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

Table 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**.**

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 [21]. 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:

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

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

Corrupting the sequence number of the ACK packet breaks the original three-way handshake, but the ACK flag set in the PSH/ACK packet finishes the handshake. Table 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:

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

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

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 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, 21, 33, 41], 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 [33, 41], 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 [41]. 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.

5.3 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 [49]. 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 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 [32]. Though this interception has fortunately since ended [20], 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 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.

5.4 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* [41] do not work against the GFW. This includes related subspecies, such as the *TCB Turnaround* [41].

TCB Teardown using a `FIN` or `FIN/ACK` packet [41] 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 [41], 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 [41] 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.

6 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 [41] 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[3] 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 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 paper—our primary goal 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.

Limitations of Our Evaluation We did not evaluate our system on as many vantage points in China as some prior work [33, 41] 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 [21, 33, 41], we evaluated Geneva by running it solely on hosts that we rented and controlled—as opposed to recruiting unwitting users [4]—to mitigate ethical concerns.

7 RELATED WORK

Circumventing Censors In this paper, we have focused on automating and improving packet-manipulation-based censorship evasion (we reviewed prior work in that space in §2). Additionally, there is a much wider space of strategies for circumventing censorship. Researchers have explored tunneling traffic over a wide variety of mediums, including email [50], video games [39], VoIP [18], SSH [43], WebRTC [11], HTTP [12], just to name a few. Other systems seek to hide the true destination of traffic, such as with Tor [8],

[3] 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.

domain fronting [13], Decoy or Refraction Routing [9, 19, 46, 47], or to avoid the censoring country altogether (Alibi Routing [24], DeTor [25]). Traffic mimicry systems have also been developed to disguise network traffic as another protocol [28, 40, 42]; though these appear to have inherent limitations [17].

Geneva is orthogonal to all of these systems, and, as demonstrated with INTANG [41], could be used in tandem with them to help bolster their ability to circumvent censors.

Fuzz Testing Fuzz testers [22] mutate inputs nondeterministically in an effort to evaluate the correctness, security, and coverage of programs. At a high level, Geneva shares some properties with fuzz testers: both perform random mutations and use the output of a program (a censor) to evaluate whether those mutations were beneficial. However, there is a subtle but fundamental distinction: whereas fuzz testers generate inputs, Geneva generates what amounts to small pieces of code (packet manipulation strategies) that are in turn applied to inputs (user traffic). Geneva thus performs its mutations and evolutions over the space of manipulation actions (drop, tamper, etc.), not over the input space (packets) itself.

Genetic algorithms have been used for fuzzing, including in the well known American Fuzzy Lop (AFL) [2] and iFuzzer [38]. Genetic algorithm fuzzing techniques have been applied to web applications [36] and other popular protocols [15]. To our knowledge, we are the first to apply such techniques to censorship evasion.

8 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 paper, we presented Geneva, a genetic algorithm for automatically discovering censorship evasion strategies against on-path network censors. Through evaluation both in-lab and against the GFW, we have demonstrated that Geneva efficiently discovers strategies, and that its genetic building blocks allow it to both re-derive all previously published schemes that it can support, as well as derive altogether new strategies that prior work posited would not be effective. We believe Geneva represents an important first step towards automating censorship evasion. To this end, we have made our code and data publicly available at `https://geneva.cs.umd.edu`

ACKNOWLEDGMENTS

We thank Ramakrishna Padmanabhan, Neil Spring, the Breakerspace lab, and the anonymous reviewers for their helpful feedback. This research was supported in part by the Open Technology Fund and NSF grant CNS-1816802.

REFERENCES

[1] Claudio Agosti and Giovanni Pellerano. 2011. SniffJoke: transparent TCP connection scrambler. `https://github.com/vecna/sniffjoke`. (2011).
[2] american fuzzy lop [n. d.]. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. ([n. d.]).
[3] Simurgh Aryan, Homa Aryan, and J. Alex Halderman. 2013. Internet Censorship in Iran: A First Look. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*.
[4] Sam Burnett and Nick Feamster. 2015. Encore: Lightweight Measurement of Web Censorship with Cross-Origin Requests. In *ACM SIGCOMM*.
[5] Censorship of Alexa Top 1000 Domains in China [n. d.]. Censorship of Alexa Top 1000 Domains in China. https://en.greatfire.org/search/alexa-top-1000-domains. ([n. d.]).
[6] Richard Clayton, Steven J. Murdoch, and Robert N. M. Watson. 2006. Ignoring the Great Firewall of China. In *Privacy Enhancing Technologies Symposium (PETS)*.
[7] Lawrence Davis. 1991. *Handbook of genetic algorithms*. CUMINCAD.
[8] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*.
[9] Daniel Ellard, Christine Jones, Victoria Manfredi, W. Timothy Strayer, Bishal Thapa, Megan Van Welie, and Alden Jackson. 2015. Rebound: Decoy routing on asymmetric routes via error messages. In *IEEE Conference on Local Computer Networks (LCN)*.
[10] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. 2015. Examining How the Great Firewall Discovers Hidden Circumvention Servers. In *ACM Internet Measurement Conference (IMC)*.
[11] David Fifield. 2017. Threat modeling and circumvention of Internet censorship. In *PhD thesis*.
[12] David Fifield, Nate Hardison, Jonathan Ellithorpe, Emily Stark, Dan Boneh, Roger Dingledine, and Phil Porras. 2012. Evading Censorship with Browser-Based Proxies. In *Privacy Enhancing Technologies Symposium (PETS)*.
[13] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. 2015. Blocking-resistant communication through domain fronting. In *Privacy Enhancing Technologies Symposium (PETS)*.
[14] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (July 2012), 2171–2175.
[15] Li Haifeng, Wang Shaolei, Zhang Bin, Shuai Bo, and Tang Chaojing. 2015. Network protocol security testing based on fuzz. In *International Conference on Computer Science and Network Technology (ICCSNT)*.
[16] Mark Handley, Vern Paxson, and Christian Kreibich. 2001. Network Intrusion Detection: Evasion, Traffic Normalization, and End-To-End Protocol Semantics. In *USENIX Security Symposium*.
[17] Amirr Houmansadr, Chad Brubaker, and Vitaly Shmatikov. 2013. The Parrot is Dead: Observing Unobservable Network Communications. In *IEEE Symposium on Security and Privacy*.
[18] Amir Houmansadr, Thomas Riedl, Nikita Borisov, and Andrew Singer. 2012. IP over Voice-over-IP for censorship circumvention. In *arXiv preprint arXiv:1207.2683*.
[19] Amir Housmandr, Giang T. K. Ngyuen, Matthew Caesar, and Nikita Borisov. 2011. Cirripede: Circumvention Infrastructure using Router Redirection with Plausible Deniability. In *ACM Conference on Computer and Communications Security (CCS)*.
[20] Kazakhstan's HTTPS Interception Live! 2019. Kazakhstan's HTTPS Interception Live! `https://censoredplanet.org/kazakhstan/live`. (2019).
[21] Sheharbano Khattak, Mobin Javed, Philip D. Anderson, and Vern Paxson. 2013. Towards Illuminating a Censorship Monitor's Model to Facilitate Evasion. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*.
[22] George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*.
[23] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems* 18, 3 (2000), 263–297.
[24] Dave Levin, Youndo Lee, Luke Valenta, Zhihao Li, Victoria Lai, Cristian Lumenzanu, Neil Spring, and Bobby Bhattacharjee. 2015. Alibi Routing. In *ACM SIGCOMM*.
[25] Zhihao Li, Stephen Herwig, and Dave Levin. 2017. DeTor: Provably Avoiding Geographic Regions in Tor. In *USENIX Security Symposium*.
[26] Moxie Marlinspike. 2017. Doodles, stickers, and censorship circumvention for Signal Android. `https://signal.org/blog/doodles-stickers-censorship/`. (2017).
[27] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* 239, 2 (2014).
[28] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. 2012. SkypeMorph: Protocol Obfuscation for Tor Bridges. In *ACM Conference on Computer and Communications Security (CCS)*.
[29] Zubair Nabi. 2013. The Anatomy of Web Censorship in Pakistan. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*.
[30] NetFilter [n. d.]. NetFilter. https://netfilter.org. ([n. d.]).
[31] Thomas H. Ptacek and Timothy N. Newsham. 1998. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. In *Secure Networks, Inc.*
[32] Ram Sundara Raman, Leonid Evdokimov, Eric Wustrow, Alex Halderman, and Roya Ensafi. 2019. Kazakhstan's HTTPS Interception. `https://censoredplanet.org/kazakhstan`. (2019).
[33] Fangfan Liand Abbas Razaghpanah, Arash Molavi Kakhki, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. 2017. lib·erate, (n): A library for exposing (traffic-classification) rules and avoiding them efficiently. In *ACM Internet Measurement Conference (IMC)*.

[34] Reporters Without Borders. 2013. Enemies of the Internet 2013 Report. https://surveillance.rsf.org/en/wp-content/uploads/sites/2/2013/03/enemies-of-the-internet_2013.pdf. (March 2013).
[35] Scapy [n. d.]. Scapy. https://scapy.net. ([n. d.]).
[36] Scott Michael Seal. 2016. Optimizing Web Application Fuzzing with Genetic Algorithms and Language Theory. In *Master of Science Thesis*.
[37] Signal. 2017. 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).
[38] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer using Genetic Programming. In *European Symposium on Research in Computer Security (ESORICS)*.
[39] Paul Vines and Tadayoshi Kohno. 2015. Rook: Using Video Games as a Low-Bandwidth Censorship Resistant Communication Platform. In *Workshop on Privacy in the Electronic Society (WPES)*.
[40] Qiyan Wang, Xun Gong, Giang T. K. Nguyen, Amir Houmansadr, and Nikita Borisov. 2012. CensorSpoofer: Asymmetric Communication Using IP Spoofing for Censorship-Resistant Web Browsing. In *ACM Conference on Computer and Communications Security (CCS)*.
[41] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V. Krishnamurthy. 2017. Your State is Not Mine: A Closer Look at Evading Stateful Internet Censorship. In *ACM Internet Measurement Conference (IMC)*.
[42] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. 2012. StegoTorus: A Camouflage Proxy for the Tor Anonymity System. In *ACM Conference on Computer and Communications Security (CCS)*.
[43] Brandon Wiley. [n. d.]. Dust: A Blocking-Resistant Internet Transport Protocol. http://blanu.net/Dust.pdf. ([n. d.]).
[44] Philipp Winter. 2012. brdgrd (Bridge Guard). https://github.com/NullHypothesis/brdgrd. (2012).
[45] Philipp Winter and Jedidiah R. Crandall. 2012. The Great Firewall of China: How It Blocks Tor and Why It Is Hard to Pinpoint. *;login:* 37, 6 (2012), 42–50.
[46] Eric Wustrow, Colleen M. Swanson, and J. Alex Halderman. 2014. TapDance: End-to-Middle Anticensorship without Flow Blocking. In *USENIX Annual Technical Conference*.
[47] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. 2011. Telex: Anticensorship in the Network Infrastructure. In *USENIX Annual Technical Conference*.
[48] Xueyang Xu, Morley Mao, and J. Alex Halderman. 2011. Internet Censorship in China: Where Does the Filtering Occur?. In *Passive and Active Network Measurement Workshop (PAM)*.
[49] Tarun Kumar Yadav, Akshat Sinha, Devashish Gosain, Piyush Kumar Sharma, and Sambuddho Chakravarty. 2018. Where The Light Gets In: Analyzing Web Censorship Mechanisms in India. In *ACM Internet Measurement Conference (IMC)*.
[50] Wenxuan Zhou, Amir Houmansadr, Matthew Caesar, and Nikita Borisov. 2013. SWEET: Serving the Web by Exploiting Email Tunnels. In *Privacy Enhancing Technologies Symposium (PETS)*.

APPENDIX

In this appendix, we aggregate several low-level details about prior work: We provide a detailed list of all of the mock censors we validated against in-lab (Table 2) and a full list of all previously published strategies (Table 3).

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

Species	Strategy	Found? [21]	[33]	[41]	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 reassembly 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	GET w/ > 1 space between method and URI	✓			
	GET w/ keyword at location > 2048	✓			
	GET w/ keyword in 2nd or higher of multiple requests in one segment	✓			
	GET w/ URL encoded (except %-encoding)	✓			

Table 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.**