{,}000$). 98.6% of them geolocate to Saudi Arabia. Looking at their traceroutes, their last hops comprise just 2,068 unique router IPs, with 90% of IP addresses sharing only 10 last-hop routers (all within Saudi Arabia). It appears that Saudi Arabia also blocks traceroutes at their border, preventing us from being able to traceroute into the country. However, the response that comes back from 97% of the IP addresses in this block corresponds to the standard block page of Saudi Arabian censorship, describing that the website is blocked, and also suggesting a middlebox is responsible for this response.

Conservatively labelling the 10,000 IP addresses from TSTCH and 97% of the 465,000 Saudi Arabian IPs as encountering on-path middleboxes increases the

percent of IPs that encounter on-path middleboxes to 82.9% of the million targets we scanned. We conclude that responses from the vast majority of IP addresses in our dataset are produced by middleboxes.

8.4.4 What kind of packets do amplifiers send?

We analyzed the packets we received in our ⟨`SYN`; `PSH+ACK`⟩ scan with `www.youporn.com`. This scan received a total of over 105 GB of data from 337 million IP addresses. For each IP address, we generate a *fingerprint* from the response packet sequence, consisting of a vector of `(TCP flags, packet size)` tuples; this allows us to efficiently group IP addresses that send us similar responses. We then counted the number of IP addresses that sent each fingerprint. We ignore order to allow for packet re-ordering.

Overall, we discover **63,662 unique fingerprints**. Each fingerprint represents a unique set of packets sent by amplifiers. The fingerprint returned by the most IP addresses is a sequence of three 54-byte `RST+ACK`s, which we received from approximately 154 million IPs. This is a well-known censorship pattern produced by the Great Firewall of China (GFW) [24,40], and using the MaxMind database [159], we find 99.9% of these IPs geolocate to China. We note this is weakly-amplifying, sending 162 bytes for our 149 byte probe.

The fingerprints representing the largest number of bytes are less common. For example, the top fingerprint is 528,007 410 byte `FIN+PSH+ACK` packets and 525,110 `RST+ACK`s, sent by a single IP address in India. We investigate these mega-amplifiers

Country	#Responsive IP addresses	% Sending fingerprint	Fingerprint
China	170,858,209	90.0%	3× RST+ACK (54)
S Korea	15,981,100	7.6%	PSH+FIN+ACK (119)
Iran	8,612,544	75.7%	PSH+FIN+ACK (402–405); RST+PSH+ACK (54)
Egypt	2,909,897	89.8%	RST+ACK (54)
Bangladesh	1,375,908	81.4%	PSH+FIN+ACK (248)
Saudi Arabia	894,858	45.3%	PSH+ACK (97); 2× PSH+ACK (1354)
Oman	596,546	94.7%	RST (54)
Qatar	387,625	89.4%	RST (54)
Uzbekistan	253,098	91.8%	FIN+ACK (74)
Kuwait	173,126	31.3%	PSH+FIN+ACK (114)
UAE	161,014	52.0%	RST (54)

Table 8.4: Nation-states with nation-wide censorship infrastructure and the fingerprint they most frequently respond to clients with. Numbers in parentheses denote packet sizes in bytes.

more in §8.5. The largest fingerprints sent by more than one IP address consist of a single SYN+ACK and multiple megabytes worth of PSH+ACK packets containing data. These appear to be sent by buggy TCP servers that simply respond to our non-compliant GET request with real data. We find approximately 746,000 IP addresses with this behavior.

8.4.5 Are these national firewalls?

We find that nation-state censorship infrastructure makes up a significant fraction of the TCP amplifiers we discover. Figure 8.8 breaks down the amplification we see for the top 5 countries by number of amplifying IP addresses. Out of these, all but the US have deployed nationwide Internet censorship infrastructure [160, 161], visible by long flat plateaus in the graph which indicate a large number of IP addresses with uniform amplification. The US is a notable exception, and we explore why it is so prevalent later in this section. Amplification factors vary significantly

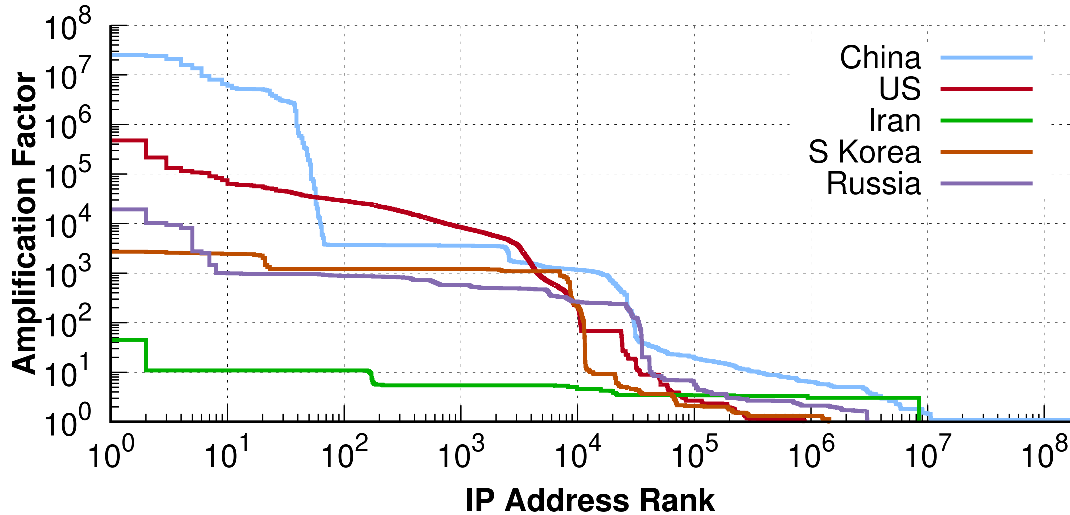


Figure 8.8: Rank order plot of the amplification factor by country for the `www.youporn.com` scan with the ⟨`SYN`; `PSH+ACK`⟩ packet sequence.

country-to-country due to different censorship methods.

By extracting fingerprints that were shared by many IP addresses that geolocate to the same country, we can identify censoring nation-states. For example, over a million IP addresses geolocate to Bangladesh and respond with a 248-byte `FIN+PSH+ACK`. Table 8.4 shows a sample of censoring countries and their most popular fingerprint. At a slightly higher amplification, we observe four similar fingerprints with two packets each: a 402–405-byte `FIN+PSH+ACK` and a 54-byte `RST+PSH+ACK`. We received these fingerprints from 8.6 million IP addresses in Iran, representing 76% of all the responding IP addresses that geolocate to Iran.

The censorship infrastructure of Saudi Arabia also shows prominently in our dataset: its fingerprint is three packets: a 97-byte `PSH+ACK` and two 1354-byte `PSH+ACK`s, offering an amplification factor of 18.9×. We received this fingerprint from over 400K IP addresses, 99% of which geolocate to Saudi Arabia, comprising 45% of all the responding IP addresses that geolocate to Saudi Arabia.

In general, we find the amplification factor from nation-state censors is small: most countries we surveyed provide less than 4× amplification. The GFW of China is the largest—but also the weakest—amplifier we find. Curiously, we find that the GFW has a *different fingerprint* between two of our scans: the ⟨SYN; PSH+ACK⟩ scan with plus.google.com elicited three RST+ACKs and a RST packet, but this extra RST packet is missing in scans for www.youporn.com. This RST was also absent when plus.google.com was sent with the ⟨SYN; PSH⟩ sequence. The presence of the RST raises the amplification factor of the GFW from 1.08× to 1.45×.

We do not understand why the GFW behaves differently between these keywords and sequences. Researchers have hypothesized that the RST+ACK and RST packets from the GFW originate from different, co-located censorship systems [24, 40]; our results support this theory, and even suggest that the block lists themselves can be processed differently between the two censorship systems depending on the sequences of packets.

We also discover hundreds of IP addresses in routing loops in Russia that contain censoring middleboxes with 250.9× amplification. The highest amplifying nation-state censors are two censoring ISPs located in Russia that seem to have *infinite* routing loops in their network, that sent us packets for weeks after our scans. We examine the effects of routing loops more closely next in §8.4.6.

Nation-state censors pose a more significant threat to the Internet than their amplification factor alone suggests. First, nation-state censorship infrastructure is located at high-speed ISPs, and is capable of sending and injecting data at incredibly high bandwidths. This allows an attacker to amplify larger amounts of traffic

without worry of amplifier saturation. Second, the enormous pool of source IP addresses that can be used to trigger amplification attacks makes it difficult for victims to simply block a handful of reflectors [162]. Nation-state censors effectively turn every routable IP addresses within their country into a potential amplifier.

While nation-state censors are well-represented in our amplifiers dataset, other large non-censoring countries, such as the US, are prevalent as well. Specifically for the US, we observe a more diverse set of fingerprints: over 13,000 unique fingerprints, compared to 7,553 in Russia, and under 3,000 from South Korea. This indicates a diversity of networks, rather than a coordinated, nationwide deployment. Indeed, we observe several university and enterprise firewalls that respond with identifiable and amplifying fingerprints.

These results demonstrate that nation-state censors enable TCP amplification attacks, but that they are far from the sole contributor to this problem.

8.4.6 Routing Loops

Routing loops are the result of network misconfigurations, inconsistencies, and errors in routing protocol implementations. Packets caught in a routing loop will typically eventually be dropped when their `TTL` reaches zero. However, even a finite routing loop can hypothetically have significant impact on amplification factor. Suppose an amplifying middlebox were in a routing loop; every time an offending packet traversed the loop, it would re-trigger the middlebox. Such a scenario would make the network self-amplifying: at no additional cost to an attacker, the effective

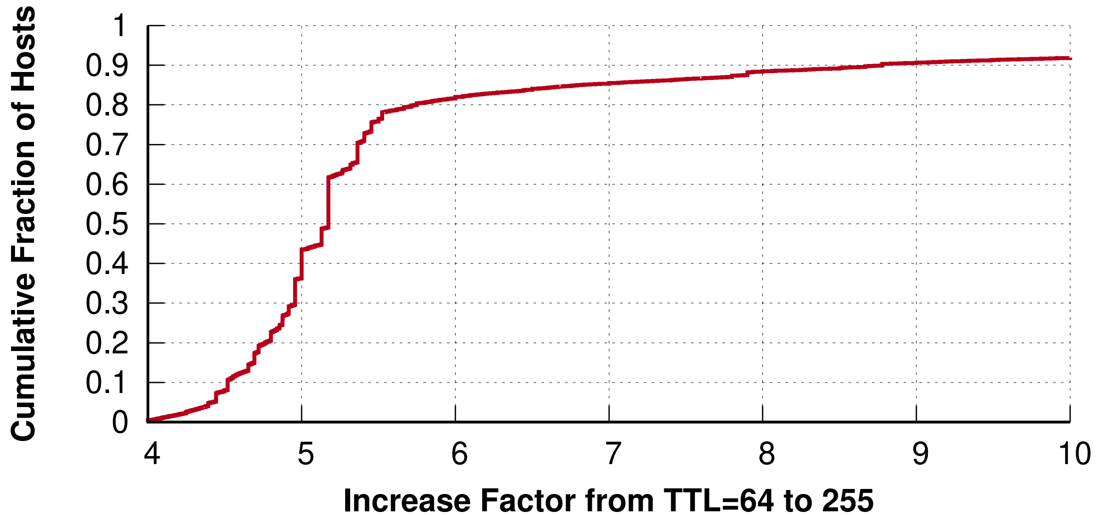


Figure 8.9: CDF of the increase factor in amplification of candidate looping IP addresses when scanned with a TTL of 255 and 64. Because the increase factor is affected by the number of hops away an IP address is, we expect routing loops to have an increase factor of at least 4. Larger increase factors are further away from our scanner, limiting the overall amplification factor from our perspective.

amplification rate of a middlebox would be increased by the number of times the packet crosses the middlebox in the routing loop.

The maximum value of TTL in the IPv4 header is 255, so the number of times a single trigger packet sequence can elicit responses from an RFC-compliant middlebox is $\ell(255 - d)$, where d is the number of hops between the attacker machine and the routing loop and ℓ is the number of times the packets traverse the amplifying middlebox per loop.

So far, our scans were conducted with a TTL value of 255, in accordance with the optimizations discovered by Geneva in §8.2. We performed follow-up scans with a reduced TTL value in order to observe which IP addresses send us a corresponding reduction in the number of packets, allowing us to identify which amplifiers involve routing loops.

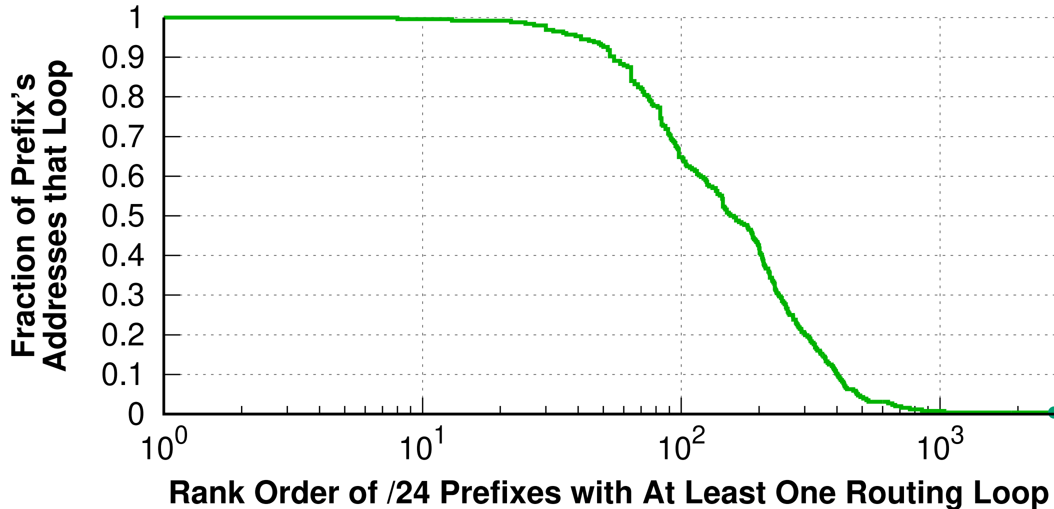


Figure 8.10: The /24 prefixes with at least one routing loop, rank-ordered by the fraction of their 256 IP addresses that we observe to loop. Of the 2,763 looping prefixes, 54 (2%) have over 90% of their IP addresses loop, but 1,705 (62%) have only one looping IP address. (Note that the x-axis is log-scale.)

For this experiment, we use the ⟨`SYN`; `PSH+ACK`⟩ packet sequence with the `www.youporn.com` trigger keyword. We use the top 1 million hosts (by number packets sent during the scans), and perform two follow-up scans to these IP addresses: one with the `TTL` set to 255 and one set to 64 (approximately 1/4 the value). As we are knowingly re-triggering machines with potentially enormous amplification factors, we reduced the scanning speed to 100 kbps[6].

We can identify routing loops by comparing the number of packets we receive per IP address across scans. For a routing loop d hops from our scanner, we expect a probe with $\texttt{TTL} = 255$ to receive $(255-d)/(64-d)$ times more packets than a probe with $\texttt{TTL} = 64$. Note that this value increases as d increases, and, for a routing loop, has a minimum value of $\sim$4 (when the routing loop is zero hops away). Therefore, we label an IP addresses as having a routing loop if it has an increase factor of at

[6]Despite our low send rate, we received back on average around 800 Mbps, representing a total amplification of 8,000× for this experiment.

least 4 and sent more than 10 packets when probed with a `TTL` of 255. From our top 1 million IP sample, we label **53,041 IP addresses as routing loop amplifiers** using this heuristic, spanning 2,763 distinct /24 prefixes. Figure 8.9 presents a CDF of the increase factor for these routing loop IPs.

Loops per subnet One would expect that if sending to a given IP address results in a routing loop, then all of the other IP addresses in its /24 prefix would experience a loop, as well. Surprisingly, we find that 62% of /24 prefixes with at least one routing loop have *exactly* one loop. Figure 8.10 shows the fraction of IP addresses found in each looping /24 prefix. Only 54 subnets have over 90% (231 of 256) of their IP addresses show evidence of being a routing-loop amplifier. On the other hand, 81.2% (2,244) of looping prefixes have fewer than 10 looping IP addresses. This means that even if an attacker can elicit responses from a middlebox by sending packets to any IP address that routes through it, she may only be able to take advantage of routing loops to a small number of IP addresses.

8.5 "Mega-amplifiers"

In our scans, we identify a surprising number of hosts that send enormous amounts of data in response to a single packet sequence—on the order of many gigabytes. We believe these are the same "mega-amplifiers" that Czyz et al. [148] reported in 2014. We identify two phenomena that contribute to mega-amplification: self-sustaining amplifiers and victim-sustained amplifiers.

Self-Sustaining Amplifiers *Self-sustaining* amplifiers are IP addresses that, once

triggered, continue sending data indefinitely. In our scans, we have observed these continuing for *weeks* after our probes. We hypothesize the cause of self-sustaining amplifiers is infinite routing loops: routing loops between middleboxes that do not decrement `TTL`s.

An infinite routing loop suggests these amplifiers are sending responses at the maximum capacity of their links. To confirm, we sent a packet sequence to a self-sustaining amplifier we identified in an ISP's censorship system in Russia. A short time later, we sent the same packet sequence from a different vantage point, and we recorded the bandwidth received from each. Figure 8.11 shows the bandwidth we received on both vantage points during our experiment. When we send a probe from a second vantage point, the response bandwidth was split equally between them.

We were unable to terminate the barrage of packets sent to us by this amplifier. We sent `RST` packets, and also tried `FIN+ACK`, `FIN`, `RST+ACK`, and ICMP `port unreachable` messages with no effect. Ultimately, the traffic stopped after approximately six days to the first vantage point, and 22 hours for the second. We believe the reason they finally stopped was because the routing loop eventually dropped a packet.

Fortunately, we find very few self-sustaining amplifiers: only 19 IP addresses sent data continuously. We identified 6 IP addresses (each in a different /24 prefix) located in China that sent the known censorship pattern from the GFW indefinitely, possibly indicating a loop across the GFW itself. Two ISPs in Russia also sent block pages indefinitely.

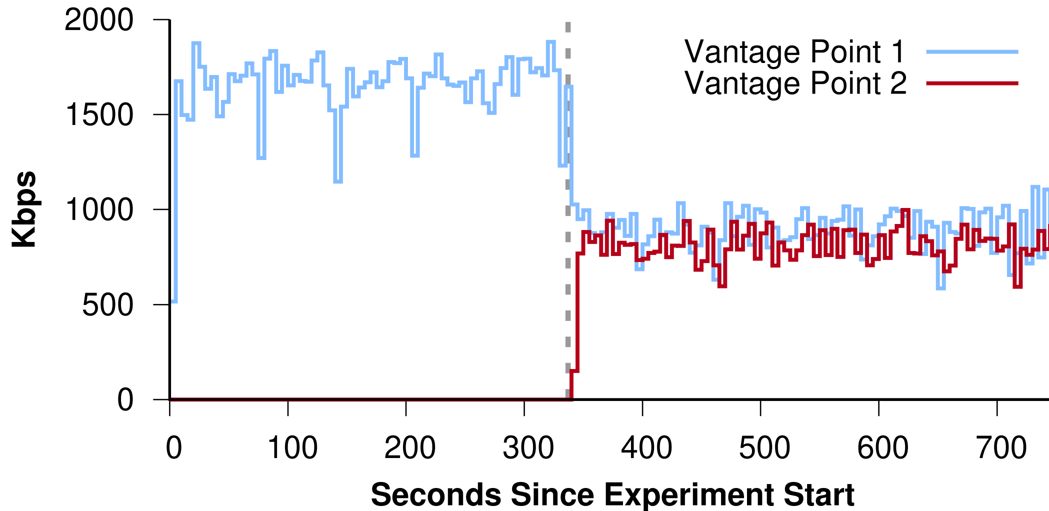


Figure 8.11: Attack bandwidth received at two vantage points from a self-sustaining amplifying IP address, which (based on its block page) appears to be a component of a Russian ISP's censorship system. The dashed line marks when the packet sequence was sent from the second vantage point. Note how the bandwidth we get from the system is divided evenly between the vantage points. This experiment supports our hypothesis that self-sustaining amplification is caused by an infinite routing loop.

Victim-Sustained Attacks The TCP standard says that when a host receives an unsolicited non-`RST` packet, it should send a `RST` packet in response [100]. For TCP amplification victims, this means they will send `RST` packets for any received (amplified) traffic. Normally, victim-generated `RST` packets have no effect on middlebox amplifiers[7].

However, our scans identify amplifying IP addresses that send an *additional response* to `RST` packets instead of ignoring them. This causes the victim to send another `RST`, inducing more responses, and so on. This packet storm continues indefinitely until a packet is dropped.

By default, our scanning machine sent outbound `RST` packets in response to data, thereby eliciting additional packets from victim-sustained amplifiers. To ex-

[7]Conversely, they may serendipitously halt `SYN`-based amplification attacks that target end-hosts [146, 147].

plore the effect that outbound RST packets have on amplification factor, we perform two additional scans: one with outbound RST packets turned off for the www.youporn.com keyword in the ⟨SYN; PSH+ACK⟩ sequence, and one with RSTs enabled (default). Figure 8.12 shows a comparison between these two scans. Dropping outbound RST packets has the effect of lowering the amplification factor for the top amplifying IP addresses, while raising the amplification factor of many IP addresses in the "long-tail".

We find several thousand IP addresses that behave this way, which we classify into two classes: censoring repeaters and "acknowledgers".

For **censoring repeaters**, we find 4,154 middleboxes that re-send a block page in response to a RST. This appears to be a buggy flow-tracking middlebox that, once a TCP flow triggers blocking, will continue injecting its block page in response to *any* subsequent packet, including RSTs.

For **acknowledgers**, we find 10,645 IPs that respond with an ACK to both data payloads and subsequent RST packets. This behavior is also not TCP compliant. To investigate what operating systems these "acknowledgers" are, we performed Operating System (OS) identification nmap [163] scans on 500 randomly sampled victim sustained IP addresses. Of the 452 (90.2%) IP addresses with a successful OS match, 267 (59%) were Dell SonicWall NSA 220. We believe this firewall model is to blame for most of the acknowledger victim-sustained behavior: the next most common OS match was Linux 2.6[8], with only 14 hosts (3%).

[8] We note this is not standard Linux 2.6 behavior.

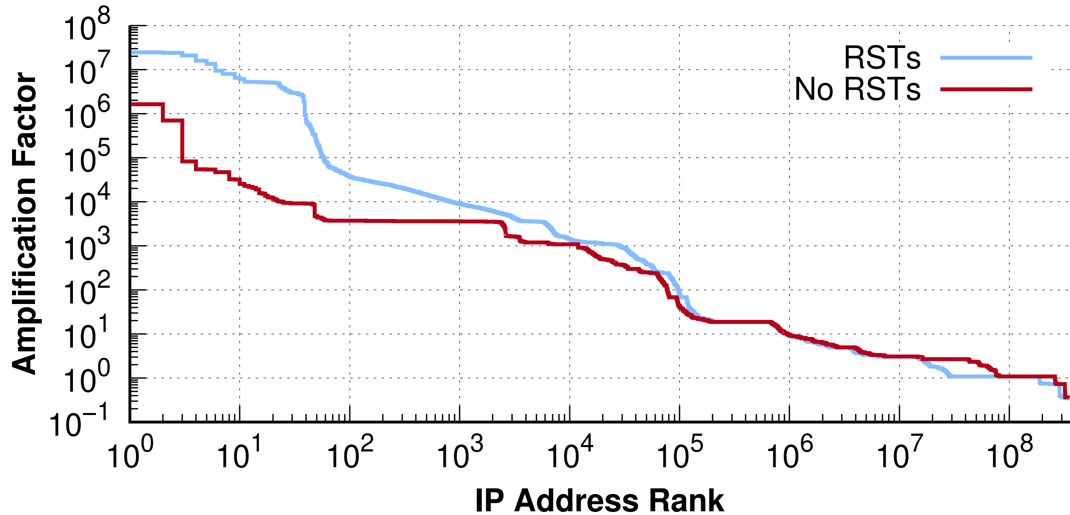


Figure 8.12: Rank order plot of amplification factor of two scans for the `www.youporn.com` keyword requested with the ⟨`SYN`; `PSH+ACK`⟩ packet sequence: one with outbound `RST` and `RST+ACK` packets being dropped and the other normally.

8.6 Ethical Considerations

Internet Scanning We followed best practices for scans as outlined by ZMap and Quack [52, 145]. We set up reverse DNS and hosted a webpage on the IP address we performed scans from, explaining the purpose of our scans. We also listed an email address to receive complaints and allow people to opt out of future scans. We received 8 removal requests over the course of our study comprising 2.1 million IP addresses which we removed from our scans.

Censorship-focused Internet-wide scans require additional careful considerations to avoid causing harm or falsely implicating users in making censored requests. In prior work on active probing to trigger censorship, researchers used alternative techniques to avoid having clients in censored countries make requests for banned content [52, 161, 164, 165]. Similarly in our work, the requests are made by our

scanning machine from outside the censored countries to all IPv4 addresses, making it unlikely that a government would punish any individual, due to the directionality and ubiquity of the scans. The packet sequences we probe with are non-TCP compliant and do not induce any in-country clients to make sensitive requests in response. For these reasons, we believe wide-scale scans of this nature pose minimal risk to individuals in censored regions.

Saturation Experiments A natural question with all amplification studies is: at what point do amplifiers' link saturate? For example, a single host with amplification factor of 5,000× may not be very valuable if it only has a 100kbps uplink.

Measuring the saturation of a specific amplifier requires sending the triggering packet sequence in rapid succession and measuring the response it triggers. For ethical reasons, we do not perform such an experiment. These experiments would effectively perform denial of service attacks against the specific middlebox or the IP address, or could adversely impact other networks on path.

We unintentionally triggered mega-amplifiers, and report on our findings in this chapter. However, after discovering these IP addresses and the nature of their responses, we removed them from future scans.

Responsible Disclosure Responsibly disclosing our findings is challenging given the large number of potentially affected vendors and network operators. It is both difficult to fingerprint specific vendors or manufacturers of middleboxes, and also difficult to identify the networks where middleboxes are responding from, as they spoof their source IP address by design.

Nonetheless, we attempted to reach out to both operators and vendors of middleboxes we discovered in our study. We contacted several country-level Computer Emergency Readiness Teams (CERT) that coordinate disclosure for their respective countries, including China, Egypt, India, Iran, Oman, Qatar, Russia, Saudi Arabia, South Korea, the United Arab Emirates, and the United States. We also reached out to several middlebox vendors and manufacturers, including Check Point, Cisco, F5, Fortinet, Juniper, Netscout, Palo Alto, SonicWall, and Sucuri.

We also publicly provide a repository of scripts that can help manufacturers and network operators test their middleboxes for amplifying behavior.

8.7 Countermeasures

Unlike previous amplification attack vectors [139, 140, 148], our attack is not isolated to a specific protocol and impacts a wide range of implementations and devices. Unfortunately, this means there is no single vendor or network that can be patched to correct the problem. Instead, this issue is systemic to middleboxes, particularly those that must operate seeing only one side of a connection.

Nonetheless, we offer potential remedies that can eliminate or partially mitigate amplification attacks, for both middleboxes and potential victims.

8.7.1 Middleboxes

Connection directionality While many middleboxes see asymmetric sides of a connection (e.g., only traffic to the server), there are others that see both sides,

such as middleboxes deployed at the gateways of networks. These middleboxes can accurately infer if a connection is live and only inject content if the three-way handshake is valid. We recommend such middleboxes require seeing traffic in both directions (to client and to server), and only inject block pages if this condition is met. This makes it more difficult for an attacker to spoof a connection, as it is infeasible for them to get both sides of a spoofed connection to pass by the same middlebox to induce injection. However, this solution will not work for large-scale middleboxes that sit in large transit networks and more frequently see only one side of a connection.

Limit injected response sizes Some middleboxes inject large block pages, directly enabling large amplification attacks. An alternative approach is for these middleboxes to only respond with a single `RST` to close a forbidden connection, or a with a minimal HTTP redirect to a different server that hosts a block page. If the middlebox's response size is smaller than the minimum size required to trigger it, this ensures that the middlebox will not be a productive amplifier.

Egress filtering Though middleboxes are only supposed to block websites for a limited group (such as a country or within a corporate or school network), many operate "bidirectionally", such that users outside the network accessing content within can also trigger injected responses. For instance, users outside China can still elicit the Great Firewall of China to inject `RST` packets despite not being the intended target of censorship. Instead, middleboxes should be configured to only censor requests originating from within the intended network, limiting the scope of

victims of amplification.

Remove or limit censorship devices Many middleboxes inject block pages into censored HTTP requests which use an outdated protocol that has been far surpassed in traffic volume and page loads by HTTPS [166]. The utility that HTTP-injecting devices provide is shrinking, and will ultimately disappear as more sites use TLS. However, the damage they inflict via amplification attacks will remain until these devices are removed. Disabling HTTP injection in these devices altogether would prevent abuse from attackers.

8.7.2 End Hosts

End hosts can take steps to mitigate the potential impact of these attacks. Hosts that drop outbound `RST` packets are more susceptible to TCP handshake-based attacks, but hosts that do not are susceptible to sustaining a packet storm from a victim-sustained amplifier. Instead, we recommend end hosts be configured to drop outbound `RST` packets probabilistically; this prevents an infinite packet storm, while still offering some protection from handshake-based amplifiers.

8.8 Conclusion

In this chapter, I presented the first non-trivial TCP-based reflected amplification attacks, and demonstrated that middleboxes could be automatically rendered ineffective at policy implementation to disastrous effect. To discover these attacks, I trained Geneva directly against censoring middleboxes with a new fitness function.

We then scanned the Internet dozens of times and find over 200 million IPv4 addresses that provide amplification from 1× to over 700,000×, as well as others that effectively yield *infinite* amplification.

Through a series of thorough follow-up experiments, we found that these TCP amplifiers are predominantly middleboxes, and frequently nation-state censorship devices. It has long been understood that nation-state censors restrict open communication for those in their borders; our work shows that they pose an even greater threat to the Internet as a whole, as attackers can weaponize their powerful infrastructures to attack anyone.

Our results show that middleboxes introduce an unexpected, as-yet untapped threat that attackers could leverage to launch powerful DoS attacks. Since the publication of this work [5], these attacks have since been found in the wild [167]. Protecting the Internet from these threats will require concerted effort from many middlebox manufacturers and operators. To assist in these efforts, we released our code publicly available at:

`https://geneva.cs.umd.edu/weaponizing`

In the next chapter, I will demonstrate another attack that renders middleboxes ineffective at correctly executing their policies by coercing them to disrupt innocuous communication.

Chapter 9: Weaponizing Censors for Availability Attacks

The previous chapter demonstrated that middleboxes can be rendered ineffective at policy implementation by implementing policy when they should not, to disastrous effect. The previous chapter focused only on HTTP, however, and did not affect censoring middleboxes that drop traffic to censor. This leads me to ask: can middleboxes that *drop* forbidden traffic to censor also be weaponized to launch attacks?

To answer this question, in this chapter I demonstrate a second attack that shows middleboxes can be coerced into executing their policy when they should not. There are additional benefits to answering this question, because this chapter also shows that censoring regimes pose a greater threat to the Internet than previously understood. In particular, we show that attackers can *weaponize* censoring infrastructure to keep two end-hosts separated by that country's borders from being able to communicate with one another, effectively blocking innocuous hosts. The attacker need not be within the censoring regime; it merely needs the ability to source-spoof packets.

The attack makes use of a little-studied but widespread feature of many censoring infrastructures: *residual censorship*. After a given TCP connection triggers a censor (e.g., by including a forbidden keyword in a plaintext HTTP GET request), some censors not only tear down the connection, but "residually censor" all *future* communication between the two endhosts (on particular ports) for some period of time—even if the subsequent traffic is completely innocuous.

Armed with this insight, our attack is relatively straightforward: the adversary spoofs the victim endhosts, sending packets with censored content across the censor's border, thereby triggering censorship and blocking the victims from communicating for some time.

Although conceptually simple, there are several challenging aspects of this attack in practice. In particular, most censoring middleboxes are stateful (they track connections across packets), and so it would seem that the attacker would have to fake a TCP three-way handshake in order to be able to send a valid censored packet in the first place. We show that, surprisingly, the attack is indeed possible, even with a completely off-path attacker.

The central contributions of this chapter are not just in demonstrating the possibility of weaponizing residual censorship, but also in performing two comprehensive feasibility studies for the attack:

First, we perform active measurements to analyze the current state of residual censorship around the world today: what countries employ it, how it operates, how long it lasts, and so on. Our results demonstrate a wide variety in the implementation of residual censorship systems—even within a given country, residual censorship

can operate very differently from one protocol to another.

Second, we analyze our attack's success and feasibility by launching it using (and targeting) hosts we control in three censoring nation-states—China, Iran, and Kazakhstan—across four protocols (HTTP, HTTPS+SNI, HTTPS+ESNI, and Iran's protocol filter [3]). This study sheds light on the limitations of the attack—for instance, we find that the attacker generally needs to be on the same side of the censor as the victim client. It also shows several surprising strengths of the attack. For example, Iran and Kazakhstan extend the duration of residual censorship whenever the censor sees a matching packet—as a result, once the attack is started, the victim's own packets help sustain the attack on themselves.

Our results show that even a low-resource attacker can weaponize censoring nation-states to launch an effective availability attack. In China, a source-spoofing attacker needs to send only four packets every three *minutes* to indefinitely sustain blocking between a given pair of end-hosts on a given destination port. An attacker that can sustain 1,093 packets per second (about 600 kbps) can weaponize Kazakhstan's censor, or 728 packets per second (422 kbps) to weaponize Iran's. Collectively, our results show that censorship infrastructures as they are deployed today have the potential to cause even more harm to the Internet at large than previously understood.

The rest of this chapter is organized as follows. In Section 9.1, we review related work and provide a background on nation-state censorship, residual censorship, and availability attacks. We describe our experiment methodology in Section 9.2. Section 9.3 presents our study of the current state of residual censorship, and Sec-

tion 9.4 presents our feasibility study from launching the attack against hosts under our control. We speculate about the breadth of the attack and discuss limitations in Section 9.5, explore potential mitigations in 9.6, and present ethical considerations in Section 9.7. Finally, I conclude this chapter in Section 9.8.

9.1 Background & Related Work

How censors operate There have been many measurement studies to understand how various censoring infrastructures work—far too many and varied to do full justice here. Instead, we highlight several key properties that are critical to understanding our results.

In-network censors generally have two broad components: a mechanism for determining whether to censor, and a set of mechanisms for actually tearing down the offensive connection. Determining whether to censor a connection has been shown to depend on keywords (e.g., in HTTP GET requests [40,168]), domain names (e.g., in the Server Name Indication (SNI) field during an HTTPS connection [2,7,36]), or the very protocol being used [3,12]. Our evaluation spans different types of these.

To actually tear down a connection, censors often employ one of two tactics: Some simply drop the offending user's (or connection's) traffic. This is referred to as *null routing*, and is obviously a very effective way of terminating a connection. However, it is also costly for the censor, as it requires them to have a box on the path between source and destination at which they can drop the traffic. More

commonly, censors are deployed not as man-in-the-middle adversaries, but as man-on-the-side: they sit just off of the path, and the ISPs send copies of packets (in both directions) to the censor for processing. For such deployments, the censor tears down the connection not by dropping the offending traffic, but by injecting spoofed TCP `RST`s (or lemon DNS responses [38]) to both client and server, causing them both to believe the other had terminated the connection. In our experiments, we study both null-routing and tear-down censors.

Residual censorship Residual censorship is a feature observed in some censorship systems in which the censor continues to block innocuous requests for a short period of time *after* censoring a forbidden request. We are not the first to observe this behavior; the Censored Planet datasets [54] report on instances where innocuous queries are blocked shortly after sending a censored query. It has also been noted in the context of studying censorship in China [7], Iran [3], and others [40] that, for some countries and some protocols, once a connection triggers censorship, subsequent connections can also be censored. However, to the best of our knowledge, we are the first to systematically study residual censorship—what precise protocols and ports it targets, for how long, and whether innocuous traffic can keep residual censorship in place—and how attackers can weaponize it.

An important facet of residual censorship is precisely what the censor blocks after censorship is initially triggered. There are three basic options available to an adversary: *2-tuple* (client IP, server IP), *3-tuple* (client IP, server IP+port), or *4-tuple* (client IP+port, server IP+port)[1]. We are not aware of any censors who use

[1] It is also conceivable that a censor could block *multiple* IP addresses at a time, such as a /24,

2-tuple residual censorship. All prior work of which we are aware that had identified some form of residual censorship focused only on 3-tuple. To our knowledge, we are the first to identify 4-tuple censorship, and yet, as we will show, it is one of the most widespread forms of residual censorship.

Weaponizing censors Besides the attack outlined in the previous chapter, I am aware of only one instance of coercing a censor into blocking someone else. In 2014, the developers of VPN Gate realized that the Great Firewall of China (GFW) had developed an active system for scraping the IP addresses of their VPNs and automatically blocking them without validating that these IP addresses were actually VPNs. The researchers began to mix innocent IP addresses into their published list of VPN servers and were able to control which IP addresses were globally blocked by the GFW for two days until the GFW added verification checks [169]. Our approach differs considerably; in our setting, an attacker can trigger the censorship, without needing the GFW to actively scan them. Moreover, our attack appears to be more difficult for the GFW to mitigate.

Off-path attacks This chapter fits into a much broader space of off-path attacks. Prior work has explored how to adversely affect TCP connections between two end-hosts in myriad ways, including TCP side channels [170] and data injection [171]. Other work has shown that an off-path attacker can weaponize network infrastructure to launch amplification attacks [147, 172, 173]. Each of these prior attacks manipulate the state at the end-hosts it targets. Our work broadens this space by showing that attackers can manipulate the state of *middleboxes in the network*

but we did not study this.

itself to adversely affect end-hosts' ability to communicate.

9.2 Measurement Methodology

As with all censorship measurement research, we are limited by the vantage points we can access and the censorship we can experience. For our experiments, we obtained four vantage points within censoring countries: two in China (Beijing), one in Iran (Tehran), and one in Kazakhstan (Qaraghandy). We also performed experiments from two vantage points we obtained in India (Bangalore) and one vantage point we obtained in Russia (Khabarovsk), but as we will see in the next section, we were unable to identify residual censorship in either location. We also obtained vantage points located in geographically disparate locations around the world that do not experience censorship: Australia (Sydney), India (Mumbai), Ireland (Dublin), Japan (Tokyo), United Arab Emirates (Dubai), and the United States (Iowa, Colorado, and Virginia). Figure 9.1 shows the locations of each of these vantage points, along with the censoring regimes in which we validated our attack.

To test for residual censorship, we issued queries that trigger censorship followed by queries that do not trigger censorship on their own and observed if the censor interferes. The specific queries we issued for each protocol are as follows (for ease of exposition, we will refer to HTTPS with SNI as simply "SNI", and HTTPS with ESNI as simply "ESNI"):

- **SMTP**: Sent an SMTP request with a forbidden email address (such as "xiazai@upup.info" in China [2]) in the `MAIL FROM:` field.

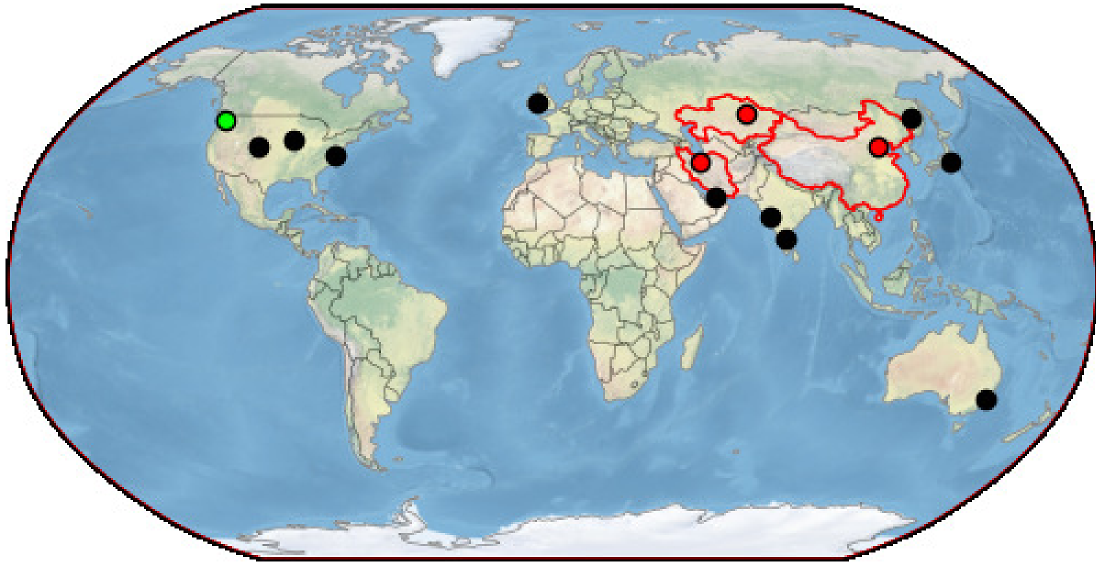


Figure 9.1: Vantage points in our experiments. The green dot is our attacker running SP^3 [9]; black dots represent victim vantage points; and the red dots denote the location of the servers inside the censoring regimes we studied: China, Iran, and Kazakhstan (outlined in red). Note that some dots overlap.

- **DNS**: Issued a DNS query (over both UDP and TCP) with a forbidden question record (such as “facebook.com” in China) both to real DNS resolvers and to resolvers we controlled.

- **HTTP**: Issued a HTTP GET request with a forbidden URL in the host header (such as `Host: youporn.com`), or with a forbidden keyword as an HTTP parameter (such as `?q=ultrasurf`).

- **HTTPS (SNI)**: Initiated a TLS handshake with a forbidden domain in the SNI field to servers we controlled.

- **HTTPS (ESNI)**: Initiated a TLS handshake configured with ESNI to servers we controlled.

- **Protocol Filter (Iran)**[2]: Sent two messages back to back containing the message "test". As this trivially does not match any approved protocol, it triggers censorship [3].

We also tested different patterns of follow-up requests and packets. To identify 3-tuple residual censorship, we issued follow-up queries with the same protocol to the same destination, containing an innocuous payload (such as "example.com"). We also tested making innocuous queries of different protocols and malformed payloads that do not resemble any protocol (such as just the string "test"). To identify 4-tuple residual censorship, we sent follow-up packets with the same source port to the same destination IP address and port (but with an out-of-window TCP sequence and acknowledgment number) and confirmed that our packets arrived at the destination correctly and without interference. We performed this check with `SYN` packets, `PSH` packets, `PSH+ACK` packets, and `RST` packets. We then repeated these experiments across many ports to identify which ports were affected.

9.3 State of Residual Censorship

In this section, we present the results from our comprehensive study of the current state of residual censorship in China, Iran, and Kazakhstan. Table 9.1 provides a breakdown of all of our results in this section.

Which countries employ residual censorship? We found some form of residual censorship (3-tuple or 4-tuple) for multiple protocols in China (SNI, ESNI, and

[2]In addition to its standard content filter, Iran uses a protocol filter, which censors unrecognized protocols on monitored ports [3].

Country	Protocol	Ports	Type	Duration	Bidirectional	Timer Reset	Mechanism
China	HTTP	Any	3-tuple	90s	✓	✘	Injected `RST`
	SNI	Any	3-tuple	60s	✓	Unknown	Injected `RST`
	ESNI	Any	3 and 4-tuple	120-180s	✓	✘	Null Routing
Kazakhstan	HTTP	Any	4-tuple	120s	✓	✓	Null Routing
	SNI	Any	4-tuple	120s	✓	✓	Null Routing
Iran	HTTP	53, 80, 443	4-tuple	180s	✓	✓	Null Routing
	SNI	53, 80, 443	4-tuple	180s*	✓	✓	Null Routing
	Protocol Filter	53, 80, 443	4-tuple	60s	✘	✓	Null Routing

Table 9.1: The current state of residual censorship, among the countries and protocols we tested (those that we tested but are not in the table did not residually censor in our tests). We were unable to reproduce SNI censorship in China; in that row, we report prior results [7]. *: Iran's SNI residual censorship sometimes lasts longer than 180s; in a small number of our experiments, we found it to last upwards of 5 minutes.

HTTP), Iran (HTTP, SNI, and its protocol filter), and Kazakhstan (HTTP and SNI).

China and Iran in particular employ residual censorship for only *some* of the protocols they censor. Neither have residual censorship for any of their DNS censorship (DNS-over-UDP or DNS-over-TCP)[3]. Further, China does not employ residual censorship for their SMTP censorship.

Some countries we tested do not employ residual censorship at all against our vantage points. Both of our vantage points within the Airtel ISP in India experienced HTTP and SNI censorship, but neither experienced residual censorship. We were also unable to trigger censorship from our vantage point in Russia to any of our destination vantage points, so we exclude both of these from our analysis.

What types of residual censorship do censors employ? We find that censors vary between 3-tuple and 4-tuple residual censorship, depending on the protocol being censored.

China uses 3-tuple residual censorship for HTTP traffic and censors by in-

[3]In Iran, although some prior work has reported DNS-over-TCP censorship [55], we are unable to trigger any DNS-over-TCP censorship at this time (similar to what was reported in [2]).

jecting TCP `RST` packets. This has been observed in the past [24, 40]. Prior work has reported residual censorship in China for SNI [7] by injecting `RST`s, but neither of our two vantage points experienced any SNI residual censorship to any of our vantage destinations.

ESNI censorship in China presents a more complicated picture. Less than 1 second *after* the GFW sees a TLS ClientHello containing the ESNI extension, it begins dropping all traffic that matches the connection's 4-tuple (note that the ESNI packet itself reaches the server unaffected). This is 4-tuple residual censorship. For approximately five seconds, the GFW also drops all traffic that matches the connection's 3-tuple: a short window of 3-tuple residual censorship. But if the client sends a second ESNI request with the same 3-tuple within the next three minutes, the GFW will begin dropping all traffic that matches the 3-tuple for three minutes: a long window of 3-tuple residual censorship. Unlike for HTTP and SNI, ESNI's residual censorship does not operate equally in both directions. Researchers have hypothesized in the past that China censors each protocol using a different set of middleboxes; the vast disparity between residual censorship implementation across our vantage points supports this hypothesis [2, 174].

In Iran and Kazakhstan, we find that the mechanism used for residual censorship (null-routing) and type of residual censorship (4-tuple) is consistent between protocols. As we will see later in this section, however, there are other inconsistencies in the implementations of the residual censorship for each censored protocol within Iran and Kazakhstan, such as the duration of censorship.

Does residual censorship use the same mechanisms as the initial censorship? We find that residual censorship is generally enforced using the same mechanism as the initial censorship. For example, China injects `RST` packets to censor HTTP normally, and injects `RST` packets for its residual censorship (the same is also reported for China's SNI censorship [7]). China's ESNI censorship operates with null-routing, as does its residual censorship. The censorship mechanisms are also consistent in Iran and Kazakhstan, with one exception.

We find that Iran censors HTTP using multiple methods simultaneously: injecting a block page with a packet that has the `RST` flag set while simultaneously null routing the connection. Despite using three censorship mechanisms for regular censorship, only 4-tuple null-routing continues for residual censorship.

What ports are affected by residual censorship? We tested this by issuing censored requests to vantage points we controlled destined to all 65,535 ports and confirmed that all were affected. We find that the ports affected by residual censorship match the ports affected by the regular censorship in each country we studied, but each country monitors a different set of ports. In China (with HTTP and ESNI) and Kazakhstan (with HTTP and SNI), we find that we can trigger residual censorship on any arbitrary port, including ephemeral ports. In Iran, however, both the protocol filter and the standard censorship system only monitor ports 53, 80, and 443, and therefore we can only trigger residual censorship to these ports. Note that in Iran, residual censorship can be triggered for any protocol on any of those three ports: for example, we can trigger HTTP residual censorship to port 53.

Is residual censorship applied bidirectionally? Even within the same country, residual censorship is not always applied equally to connections entering the country as to those exiting the country. Although we find that Iran's standard censorship system can be triggered bidirectionally, we confirm the findings of [3] that the protocol filter (and by extension, its residual censorship) only operates on flows leaving Iran. China's ESNI censorship operates bidirectionally, but it operates differently (and more aggressively) against traffic entering the country than exiting the country.

For every other censorship system we tested, we were able to trigger censorship (and residual censorship) equally from outside the country. Like all censorship research, our study is limited by the vantage points we can access; it is possible that there are other censorship systems that only employ residual censorship on connections leaving the country that we cannot study.

We find that the direction of subsequent traffic is important in whether it is affected by residual censorship. If a client within a censored regime makes a forbidden request to a server outside, we find that only traffic sent by the client is affected by residual censorship. This makes sense: traffic direction is encoded in both 3-tuple and 4-tuple flow tracking. However, this does impose an important limitation on attackers: an attacker generally must be on the same side of the censor as their victim.

What packets are affected by residual censorship? Which packets are impacted by residual censorship changes depending on the censorship mechanism

used. China's HTTP residual censorship mechanism of injecting `RST` packets does not initiate until *after* the client has sent a new request in a `PSH+ACK` packet. None of the 3-way handshake is impacted; it reaches the server without interference. However, China's ESNI residual censorship (both 3-tuple or 4-tuple) null-routes: all packets leaving the client, including `SYN` packets are affected by the residual censorship.

We find the same effect for the null-routing residual censorship in Kazakhstan and Iran. Note that the direction of traffic matters for every censor we studied: only packets from the client are impacted. If a server sends packets in a connection being null-routed, the packets will reach the client unaffected.

How long does residual censorship last? To determine the duration of residual censorship, we performed an experiment in which we varied the duration of time between triggering censorship and making a follow-up request, and recorded whether residual censorship took place.

We find the duration of residual censorship also varies between countries and protocols, but is generally less than three minutes in every country we studied. HTTP residual censorship in China lasts approximately 90 seconds (as observed in [24,40]) and ESNI is residually censored for 120 seconds (as observed in [36]). We note that for ESNI censorship in China, other researchers have reported both 120 and 180 seconds of residual censorship [36]. In Iran, while its protocol filter residually censors for 60 seconds, its HTTP and SNI censorship systems residually censor for 180 seconds (and in a small number of our experiments, the SNI system *continued*

to residually censor requests up to approximately 5 minutes). In Kazakhstan, both HTTP and SNI residual censorship systems operate for 120 seconds.

We find that both Iran and Kazakhstan *restarts* their residual censorship timer if the client sends a matching packet, thereby extending the duration of time that the client is affected. Due to TCP retransmissions, in practice this means that Iran and Kazakhstan will drop traffic for much longer than their original time. This is presumably done to make their censorship systems more robust against TCP retransmissions. As we will see in the next section, however, this timer reset makes our attack easier to launch.

Does residual censorship require a full 3-way handshake? No! We were able to trigger residual censorship without a proper 3-way handshake for every censor we studied. To discover this, we followed the methodology of Bock et al. [5] to attempt subsets of the TCP 3-way handshake before sending a `PSH+ACK` with a censored keyword.

The Airtel ISP in India enacted residual censorship without *any* of the 3-way handshake (one needs only send the `PSH+ACK`). Censorship of clients within this ISP appears to maintain no TCP state for their censored system.

Other countries required a subset, but not the entirety, of the TCP 3-way handshake. We sent a single `SYN` packet with a decremented sequence number, followed by a `PSH+ACK` containing the forbidden payload (we will refer to these two packets as the "censorship trigger"). This successfully triggered censorship (and residual censorship) for every censorship system we studied.

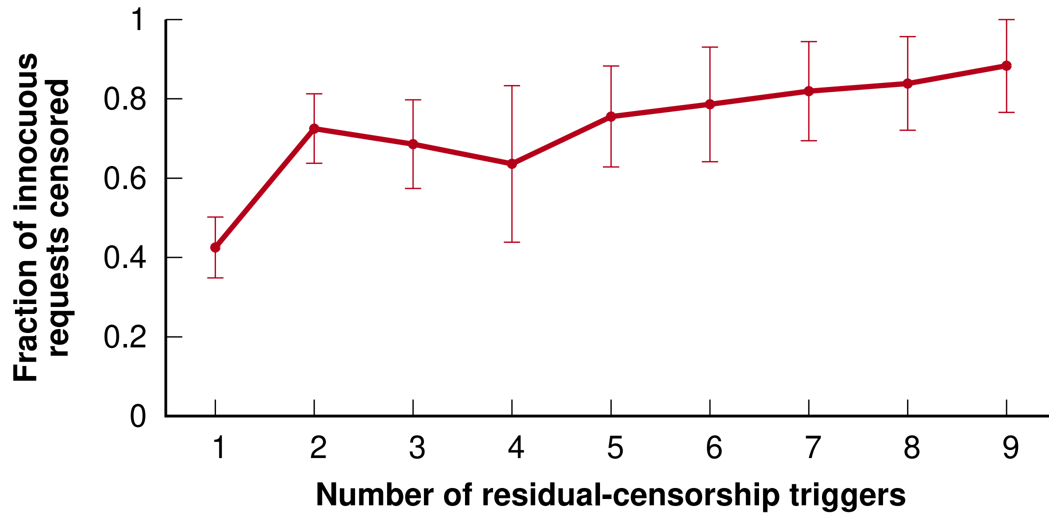


Figure 9.2: The relationship between the number of times censorship is triggered and the reliability of HTTP residual censorship, as measured from our Beijing 2 vantage point. As the number of times residual censorship is triggered increases, the reliability improves. (Error bars represent 95% confidence.)

How reliable is residual censorship? We define the "reliability" of residual censorship as the fraction of follow-up innocuous requests made within the residual censorship window that are successfully censored. Note that this is distinct from the reliability of censorship itself, which traditionally refers to the fraction of forbidden requests a censor successfully censors [2].

We performed an experiment to measure residual censorship reliability from each of our censored vantage points. We triggered censorship and then made one innocuous request per second and recorded how many requests were impacted; this experiment was repeated 10 times, spaced evenly throughout a 24 hour period. For every protocol in Iran and Kazakhstan and for ESNI censorship in China, we find that 100% of our requests were residually censored as expected. For HTTP residual censorship in China, however, we find that only approximately 50% of our requests are correctly residually censored. We find this pattern holds bidirectionally.

We next explored if we could improve the reliability of HTTP residual censorship. We performed an experiment in which we varied the number of forbidden requests we made before starting our test innocuous queries. From our Beijing 1 vantage point, we varied the number times we issued forbidden requests between 1 and 9 times, and then made one innocuous request per second for one minute. We randomized the order of the trials, implemented 5 minutes of sleep between each, and issued innocuous test queries before starting each experiment to ensure that the experiments did not interfere with each other. We repeated this experiment 6 times.

Figure 9.2 shows the average fraction of innocuous queries that were censored as a function of the number of residual-censorship triggers we send ahead of time. We find that as we increase the number of forbidden queries, we improve the reliability of residual censorship and after seven retries, the success rate levels out.

We hypothesize that the GFW is internally load balancing queries from this vantage point and that different middleboxes within the GFW do not communicate with one another when residual censorship starts. As we add additional queries, we are more likely to trigger residual censorship with multiple middleboxes, thereby increasing the likelihood that as future requests are made, they will get routed through a middlebox with active residual censorship.

Victim Location		Destination Location: Kazakhstan HTTP	Kazakhstan HTTPS	Iran HTTP	Iran HTTPS	Beijing 1 HTTP	Beijing 1 ESNI	Beijing 2 HTTP	Beijing 2 ESNI
Australia	**Sydney**	✓	✓	✓	✓	50%	10%	55%	✓
China	**Beijing 1**	✘	✓	✓	✓	N/A	N/A	N/A	N/A
	Beijing 2	✘	✓	✓	✓	N/A	N/A	N/A	N/A
India	**Mumbai**	✘	✓	✓	✓	✘	✘	✘	30%
	Bangalore 1	✓	✓	✓	✓	50%	10%	✓	✓
	Bangalore 2	✓	✓	✓	✓	25%	10%	✓	✓
Iran	**Tehran**	✓	✓	N/A	N/A	✘	50%	75%	✓
Ireland	**Dublin 1**	✘	✓	✓	✓	✘	✘	✘	5%
	Dublin 2	✘	✓	✓	✓	50%	✘	✘	✘
Japan	**Tokyo**	✓	✓	✓	✓	25%	✘	✘	✓
Kazakhstan	**Qaraghandy**	N/A	N/A	✓	✓	50%	✘	20%	✘
Russia	**Khabarovsk**	✓	✓	✓	✓	✓	✘	✓	✘
UAE	**Dubai 1**	✘	✓	✓	✓	85%	✘	95%	✘
	Dubai 2	✘	✓	✓	✓	✘	10%	✘	50%
USA	**Colorado**	✓	✓	✓	✓	✘	✘	✓	✘
	Iowa	✘	✓	✓	✓	✘	✘	✘	60%
	Virginia	✓	✓	✓	✓	50%	✓	55%	✘

Table 9.2: Success rates in weaponizing each country's censorship infrastructure against each victim vantage point from our attacker in Seattle, WA. (✓ denotes 100%, ✘ denotes 0%, and N/A denotes a location that does not cross the border of the censor.) Note that the success rates are not always consistent, even to victims in the same country, or between censored protocols in each censored regime. Iran is consistent and reliable; Kazakhstan is consistently unreliable for HTTP, but consistently reliable for HTTPS. In China, however, the attack was not always consistent by protocol, victim location, or server location.

9.4 Residual Censorship Attack

The results from our measurement of residual censorship indicate that it would be possible for an off-path attacker to get a victim's connections residually censored. Because censors do not look for the entire 3-way handshake, an attacker could simply source-spoof the victim, send a censored request, thereby residually censoring communication between the victim and server.

In this section, we empirically evaluate the feasibility of this attack by launching it against ourselves.

9.4.1 Launching the Attack

Since all of our vantage points employed egress filtering, we cannot launch the attack directly from our censored vantage points within China, Iran, or Kazakhstan.

Instead, we leverage a public deployment of SP^3 (A Simple Practical & Safe Packet Spoofing Protocol) [9] deployed at the University of Washington, to ethically send source-spoofed packets and thus act as our attacker. SP^3 is a web server that offers the ability to send spoofed packets, but mandates that a client *consent* to receiving source-spoofed packets. A client gives this consent by creating and holding open a websocket connection to SP^3. When the client connects, SP^3 returns a UUID16 challenge string. As long as the websocket connection is held open, other servers can connect to SP^3 with a websocket, supply the challenge code, and can give SP^3 packets through binary frames to send to that client.

We launched the attack on ourselves as follows. We used SP^3 to send a sequence of packets to trigger residual censorship to a server that crosses the censor, with the source addresses spoofed to be a test victim under our control. Recall that traffic direction matters to residual censorship in each of these three countries: the attacker must be on the same side of the censor as the victim. Since SP^3 is located in the United States, this means we are launching the attack from outside-in for each censoring country. Fortunately, as we saw in Section 9.3, residual censorship is bidirectional for most of the protocols we study. Our vantage points within each country acted as the server; we launched the attack against all of our geographically disparate vantage points around the world as victims.

Then, we used our "victim" to make requests to the server, and recorded if the connection succeeded or if it was impacted by residual censorship. We varied our test request based on the protocol and type of residual censorship. For 3-tuple residual censorship, the client makes an innocuous request with a different source

port to the same server IP address and port. For 4-tuple residual censorship, we ensure the client uses the same source port as the attacker. Of course, in a real attack scenario, the attacker cannot know the source port a victim will use *a priori*. Therefore, to weaponize 4-tuple residual censorship systems, the attacker would re-trigger censorship for all 65,535 possible source ports. We investigate the limitations imposed by this later in this section; for now to demonstrate the attack, we allow the attacker to access the source port.

We launched this attack against every uncensored vantage point for every bidirectional, residually censored protocol to each of our vantage points in China, with HTTP and ESNI, in Kazakhstan, with HTTP and SNI, and in Iran, with HTTP and SNI. Recall that Iran's protocol filter censorship cannot be triggered from outside the country, and therefore we omit it from these experiments. To determine attack reliability, we repeated each attack 20 times.

Before we launched each attack, we also record two traceroutes. First, we performed a regular traceroute between the victim and the destination. Second, we performed a source-spoofed traceroute using SP^3. Our server (inside the censored regime) connects to SP^3 and consents to receive TCP `SYN` packets with the `TTL` ranging from 1 to 30, with the source address of the packets spoofed to be the victim. While SP^3 sends these packets, the victim (a vantage outside of the censored country) records `TTL` "Time Exceeded" messages. This allows us to reconstruct the network path taken by the packets spoofed by SP^3, and compare it to the network path taken by the victim's test request.

9.4.2 Results

In every country we tested, we could successfully weaponize the censorship infrastructure against every victim vantage point at least once around the world. We find that the attack is sensitive to the chosen protocol (for example, HTTPS offers better results in Kazakhstan than HTTP). Table 9.2 presents an overview of our results.

Collectively, our results suggest that there are many shared paths through the censorship infrastructure of each country, and an attacker that can access just one source spoofed capable machine is capable of launching highly effective availability attacks. A more well resourced attacker could likely get even better results by choosing vantage points with even more similar paths as their victims.

In the remainder of this section, we detail the results in each of the countries we tested.

Kazakhstan In Kazakhstan, 100% of the attacks succeeded if the attacker triggered residual censorship with SNI payloads. However, we find that if a forbidden HTTP payload is used instead, the success varies depending on the victim vantage point, and this pattern persists irrespective of the port the attacker uses.

First, we explored why the success of the HTTP attack changes depending on the victim location. We hypothesize the reason for this is that the network path of the packets sent by the attacker and sent by the victim enter at different ingress points within the censor's infrastructure, and triggering censorship at one ingress does not initiate residual censorship at the other. To gain insight into this, we

can compare the two traceroutes taken before the attack is launched: one from the attacker and one from the victim. Although both traceroutes are performed with the same source IP address, since they start from different geographic locations, the packets will necessarily take (at least partially) different paths to reach the server. By comparing the paths taken for each traceroute, we can try to determine if the paths converged *before* the packets reached the censor, or afterwards. If the paths converge after the packets reach the censor, it is possible that the attacker's traffic and victim's traffic will take different ingress points, and therefore be processed by different censoring middleboxes. To determine how many hops away the censor is from the server inside the censoring regime, we send `TTL`-limited forbidden queries until we initiate censorship. We find that our vantage point inside Kazakhstan is 5 hops away from the censor. Necessarily, this analysis will not be perfect; many routers and middleboxes can simply choose not to send a `TTL` Time Exceeded message and hide themselves from this analysis.

Nevertheless, for all victims for which the attack failed, we find that paths do not converge until less than 5 hops away from reaching the server.

Why then, even for victims with paths that do not converge, does the attack succeed when HTTPS is used, even when the same destination ports are used as in HTTP? Frankly, we do not know. We hypothesize this could be due to Kazakhstan having physically fewer HTTPS censoring middleboxes, and therefore fewer internal paths for the attacker and victim's traffic to be split between.

What sending rate is required for an attacker to weaponize Kazakhstan's censor to block a 3-tuple (source IP address, destination IP address, destination port)?

Since both HTTP and SNI residual censorship can be triggered on any port, the attacker can choose to use whichever is more convenient. Both are 4-tuple residual censorship systems, which means the attacker must trigger censorship with the same source port that the victim will use. Since the attacker cannot know the victim's source ports ahead of time, instead the attacker will trigger censorship for all 65,535 possible source ports. It requires 2 packets to trigger censorship (a `SYN`, followed by a `PSH+ACK` with the forbidden payload), and once triggered, residual censorship will last for 120 seconds. Therefore, an attacker needs to send $2 \times \frac{65{,}535}{120} = 1{,}093$ packets per second to sustain the attack indefinitely. The `SYN` packet is 54 bytes long (including the Ethernet header), but the length of the `PSH+ACK` will change depending on the protocol. Our HTTP trigger payload is 91 bytes long (54 bytes of headers and 37 bytes for the HTTP request), and our HTTPS trigger payload is 379 bytes long (54 bytes of headers and 325 bytes of TLS ClientHello). To sustain the HTTP attack, an attacker must be able to send $(54 + 91) \times \frac{65{,}535}{120} = 79{,}188$ bytes per second, or 634 kbps. For HTTPS: $(54 + 379) \times \frac{65{,}535}{120} = 236{,}473$ bytes per second, or 1,892 kbps.

Recall that we found no difference in reliability between HTTP and SNI, and therefore an attacker could opt to use the smaller HTTP triggers and reduce the amount of required bandwidth unless their victim was located in a geographically disadvantageous location.

Would it be advantageous for an attacker to try to trigger residual censorship with both protocols? We cannot be sure, but an attacker likely does not need to. Since both censorship systems reset the duration of their residual censorship anytime

a matching packet is encountered, once the attacker triggers one censorship system, any packets sent to trigger the other will reset the timer for the first. We also note that the effects of censorship for HTTP and SNI are identical: for this reason, we cannot be certain whether packets being residually censored by one censorship system reach the other.

China The attack was inconsistent to both of our vantage points in China. The success rate of the attack varied based on multiple factors: the victim location, server location, and the chosen residually censored protocol.

As in Kazakhstan, we consulted the traceroutes to examine if the network paths could explain the lack of success for the attack. We repeatedly sent `TTL`-limited forbidden requests to determine how many hops both of our machines are away from the GFW (6 hops and 9 hops respectively). We hypothesized that the attack should succeed greater than 0% of the time if the paths converge before it reaches the censor. Recall from Section 9.3 that in China, triggering HTTP residual censorship once does not guarantee that all future requests that match the 3-tuple will be censored; therefore, even if the attacker's and victim's paths converge before packets reach the GFW, we cannot guarantee success. Nevertheless, the traceroutes do not contradict our hypothesis: we find almost no path convergence for every victim against which the attack frequently failed (such as Ireland 1& 2).

Why are these success rates not either 100% or 0%, as in Iran and Kazakhstan? Bock et al. observed a similar phenomenon in [2] and posited that the GFW is a heterogeneous deployment of many different middleboxes, all running in parallel.

We hypothesize that fractional success rates are caused by geographic variation in deployments of the GFW itself, and load balancing between multiple middleboxes running in parallel.

For an attacker, weaponizing the GFW poses an interesting opportunity, as it offers both types of residual censorship (3-tuple or 4-tuple) and multiple different censorship mechanisms (null routing or injected `RST`s). Attackers within the country can choose to trigger ESNI residual censorship at either the 3-tuple or 4-tuple with null routing, or trigger 3-tuple HTTP residual censorship to get injected `RST`s. Outside the country, ESNI censorship is limited to 4-tuple residual censorship, so the attacker can choose whether to launch one or the other depending on the location of their victim.

With 3-tuple censorship systems at an attackers disposal, weaponizing the GFW to prevent a victim from communicating with a given destination IP address and port is trivial. An attacker needs to trigger censorship only once to initiate the residual censorship, and can trivially re-send the censorship triggers to improve the reliability if needed. If 3-tuple residual censorship is unavailable, the attacker can fall back to leveraging 4-tuple residual censorship, as we demonstrated in Iran and Kazakhstan, which also lasts for 120 seconds. To trigger ESNI's 4-tuple residual censorship, the attacker must send a `SYN` (54 bytes), followed by the `PSH+ACK` containing the ESNI trigger (54 bytes for headers and 65 bytes of payload). An attacker needs to send $2 \times \frac{65{,}535}{120} = 1{,}093$ packets per second, equivalent to $(54 + 119) \times \frac{65{,}535}{120} = 94{,}480$ bytes per second, or 756 kbps to sustain the attack indefinitely.

Could an attacker simply try to invoke both censorship systems simultaneously

in an attempt to improve the reliability of this attack? We find the answer is yes: the attacker can send multiple back-to-back packet sequences to trigger censorship using different protocols, as long as each source port is different. For example, the attacker can trigger 3-tuple HTTP residual censorship, followed by a trigger for 4-tuple ESNI censorship with a different source port. We find that if both triggers are sent with the same source port, only the first trigger will be successful. The reason for this was posited by [2]: once the HTTP censorship system sees the ESNI payload, it stops paying attention to the connection. However, since the HTTP residual censorship is 3-tuple, the attacker can use one source port to trigger the HTTP residual censorship system and still trigger 4-tuple residual censorship on all of the other source ports.

With both censorship systems performing residual censorship in parallel, which one affects a victim? We find the answer is the ESNI censorship system: this is because the ESNI residual censorship affects all packets, but the HTTP residual censorship system does not teardown a connection until after the 3-way handshake has completed. In our testing, we did not see an improvement in reliability when combining censorship triggers, but its utility may increase for victims in other geographic locations.

Iran Our attack was most successful in Iran. Here, 100% of the attacks succeeded using both forbidden HTTP and HTTPS (SNI) against every victim we tested. Both of these protocols are 4-tuple censored for a full 180 seconds, and both timers reset in the presence of any matching packet.

What is required for an attacker to effectively block a victim from communicating with a destination IP address and port across the censor? The attacker requires 2 packets to trigger censorship (a `SYN`, followed by a `PSH+ACK` with the forbidden payload), and once triggered, residual censorship will last for 180 seconds. Therefore, an attacker needs to send $2 \times \frac{65{,}535}{180} = 729$ packets per second to sustain the attack indefinitely. The triggers are the same for Iran as for Kazakhstan: the `SYN` packet is 54 bytes long (including the Ethernet header), our HTTP trigger payload is 91 bytes long (54 bytes of headers and 37 bytes for the HTTP request), and our HTTPS trigger payload is 379 bytes long (54 bytes of headers and 325 bytes of TLS ClientHello). To sustain the HTTP attack, an attacker must be able to send $(54 + 91) \times \frac{65{,}535}{180} = 52{,}792$ bytes per second, or 422 kbps. For HTTPS: $(54 + 325) \times \frac{65{,}535}{180} = 137{,}987$ bytes per second, or 1.1 Mbps—a modest amount.

The length of the payload required to trigger SNI censorship is significantly larger than the payload required to trigger HTTP censorship, and since each protocol worked equally well for our attacker, there is no incentive to use the longer SNI trigger. Of course, like in Kazakhstan, if the HTTP trigger fails for a given victim location, A bandwidth constrained attacker could opt to start with HTTP triggers and only switch to SNI triggers if their victim is in a disadvantageous geographic area.

9.5 Attack Impact

Here, we reason about the potential impact of this attack by considering the potential breadth and limitations.

Breadth What is the true breadth of this attack? Unfortunately, we are limited by our vantage points to answer this definitively. Nevertheless, we can speculate about what other systems could potentially be weaponized.

We restricted our analysis only to censoring countries in which we could obtain vantage points that experienced residual censorship. Although we were unable to test this attack in India or Russia, prior work has found that other ISPs in India (Vodafone and Idea [28]) and Russia [175] employ null routing for censorship. Depending on how the null routing is implemented, these ISPs may be vulnerable to this attack, but we were unable to obtain vantage points within these systems to confirm this.

Our analysis assumed that either the server or victim is located physically inside a censoring regime. However, researchers in the past have observed that traffic that simply traverses the Internet borders of a censored regime can trigger censorship, even if neither the client nor server are located within the country [38]. Performing this attack against traversing traffic is an interesting area of future work.

We can also speculate about the breadth of this attack by examining the results of Quack, a powerful censorship scanning tool from Censored Planet [52]. Every day, Quack sends well-formed HTTP GET requests with potentially forbidden domains in the `Host:` header to echo servers around the world to identify interference. Quack

records the cause of censorship and also monitors for 3-tuple residual censorship (called "stateful disruption"). In the December 27th, 2020 dataset, Quack had identified censoring middleboxes in 33 countries where 3-tuple stateful disruption was present and in 18 countries where null routing was used to censor. These results suggest that this attack may be significantly more broadly applicable.

Limitations Despite the potential breadth, there are limitations to this attack. An attacker must be able to obtain a vantage point (1) without egress filtering that (2) shares a similar enough path with their victim and (3) the traffic crosses a censor (4) with residual censorship (5) that can be triggered statelessly.

Our experiments suggest that there are a surprisingly high number of joint network paths, even for geographically disparate victims (such as Australia and USA). Still, not every attacking vantage point will be able to affect every victim, and the attacker has no mechanism to confirm whether their attack successfully blocked the victim.

Another potential limitation is that this attack may not work for every IP address. Researchers have observed in the past that some censorship systems vary their response based on the destination [3]. We were unaffected by this for all of our victim locations, but an interesting area of future work would be to repeat this study across a very broad range of IP addresses.

Lastly, there are some limitations to how completely an attacker could cut off two hosts. Could an attacker weaponize these censorship systems to *completely* cut two hosts from communicating? It depends on the type of residual censorship.

We believe it is infeasible for an attacker to use a 4-tuple censorship system to completely prevent two IP addresses from communicating, as this would require triggering censorship for all 2^{32} possible combinations of source and destination ports. However, for a 3-tuple residual censorship system, the attacker could trigger residual censorship 65,535 times to all possible destination ports and accomplish this.

Does this attack become infeasible if middleboxes start properly tracking the 3-way handshake? Yes, but we believe it would be difficult for censors to do so. Particularly at the scale at which nation-state censors must operate, censors must content with path asymmetry: the network path used by traffic exiting the country may be different than the path used by traffic entering the country, even for the same connection. This makes properly tracking the 3-way handshake difficult: different middleboxes may see the `SYN` packet from the client than those that see the `SYN+ACK` packet from the server.

Can the attacker trigger residual censorship for UDP-based protocols as well? In our experiments, we only identified residual censorship for TCP-based protocols. However, this is only a partial limitation, since all of the null-routing residual censorship we studied affected both TCP and UDP traffic. If an attacker wishes to interfere with UDP traffic, she can simply trigger null-routing residual censorship over TCP and the victim's UDP traffic will be censored.

9.6 Mitigations

In this section, we discuss our recommendations to potential victims and censoring regimes to mitigate this attack.

9.6.1 Censors

Null-routing should track sequence numbers, or should not be used. All of the null-routing censorship systems we study (Iran, Kazakhstan, and China's ESNI censorship) operate only at the 4-tuple, and do not do any validation of the sequence or acknowledgment numbers of the packets they drop. Unfortunately, this implementation of censorship with null-routing is inherently flawed. TCP is designed to be tolerant to packet loss, so most end-hosts will continue to retry sending packets when confronted with null-routing. This forces censors to maintain the flow's null-routing for a long enough period of time to exceed the duration of time that network stacks will retransmit (or further reset their internal timer when an offending packet is sent). Unfortunately, the longer this window of time is, the easier it is for an attacker to abuse null-routing to perform this attack. Therefore, to eliminate 4-tuple residual censorship, we recommend that middleboxes who use null-routing only drop packets with the correct sequence and acknowledgment numbers, or to avoid using null-routing entirely.

Eliminate (or modify) 3-tuple residual censorship. Presumably, 3-tuple residual censorship is designed as a deterrent system: users who search for a forbid-

den term are "punished" and forbidden from trying to communicate with the same server again for a small period of time. Unlike 4-tuple residual censorship, the effect of 3-tuple residual censorship is salient to the user. However, we question the efficacy of this feature as a deterrent, since there is no communication or information to the end-user to alert them *why* they are continually being censored in all countries we tested in (China, Iran, Kazakhstan). Consider a user in China that searches for a long string of text containing a single verboten word. The GFW only sends `RST` packets: it does not inform the user the cause of censorship, and an uneducated user may be unaware that censorship is the reason their subsequent connections continue to fail. Worse, as we showed in Section 9.3, residual censorship is not even always be effective, and can fail depending on the users network route. For these reasons, we recommend that middleboxes—particularly the GFW—remove their residual censorship components altogether or modify their response from null routing to sending a block page or some response that indicates to the user who is being censored that they are being "punished" for their search.

We also echo many of the suggestions made by Bock et al. [5], as the root of our attack also stems from the ability to trigger censorship systems without a proper 3-way handshake.

9.6.2 Potential Victims

Unfortunately, once the attack is initiated, there is very little a victim can do to stop it. Nevertheless, we make recommendations here to mitigate or work around

this attack.

Use a proxy. Since our availability attack is generally limited by the 3-tuple or 4-tuple, changing the source IP address that the censor sees is an effective way to bypass the attack. Therefore, we recommend that an affected user switch to use some proxying system, such as VPN, Tor, or an HTTP proxy. Further, a victim can rapidly rotate between proxies in an effort to stay ahead of an attacker. Unfortunately, this is only a stopgap solution; if the path from the victim to the proxy's entry nodes also crosses the censor, an attacker can simply switch to attacking the proxy itself.

Do not immediately try to reconnect. In some censorship systems, the presence of additional matching traffic causes the residual censorship timer to reset, thereby prolonging the attack. Therefore, if a user is affected, they should not continue trying to reconnect; instead, they should stop sending network traffic and wait a few minutes.

9.7 Ethical Considerations

Experiment Design We took care in designing our experiment to ensure that it would not involve or cause harm to any other users. Our experiments do not induce any in-country clients outside of our control to send forbidden requests; all communication was strictly between hosts we fully controlled. To the best of our knowledge, none of our vantage points in-country were NATted with other hosts, making it unlikely other users were affected.

Responsible Disclosure It is difficult to responsibly disclose our findings, as the affected censorship systems have historically been unresponsive to similar issues [5] or unwilling to intentionally weaken their censorship systems. Nevertheless, we are in the process of contacting several country-level Computer Emergency Readiness Teams (CERT) that coordinate disclosure for their respective countries.

9.8 Conclusion

In this chapter, I demonstrated that it is possible to weaponize the censorship infrastructure in Iran, Kazakhstan, and China to perform availability attacks. We launched this attack against 17 different geographically disparate victims under our control and show that even a weak attacker (with access to a single low-bandwidth source spoofer) can launch effective availability attacks.

Collectively, Chapters 8 and 9 show that middleboxes can be rendered ineffective at executing their network policy by coercing them to censor content they should not. These results show that the negative impact of censorship extends well beyond the censor's borders, and that they pose an even larger threat to the Internet writ large. Taken together, Chapters 3-9 constructively prove my thesis, showing multiple ways that censoring middleboxes' policies can be rendered ineffective in automated manners.

In the next chapter, I will take a step back and discuss what it would take for a censored regime to defend itself against the myriad attacks I presented in this dissertation and reason about the limits of my automated approach.

Chapter 10: Defending Against Geneva

10.1 What would it take to defend against Geneva?

What would it take to defend against Geneva's strategies? In this dissertation, I have presented a total of 141 evasion strategies that evade censorship in 4 countries (China, India, Iran, and Kazakhstan) across 16 unique, real-world censorship systems (China: HTTP, HTTPS SNI Primary, HTTPS SNI Secondary, HTTPS ESNI, SMTP, FTP, DNS; India: HTTP, HTTPS; Iran: HTTP, HTTPS, DNS-over-TCP, Protocol Fitler; Kazakhstan: HTTP, HTTPS, HTTPS MITM). To defend against all of these strategies, the minimal characteristics that a middlebox must have are: it must possess no bugs, fully process every packet in a connection, and always maintain consistent state with the end hosts. Intuitively, if all of these conditions are met, then the middlebox will correctly process exactly the same set of packets as the end server, or the packets will not be delivered. In this section, I will show that each of these are necessary, and that if any one does not hold, there may be a potential for attack. Much of this section will focus on TCP-based protocols, as they require more from the censor, but I will also argue these characteristics are still necessary to censor DNS over UDP.

Fix Bugs Most trivially, packet manipulators can make use of bugs to evade policies, so a first step is for middlebox manufacturers to fix all their bugs. Many Geneva strategies, particularly the server-side strategies, are examples of this. Bugs represent 28/141 of Geneva's strategies: Turnaround (1), Invalid Options(1), Four Element Request Line (3), Host Header Shield (6), Host Header Whitespace (14), Path Confusion (2), and Double FIN (1).

If there are exploitable bugs available in the middlebox, they may be leveraged to render the middlebox ineffective.

Fully Process All Packets There are multiple reasons for which a middlebox would not fully process every packet within a connection. Some middleboxes stop paying attention to a connection after a certain threshold number of packets have been exchanged, such as Iran's Protocol Filter, which only tracked the first 9 packets in a connection [3]. Some middleboxes watch only until specific packets have been sent [3,4], such as China's backup SNI censorship system, that stops watching after certain TLS messages have been sent by the client. I have reported on cases in which middleboxes stop processing packets after the connection appears to have been terminated. This problem also arises in the application-layer space: some middleboxes have a fixed amount of buffer space they store requests in, and if a request is too long, the middlebox can miss the forbidden request. Other middleboxes miss traffic due to asymmetric routes, load balancing, and more [5].

This broad category encompasses the majority of the strategies reported in this dissertation, as in particular, most of the application-layer strategies trick the

middlebox into not processing or identifying the forbidden keyword. In total, fully processing all packets would eliminate 74/141 strategies in the species.

If a middlebox does not monitor all traffic and fully process each packet in a given connection, a packet manipulator may be able to inject a packet that causes the middlebox to ignore the rest of the connection, become desynchronized from the connection, or miss the forbidden query entirely.

Mandate Consistent State Many packet manipulation attacks exploit the eavesdropper's dilemma, which states that it is difficult for a middlebox to maintain consistent state with the end-hosts of the connection. For example, injecting a payload that the middlebox processes with a limited `TTL` will cause the censor's state to update without reaching the server, making the middlebox desynchronized. I foresee two possible approaches that enable a middlebox to mandate consistent state, despite the eavesdropper's dilemma.

First, a middlebox could operate *in-path* and *fail-closed*. The idea of a fail-closed system is straightforward: if the middlebox encounters any packet or request that it cannot parse, does not match its internal state, or contains ambiguity in its interpretation, then that packet should not be delivered. In order for a fail-closed system to be effective, however, it must operate in-path and drop offending traffic: if the middlebox requires per-flow state to disrupt a connection and its internal state is incorrect, it will not be able to correctly disrupt the connection. Operating fail-closed and in-path defends against eavesdropper's dilemma-based attacks by simply mandating that only traffic that matches its internal state will be allowed through.

Under this model, an attacker is welcome to try to desynchronize the middlebox from the connection, but in so doing, the attacker will cause the middlebox to drop the real connection when it does not match any internal state.

Second, a middlebox could *normalize* the traffic. A defensive traffic normalizer was first proposed by Vern Paxson et al. in 2001 [84] to defend against packet manipulation attacks and contend with the eavesdropper's dilemma. The normalizer's goal is to ensure that the state of the middlebox is always consistent with the state at the end-host. To achieve this, the normalizer modifies network traffic as it goes by: it overwrites `TTL` values to ensure packets reach the end-host, drops packets with incorrect checksums or that will be ignored by the server, etc.

Neither of these approaches can be perfect, however. A key limitation to traffic normalizing middleboxes is that they cannot know a priori the semantics for a given connection [84]. A canonical example of this is with the TCP Urgent pointer: if a client sends the message `robot` with the urgent pointer pointed to `b`, depending on the server's connection setup, the server may process either `root` or `robot`. If there are other semantics imposed by the application-layer on the underlying connection as to what bytes should be accepted or not, it is possible that a packet manipulator could sneak data or a request past the middlebox, even with consistent state. Inconsistent state issues were responsible for 39/141 of Geneva's strategies.

If the middlebox does not store consistent state with the end-hosts, it may be vulnerable to desynchronization attacks.

DNS Censorship Much of this section has focused on TCP-based protocols, and

the limitations inherit to reliably censoring these protocols. However, one of the most important protocols for censors, DNS, runs over UDP. For middleboxes, DNS-over-UDP requires less complexity to censor compared to any TCP-based protocols, as the middlebox does not need to track state, reassemble data streams, and more.

The above requirements still hold for DNS censorship. If there are exploitable bugs present, Geneva may be able to discover a packet modification to evade the censor. If the packets are not processed completely, Geneva may be able to pad the packet with innocuous data until the forbidden query is ignored. Finally, if the middlebox does not mandate that only packets that are completely and correctly processed should be delivered, Geneva may be able to send a request that is not correctly parsed by the censor due to RFC ambiguities and subvert censorship. I presented an example of all three of these scenarios in Chapter 5.

10.2 Does Geneva help the censor?

I report on many circumvention strategies (including those that are likely bugs in censor implementations) in this dissertation, and discuss what would be required for a censor to mitigate 100% of the issues in this chapter. Are these requirements a recipe for censors to follow in the future? Although they would defend against all the packet manipulation attacks discovered by Geneva and discussed in this dissertation, actually implementing these changes would likely be exceedingly challenging at scale.

For example, mandating consistent state in the presence of asymmetric routes and load balancing may be very difficult. As an example, the GFW is currently a

fail-open system, and we hypothesized this is the case because they operate many independent middleboxes in parallel [2]. In this deployment context, every middlebox must be fail-open, because each middlebox must assume that some other middlebox may be able to handle any traffic it cannot. In these circumstances, imposing the requirements stated in this chapter could require a significant re-architecture and re-implementation of their entire censorship system.

There may also be a high cost to imposing these requirements. For example, mandating that traffic must be correctly parsed and understood to be delivered may cause a high degree of collateral damage, as there are a wide variety of server implementations running in the wild. Storing more state about every connection than the most stateful end-server may impose a high memory cost.

Therefore, even though a censor may use Geneva to identify bugs and limitations, actually fixing those limitations may be challenging in practice.

Lastly, although Geneva is one mechanism that a censor can use to identify their own bugs, middlebox manufacturers have access to their own code. Existing tools have demonstrated that fuzzing can be done significantly faster with code instrumentation [74], so censors could have been fuzzing their own systems to find these issues from their initial development. By releasing Geneva open source, we are *democratizing* the ability to find bugs and limitations in their censorship systems.

Chapter 11: Conclusion and Future Work

In this thesis, I demonstrated that it is possible to automate the discovery of ways to render middleboxes ineffective at implementing their network policies. I developed Geneva, a novel genetic algorithm that can learn packet sequence modifications against a live adversary, and I showed that it could be used to discovery both new ways to evade censorship (across multiple network protocols and deployment contexts) and to launch dangerous network attacks. In this chapter, I will speak to future work in this space.

11.1 Immediate Term Challenges

Before speaking to longer term future work, I will note several challenges for the immediate term.

TLS Support Although Geneva has support for HTTP and DNS, extending it to support TLS has the potential for great impact, as the web is increasingly moving to HTTPS. There are several challenges in adding TLS support. First, the TLS state machine is significantly more complicated than any of the other protocols that Geneva supports, dramatically expanding the search space. Techniques to reduce

the search space, such as testing a strategy against a local server before testing it against a live adversary, will likely be required to make the problem tractable. Second, there are many implementations and versions of TLS in active use. In order to effectively walk through the entire search space, Geneva must be able to handle each TLS version and extension, even if those implementations are conflicting. For example, there have been multiple implementations of TLS 1.3's Encrypted Client Hello (ECH) as the standard evolved. Still, with the widespread use of HTTPS, TLS support would be an impactful direction to explore, and could lend us insights into how middleboxes themselves have handled TLS's evolution over time.

Training without client instrumentation Server-side evasion strategies are easier to deploy than client-side evasion techniques. Unfortunately, the process of discovering server-side evasion strategies with Geneva has historically required instrumentation from a client: to make requests with specific parameters to evaluate each strategy during the evolution process. As a consequence, training is limited only to those countries within which we can safely procure a vantage point that can be remotely instrumented.

In the future, it would be impactful if it were possible to train Geneva without requiring control of the client. Designing such a mechanism has its challenges, however. First, if Geneva cannot control or instrument the client, the first major challenge is how to direct traffic that will trigger the censor to its active strategies. This may require cultivating a dedicated user base of testers, a standalone program that can generate connections, or by partnering with an existing forbidden server.

Second, Geneva benefits from the ability to collect additional information from its clients while evaluating strategies, such as how the strategy impacted the underlying connection, in order to inform the fitness function. It is an engineering challenge to recover this information from the server-side of the connection. Finally, if the learning algorithm must depend on clients over which it has no control, there may risk of a sybil attack from the adversary trying to pollute the algorithm's training set.

Measuring Middlebox-based Amplification Attacks Already, the middlebox-based TCP reflected amplification attacks have been detected in the wild [167], but there is no measurement yet of how these attacks have progressed, who they are attacking, and who is launching the attacks. In the future, developing a system to globally monitor for attackers trying to use this attack vector could help us learn more about how quickly attackers can incorporate and optimize the attack, and better protect those under attack. I foresee two principle ways that we can detect attackers using this threat vector: during the attacker's discovery phase or during the attack phase. During the discovery phase, the attacker must find and discover potential amplifiers on the Internet, which requires Internet scanning. In the future, we can develop tools to detect these Internet-wide scans to determine who is scanning to identify potential amplifiers. Such a system could even respond to these scans with a modest (but bandwidth constrained) amplification amount, so that the system also gets included in the attacker's attack phase. In the attack phase, we can develop tools to detect the fingerprints of middlebox responses and

partner with organizations that have a wide network view to detect and measure live attacks.

Understanding the True Limits of Automating Evasion The eavesdropper's dilemma suggests some fundamental limitations for middleboxes, and helps to inform the limits of this approach [50]. However, it is unknown the true limits of this approach: is it the case that in order for any middlebox system to render binary censorship decisions at line-speed will necessarily incur one of the weaknesses described in Chapter 10? Defining a formalization for middlebox network functions might allow us to formally reason about whether every type of network middlebox will be vulnerable to packet manipulation attacks (and if so, if those attacks can be automatically discovered deterministically).

11.2 Long Term Challenges

Contending with Adversarial Systems Today, Geneva's adversaries are relatively static while it is training. Censors may make changes or deploy new systems over time, but in the timespan of the hours that Geneva is training, to the best of my knowledge, the functionality of censors is static. This means that the censor does not adapt to what Geneva is doing in real time. In the future however, middleboxes and censors may take a more adversarial role during the training process, and directly try to interfere with Geneva's training.

For example, if a censor were to identify hosts training Geneva, they could apply different network policies, or change their network policies dynamically to

pollute Geneva's training data. Alternatively, a censor could simply cut Geneva off from the network entirely. None of the three existing automated tools for discovering evasion strategies (SymTCP, *Alembic*, and Geneva) are designed to handle a dynamic adversary that changes at runtime.

Designing a new algorithm that is equipped to handle a dynamic adversary is challenging. Ideally, a learning algorithm hardened to work against an active adversary would need to escape identification while running, blending into normal network traffic.

Preparing for the Next Censorship Arms Race Various activists I work with have warned of several troubling future censorship capabilities against which the anti-censorship community is not prepared. Sophisticated techniques like throttling instead of outright blocking, and using machine learning to fingerprint anti-censorship protocols require us to reconsider how we evade censorship. Moreover, some evidence points to countries like China *personalizing* what content gets censored based on a user's occupation or social credit score. This will require a complete redesign of how we approach censorship measurement: no longer will it suffice to say that a site is blocked, we will have to understand *for whom* a site is blocked. Advances like these point to an even greater need for automated techniques to measure and circumvent censorship. Personalized censorship may require personalized evasion, but one of the challenges I foresee is that training could put users at risk against an aggressive adversary. Developing new ways surreptitiously and collaboratively train in a federated manner could enable users safely learn from one another.

To support future researchers in taking on these challenging problems, I have made my dissertation's various artifacts publicly available at:

`https://geneva.cs.umd.edu`.

Bibliography

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

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

[3] Kevin Bock, Yair Fax, Kyle Reese, Jasraj Singh, and Dave 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.

[4] Kevin Bock, Gabriel Naval, Kyle Reese, and Dave Levin. Even Censors Have a Backup: Examining China's Double HTTPS Censorship System. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2021.

[5] Kevin Bock, Abdulrahman Alaraj, Yair Fax, Kyle Hurley, Eric Wustrow, and Dave Levin. Weaponizing Middleboxes for TCP Reflected Amplification. In *USENIX Annual Technical Conference*, 2021.

[6] Kevin Bock, Pranav Bharadwaj, Jasraj Singh, and Dave Levin. Your Censor is My Censor: Weaponizing Censorship Infrastructure for Availability Attacks. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2021.

[7] Zimo Chai, Amirhossein Ghafari, and Amir Houmansadr. On the Importance of Encrypted-SNI (ESNI) to Censorship Circumvention. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2019.

[8] P. Mockapetris. RFC 1035, 1987. `https://datatracker.ietf.org/doc/html/rfc1035`.

[9] Will Scott. A Secure, Practical & Safe Packet Spoofing Service. 2017.

[10] Reporters Without Borders. Enemies of the Internet 2013, Report. http://surveillance.rsf.org/en/wp-content/uploads/sites/2/2013/03/enemies-of-the-internet_2013.pdf, March 2013.

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

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

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

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

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

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

[17] Ana Bita Samba Vasilis Ververis, Fadelkon. Women on Web website censored in Spain. https://blog.magma.lavafeld.org/post/women-on-web-blocking/.

[18] Kai Wang and Wanyuan Song. Peng Shuai: How China censored a tennis star. https://www.bbc.com/news/59338205.

[19] Dave Levin, Youndo Lee, Luke Valenta, Zhihao Li, Victoria Lai, Cristian Lumezanu, Neil Spring, and Bobby Bhattacharjee. Alibi Routing. In *ACM SIGCOMM*, 2015.

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

[21] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: Anticensorship in the Network Infrastructure. In *USENIX Security Symposium*, 2011.

[22] Josh Karlin, Daniel Ellard, Alden W. Jackson, Christine E. Jones, Greg Lauer, David P. Mankins, and W. Timothy Strayer. Decoy Routing: Toward Unblockable Internet Communication. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2011.

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

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

[25] Zhihao Li, Stephen Herwig, and Dave Levin. DeTor: Provably Avoiding Geographic Regions in Tor. In *USENIX Security Symposium*, 2017.

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

[27] Max Schuchard, John Geddes, Christopher Thompson, and Nicholas Hopper. Routing Around Decoys. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

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

[29] Daniel Anderson. Splinternet Behind the Great Firewall of China. *Queue*, 10(11), November 2006.

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

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

[32] John Geddes, Max Schuchard, and Nicholas Hopper. Cover Your ACKs: Pitfalls of Covert Channel Censorship Circumvention. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[33] Anonymous, Arian Akhavan Niaki, Nguyen Phong Hoang, Phillipa Gill, and Amir Houmansadr. Triplet Censors: Demystifying Great Firewall's DNS Censorship Behavior. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2020.

[34] Moxie Marlinspike. Doodles, stickers, and censorship circumvention for Signal Android. `https://signal.org/blog/doodles-stickers-censorship/`, 2017.

[35] Signal. Egypt keeps trying to block Signal, inadvertently blocking all of Google, and having to stop as a result. We'll also expand domain fronts. `https://twitter.com/signalapp/status/817062093094604800`, 2017.

[36] Kevin Bock, iyouport, Anonymous, Louis-Henri Merino, David Fifield, Amir Houmansadr, and Dave Levin. Exposing and Circumventing China's Censorship of ESNI. `https://geneva.cs.umd.edu/posts/china-censors-esni/esni/`, 2020.

[37] Robert T. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software. CSTR 117, 1985.

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

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

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

[41] 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.

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

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

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

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

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

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

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

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

[50] Eric Cronin, Micah Sherr, and Matthew Blaze. The Eavesdropper's Dilemma, 2006.

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

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

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

[54] Roya Ensafi. CensoredPlanet Raw Data. https://censoredplanet.org/data/raw.

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

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

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

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

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

[60] Wenxuan Zhou, Amir Houmansadr, Matthew Caesar, and Nikita Borisov. SWEET: Serving the Web by Exploiting Email Tunnels. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

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

[62] Amir Houmansadr, Thomas Riedl, Nikita Borisov, and Andrew Singer. IP over Voice-over-IP for censorship circumvention. In *arXiv preprint arXiv:1207.2683*, 2012.

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

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

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

[66] Daniel Ellard, Christine Jones, Victoria Manfredi, W. Timothy Strayer, Bishal Thapa, Megan Van Welie, and Alden Jackson. Rebound: Decoy routing on asymmetric routes via error messages. 2015.

[67] Amir Houmansadr, Giang T. K. Nguyen, Matthew Caesar, and Nikita Borisov. Cirripede: Circumvention Infrastructure using Router Redirection with Plausible Deniability. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[68] Dave Levin, Youndo Lee, Luke Valenta, Zhihao Li, Victoria Lai, Cristian Lumenzanu, Neil Spring, and Bobby Bhattacharjee. Alibi Routing. In *ACM SIGCOMM*, 2015.

[69] Qiyan Wang, Xun Gong, Giang T.K. Nguyen, Amir Houmansadr, and Nikita Borisov. CensorSpoofer: Asymmetric communication using IP Spoofing for Censorship-resistant Web Browsing. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

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

[71] Kevin Bock, Yair Fax, Kyle Reese, Jasraj Singh, and Dave 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.

[72] Soo-Jin Moon, Jeffrey Helt, Yifei Yuan, Yves Bieri, Sujata Banerjee, Vyas Sekar, Wenfei Wu, Mihalis Yannakakis, and Ying Zhang. Alembic: Automated

Model Inference for Stateful Network Functions. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

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

[74] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[75] Scott Michael Seal. Optimizing Web Application Fuzzing with Genetic Algorithms and Language Theory. In *Master of Science Thesis*, 2016.

[76] Li Haifeng, Wang Shaolei, Zhang Bin, Shuai Bo, and Tang Chaojing. Network protocol security testing based on fuzz. In *International Conference on Computer Science and Network Technology (ICCSNT)*, 2015.

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

[78] Xavi Mendez. WFuzz: The Web Fuzzer, 2020. `wfuzz.io`.

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

[80] Lawrence Davis. Handbook of genetic algorithms. 1991.

[81] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, July 2012.

[82] NetFilter. https://netfilter.org.

[83] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 239(2), 2014.

[84] Mark Handley, Vern Paxson, and Christian Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-To-End Protocol Semantics. In *USENIX Security Symposium*, 2001.

[85] Scapy. https://scapy.net.

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

[87] Censorship of Alexa Top 1000 Domains in China. `https://en.greatfire.org/search/alexa-top-1000-domains`, 2019.

[88] Ram Sundara Raman, Leonid Evdokimov, Eric Wustrow, Alex Halderman, and Roya Ensafi. Kazakhstan's HTTPS Interception. https://censoredplanet.org/kazakhstan, 2019.

[89] Kazakhstan's HTTPS Interception Live! https://censoredplanet.org/kazakhstan/live, 2019.

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

[91] Roger Dingledine. Obfsproxy: the next step in the censorship arms race. https://blog.torproject.org/obfsproxy-next-step-censorship-arms-race, 2012.

[92] Sigal Samuel. China is installing a secret surveillance app on tourists' phones. https://www.vox.com/future-perfect/2019/7/3/20681258/china-uighur-surveillance-app-tourist-phone, 2019.

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

[94] agrabeli. Internet Censorship in Iran: Findings from 2014-2017. https://blog.torproject.org/internet-censorship-iran-findings-2014-2017, 2017.

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

[96] TelegramMessenger. MTProxy. https://github.com/TelegramMessenger/MTProxy, 2019.

[97] Inc. The Tor Project. Tor Project: Bridges. https://2019.www.torproject.org/docs/bridges.html.en.

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

[99] DNS Transport over TCP - Implementation Requirements. RFC 7766, RFC Editor, March 2016.

[100] Transmission Control Protocol. RFC 793, RFC Editor, September 1981.

[101] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. Measuring HTTPS Adoption on the Web. In *USENIX Security Symposium*, 2017.

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

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

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

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

[106] Ram Sundara Raman, Leonid Evdokimov, Eric Wustrow, Alex Halderman, and Roya Ensafi. Kazakhstan's HTTPS Interception. https://censoredplanet.org/kazakhstan, 2019.

[107] Ram Sundara Raman, Leonid Evdokimov, Eric Wustrow, Alex Halderman, and Roya Ensafi. Investigating Large Scale HTTPS Interception in Kazakhstan. In *ACM Internet Measurement Conference (IMC)*, 2020.

[108] Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin Kirda. T-Reqs: HTTP Request Smuggling with Differential Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[109] RFC 2616, 1999. https://datatracker.ietf.org/doc/html/rfc2616.

[110] Roy Fielding and Julian Reschke. RFC 7230, 2014. https://www.rfc-editor.org/rfc/rfc7230.html.

[111] Roy Fielding and Julian Reschke. RFC 7231, 2014. https://www.rfc-editor.org/rfc/rfc7231.html.

[112] Roy Fielding and Julian Reschke. RFC 7232, 2014. https://www.rfc-editor.org/rfc/rfc7232.html.

[113] Roy Fielding, Yves Lafon, and Julian Reschke. RFC 7233, 2014. https://www.rfc-editor.org/rfc/rfc7233.html.

[114] Roy Fielding, Mark Nottingham, and Julian Reschke. RFC 7234, 2014. https://www.rfc-editor.org/rfc/rfc7234.html.

[115] Roy Fielding and Julian Reschke. RFC 7235, 2014. https://www.rfc-editor.org/rfc/rfc7235.html.

[116] Tim Berners-Lee, Roy Fielding, and Larry Masinter. RFC 3986, 2005. https://www.rfc-editor.org/rfc/rfc3986.

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

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

[119] COMMUNITY-LED DEVELOPMENT "THE APACHE WAY", 2022. https://www.apache.org/.

[120] NGINX Part of F5, 2022. https://www.nginx.com/.

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

[122] Charles Hornig. RFC 894, 1984. https://datatracker.ietf.org/doc/html/rfc894.

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

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

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

[126] Ram Sundara Raman, Adrian Stoll, Jakub Dalek, Armin Sarabi, Reethika Ramesh, Will Scott, and Roya Ensafi. Measuring the deployment of network censorship filters at global scale. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

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

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

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

[130] Collin Anderson. Dimming the Internet: Detecting Throttling as a Mechanism of Censorship in Iran. In *arXiv preprint arXiv:1306.4361*, 2013.

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

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

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

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

[135] Kevin Bock, Pranav Bharadwaj, Jasraj Singh, and Dave Levin. Your censor is my censor: Weaponizing censorship infrastructure for availability attacks. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2021.

[136] Catalin Cimpanu. Russia wants to ban the use of secure protocols such as TLS 1.3, DoH, DoT, ESNI. https://www.zdnet.com/article/russia-wants-to-ban-the-use-of-secure-protocols-such-as-tls-1-3-doh-dot-esni/, 2020.

[137] Xueyang Xu, Z. Morley Mao, and J. Alex Halderman. "Internet Censorship in China: Where Does the Filtering Occur?". In Neil Spring and George F. Riley, editors, *Passive and Active Measurement*, pages 133–142, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[138] Russia Censoring Omitted SNI. https://github.com/net4people/bbs/issues/10, 2019.

[139] Christian Rossow. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[140] Kulvinder Singh and Ajit Singh. Memcached DDoS Exploits: Operations, Vulnerabilities, Preventions and Mitigations. 2018.

[141] UDP-Based Amplification Attacks: Alert (TA14-017A). National Cyber Awareness System Alerts, January 2014. https://www.us-cert.gov/ncas/alerts/TA14-017A.

[142] CVE-2018-1000115: Memcached version 1.5.5. National Vulnerability Database, March 2018. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2018-1000115.

[143] Sam Kottler. February 28th DDoS incident report. https://github.blog/2018-03-01-ddos-incident-report/, Mar 2018.

[144] Ben Jones, Tzu-Wen Lee, Nick Feamster, and Phillipa Gill. Automated Detection and Fingerprinting of Censorship Block Pages. In *ACM Internet Measurement Conference (IMC)*, 2014.

[145] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast Internet-wide Scanning and its Security Applications. In *USENIX Security Symposium*, 2013.

[146] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In *USENIX Security Symposium*, 2014.

[147] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Hell of a Handshake: Abusing TCP for Reflective Amplification DDoS Attacks. In *USENIX Security Symposium*, 2014.

[148] Jakub Czyz, Michael Kallitsis, Manaf Gharaibeh, Christos Papadopoulos, Michael Bailey, and Manish Karir. Taming the 800 Pound Gorilla: The Rise and Decline of NTP DDoS Attacks. In *ACM Internet Measurement Conference (IMC)*, 2014.

[149] Robert Beverly and Steven Bauer. The Spoofer Project: inferring the Extent of Source Address Filtering on the Internet. In *USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, 2005.

[150] The Spoofer Project: State of IP Spoofing. `https://spoofer.caida.org/summary.php`.

[151] Vern Paxson. End-to-End Routing Behavior in the Internet. In *ACM SIGCOMM*, 1996.

[152] Rob Sherwood, Bobby Bhattacharjee, and Ryan Braud. Misbehaving TCP Receivers Can Cause Internet-Wide Congestion Collapse. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[153] Bill Marczak, Nicholas Weaver, Jakub Dalek, Roya Ensafi, David Fifield, Sarah McKune, Arn Rey, John Scott-Railton, Ron Deibert, and Vern Paxson. An Analysis of China's "Great Cannon". In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2015.

[154] Marios Anagnostopoulos, Georgios Kambourakis, Panagiotis Kopanos, Georgios Louloudakis, and Stefanos Gritzalis. DNS Amplification Attack Revisited. *Computers & Security*, 39(B):475–485, November 2013.

[155] Bingshuang Liu, Skyler Berg, Jun Li, Tao Wei, Chao Zhang, and Xinhui Han. The Store-and-Flood Distributed Reflective Denial of Service Attack. 2014.

[156] Matthew Sargent, John Kristoff, Vern Paxson, and Mark Allman. On the Potential Abuse of IGMP. *ACM SIGCOMM Computer Communication Review (CCR)*, 47(1), 2017.

[157] Ram Sundara Raman, Prerana Shenoy, Katharina Kohls, and Roya Ensafi. Censored Planet: An Internet-wide, Longitudinal Censorship Observatory. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[158] Citizen Lab. Block test list. https://github.com/citizenlab/test-lists.

[159] MaxMind. GeoLite2. https://dev.maxmind.com/geoip/geoip2/geolite2, 2020.

[160] Freedom House. Freedom in the world report. https://freedomhouse.org/countries/freedom-world/scores.

[161] Arturo Filasto and Jacob Appelbaum. OONI: Open Observatory of Network Interference. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2012.

[162] Matthew Prince. The DDoS That Almost Broke the Internet. Cloudflare Blog, March 2013. https://blog.cloudflare.com/the-ddos-that-almost-broke-the-internet/.

[163] Gordon Lyon. nmap. https://nmap.org/.

[164] Paul Pearce, Ben Jones, Frank Li, Nick Feamster, Nick Weaver, and Vern Paxson. Global Measurement of DNS Manipulation. In *USENIX Annual Technical Conference*, 2017.

[165] Craig Partridge and Mark Allman. Addressing ethical considerations in network measurement papers. In *NS Ethics@ SIGCOMM*, 2015.

[166] Let's Encrypt Stats. Percentage of Web Pages Loaded by Firefox Using HTTPS. https://letsencrypt.org/stats/#percent-pageloads, 2018.

[167] TCP Middlebox Reflection: Coming to a DDoS Near You, 2022. https://www.akamai.com/blog/security/tcp-middlebox-reflection.

[168] Roya Ensafi, Philipp Winter, Abdullah Mueen, and Jedidiah R. Crandall. Analyzing the Great Firewall of China Over Space and Time. In *Privacy Enhancing Technologies Symposium (PETS)*, 2015.

[169] Daiyuu Nobori and Yasushi Shinjo. VPN Gate: A Volunteer-Organized Public VPN Relay System with Blocking Resistance for Bypassing Government Censorship Firewalls. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

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

[171] Yossi Gilad and Amir Herzberg. Off-Path Attacking the Web. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[172] Florian Adamsky, Syed Ali Khayam, Rudolf Jäger, and Muttukrishnan Rajarajan. P2P File-Sharing in Hell: Exploiting BitTorrent Vulnerabilities to Launch Distributed Reflective DoS Attacks. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[173] Jonas Bushart. Optimizing Recurrent Pulsing Attacks using Application-Layer Amplification of Open DNS Resolvers. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[174] Jan Beznazwy and Amir Houmansadr. How china detects and blocks shadowsocks. In *ACM Internet Measurement Conference (IMC)*, 2020.

[175] Reethika Ramesh Ram, Sundara Raman, Matthew Bernhard, Victor Ongkowijaya, Leonid Evdokimov, Annie Edmundson, S. Sprecher, Muhammad Ikram, and Roya Ensafi. Decentralized Control: A Case Study of Russia. In *Network and Distributed System Security Symposium (NDSS)*, 2020.