# Geneva: Evolving Censorship Evasion Strategies

Kevin Bock
University of Maryland

George Hughey
University of Maryland

Xiao Qiang
UC Berkeley

Dave Levin
University of Maryland

## ABSTRACT

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 paper, we take a drastic departure from the previously *manual* evade-detect cycle by developing techniques to *automate* the discovery of censorship evasion strategies. We present Geneva, a novel genetic algorithm that evolves packet-manipulation-based censorship evasion strategies against nation-state level censors. Geneva composes, mutates, and evolves sophisticated strategies out of four basic packet manipulation primitives (drop, tamper headers, duplicate, and fragment). With experiments performed both in-lab and against several real censors (in China, India, and Kazakhstan), we demonstrate that Geneva is able to quickly and independently re-derive most strategies from prior work, and derive novel subspecies and altogether new species of packet manipulation strategies. Moreover, Geneva discovers successful strategies that prior work posited were not effective, and evolves extinct strategies into newly working variants. We analyze the novel strategies Geneva creates to infer previously unknown behavior in censors. Geneva is a first step towards automating censorship evasion; to this end, we have made our code and data publicly available.

## CCS CONCEPTS

• **Social and professional topics** → **Technology and censorship**; • **Computing methodologies** → **Genetic algorithms**; • **General and reference** → *Measurement*.

## KEYWORDS

Censorship; Genetic Algorithms; Geneva

**ACM Reference Format:**
Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. 2019. Geneva: Evolving Censorship Evasion Strategies. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3363189


*CCS '19, November 11–15, 2019, London, United Kingdom*

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3363189

## 1 INTRODUCTION

Multiple nations around the world today engage in country-wide censorship of Internet traffic. Although there are many forms of censorship—including political pressure [34] and outright blocking of certain protocols [10, 17]—the most pervasive form of online censorship involves *in-network monitoring* and censoring of forbidden keywords. China [48], Pakistan [29], and more [34] deploy on-path monitors—similar to network intrusion detection systems (NIDS) [21]—to detect and tear down network connections that carry a prohibited word or domain name. The distinguishing feature of such *on-path* censors is that they are not one of the hops in the target's communication path; rather, they monitor copies of packets and inject new packets (e.g., TCP RSTs) to interfere with connections they deem to be inappropriate.

One approach to circumvent on-path censorship is to perform client-side packet manipulation to confuse censors into not tearing down connections. The high-level insight is that on-path censors—like all NIDS—must maintain per-flow state in order to track TCP connections. If this state can be invalidated, then the censor will be unable to tear down the connection (e.g., if the censor perceives the wrong sequence numbers) or simply will not try to do so in the first place (e.g., if it believes the connection has already been terminated). For instance, sending a TCP RST packet with a TTL large enough to reach the censor but too small to reach the destination will make the censor's state (the connection has been terminated) inconsistent with the destination's (the connection is ongoing) [21].

For years, security researchers have been engaging in a cat-and-mouse game, developing new packet-manipulation-based evasion strategies, and updating censors to identify and defeat them. Unfortunately, censors have an inherent advantage: discovering new censorship evasion schemes is a laborious, manual process (particularly for successful packet manipulation schemes), typically involving researchers first measuring and understanding how a particular censor works before developing evasion strategies [33, 41]. As a result, when a new censorship technique is deployed, there can often be considerable loss of availability until the technique is measured, reverse-engineered, and circumvented [26, 37].

In this paper, we take a drastic departure from the previous evade-detect cycle in censorship evasion. We present Geneva[1], a genetic algorithm that *automatically* evolves censorship evasion strategies against nation-state censors. As with all prior packet-manipulation-based schemes, Geneva takes advantage of the fact that nation-state censors employ *incomplete* implementations of networking stacks. As a result, Geneva is able to run exclusively on the client-side and requires no cooperation from other hosts, including the intended destination. Unlike prior approaches, however, Geneva *discovers*

[1] *Gen*etic *Eva*sion.

packet manipulation strategies on its own, with no prior knowledge of the censor's implementation (nor shortcomings thereof).

The primary technical challenge behind designing a genetic algorithm for censorship evasion is in balancing between the breadth of strategies it can explore and the efficiency in finding successful strategies. On one extreme, we could permit the genetic algorithm to perform arbitrary bit manipulations; this could eventually learn *all* strategies, but would likely take an inordinate amount of time to do so. On the other extreme, we could permit it to only perform the packet manipulations discovered in prior work; this would likely be efficient, but risks being confined only to rediscover established techniques. One of our key contributions is a design that balances breadth with efficiency: drawing inspiration from modular router designs [23], Geneva composes sophisticated packet manipulation strategies out of basic primitives.

We have implemented Geneva and evaluated it against the Great Firewall (GFW) of China, ISP censorship in India, and HTTPS interception in Kazakhstan. Our results show that Geneva is highly successful at discovering new strategies quickly and efficiently. Across 27 experiments against the GFW, Geneva discovered 4 unique species, 8 subspecies (5 of which are novel), and 21 fundamentally different variants of these subspecies. Against both India and Kazakhstan, Geneva discovers 5 species. In in-lab experiments, Geneva independently re-derived 30 of 36 strategies suggested by prior work (failing to generate only the strategies for which we did not give Geneva the building blocks to express, such as sleeping between packets). Moreover, Geneva is efficient: Geneva re-derives all strategies in a manner of hours.

**Contributions** We summarize our contributions as follows:

- We present Geneva, a novel genetic algorithm that efficiently and independently discovers new packet manipulation strategies to circumvent on-path censorship.
- We present a detailed evaluation of Geneva using experiments both in-lab and against China, India, and Kazakhstan, and analyze the circumvention strategies that Geneva has discovered.
- We show that Geneva is able to re-derive almost all previously published evasion strategies, derive successful strategies that prior work posited were ineffective, and derive two altogether new species of evasion strategy.
- We also show that several evasion strategies that were successful in prior work have since become extinct, but that by seeding Geneva with extinct approaches, it can evolve them into new, successful strategies.

## 2 BACKGROUND

### 2.1 On-Path Censorship

A key component to the design of most nation state censors, including the GFW, is that they cannot rely entirely on in-path censorship. In-path censorship is the type of traffic monitoring most network administrators are familiar with: a direct man-in-the-middle for every connection where the censor processes every individual packet. In-path censors represent a stronger attack model, as they have the power to directly manipulate or drop traffic. However, in-path systems are more computationally expensive to run, and can impose a higher level of overhead to traffic processing. Therefore, although the GFW does have in-path components (it maintains a relatively small IP blacklist, and any packets sent to IPs on that blacklist will be immediately dropped by the GFW [45]), the majority of censorship performed by the GFW is done through "man on the side" attacks, or "on-path" censorship. "On-path" or "man-on-the-side" systems can see traffic going by, but sit physically outside the network path—these censors can view traffic, but generally cannot modify or drop the traffic. As a consequence, when the on-path system wants to censor or terminate a connection, it does so by injecting packets into the stream to disrupt it. For example, to censor a TCP connection, the GFW injects TCP `RST` packets; Airtel's ISP injects a fake response with an HTTP block page.

The backbone of most nation-state level censorship today is Deep Packet Inspection (DPI), which operates similarly to most NIDS. In order to perform DPI, any inspecting NIDS must have a near-complete implementation of the relevant payload stack (such as TCP/IP). In particular, stateful analysis of TCP connections requires NIDS to track information about the state of each connection, such as with Transmission Control Blocks (TCBs). It is up to the specific censor implementation to decide when to begin tracking state for a given connection; for example, it can choose to do so only after a valid three-way handshake [21], but also can do so based on more partial observation, such as the presence of a `SYN` flag. Due to the vast number of active connections, another important consideration for the censor is to decide when to stop maintaining state for a given connection. Monitors can commonly "tear down" their TCB for a given connection when a `FIN` handshake is done, or in the presence of valid `RST` packets [41].

### 2.2 Packet-manipulation-based Evasion

Censors typically "soft-fail"—if a packet arrives for which the censor has no matching state or mechanism to decide whether or not to censor the packet, the monitor does not censor the connection. For example, if a packet is not on any IP blacklist and has no matching TCB for stateful analysis, the censor will default to allowing the connection. Additionally, due to differences in protocol stack implementation at various endpoints, previous work has suggested that no NIDS will be capable of perfectly reconstructing a packet stream in a way that mimics the end host [31]. Many of these middleboxes, NIDS, and DPI systems also run incomplete TCP/IP stacks [33].

Discrepancies between how these systems handle TCP packets present an opportunity for censorship evasion. If a client sends specifically crafted packets, it is possible to trick a NIDS into desynchronizing or deactivating the TCB for a given connection, or even into ignoring the traffic completely, allowing the rest of the packets of the connection to pass through uncensored. As the censor soft-fails and functions primarily as an on-path system, this is tantamount to a successful defeat of the censor. Importantly, this does not require support or cooperation from outside the censoring regime. Evasion can be accomplished simply by altering the packet stream on the client side.

A *packet-manipulation-based* evasion strategy is a sequence of packet manipulations (modifying, adding, or dropping packets) to evade a censor. Prior work has suggested different strategies that can be used to invalidate the state that censors maintain. Here, we discuss three recent efforts that are highly relevant to our design

and evaluation; we review related work more broadly in §7 and we summarize these papers' strategies in Table 3 in the appendix.

**Khattak et al. [21]** In 2013, Khattak et al. [21] crafted 17 different evasion strategies to exploit specific implementational weaknesses against the GFW. Though a follow-up study [41] found most of the strategies to be ineffective in 2017, Geneva identifies some strategies that fall within their taxonomy that are still highly effective today.

**INTANG [41]** In 2017, Wang et al. [41] released a comprehensive work on evaluating the GFW. They 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.

**lib·erate [33]** Li et al. [33] 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.

Our system, Geneva, takes a drastically different approach from those described in each of these three papers. Each of these works developed strategies empirically by hand, informed by human interaction with the censor. Although some work has included highly successful automated components [33, 41], these amounted to choosing from a pre-built library of evasion techniques based on the network path, application, or server. Geneva, on the other hand, derives the evasion techniques itself, and can thus naturally adapt to different deployment scenarios.

### 2.3 Packet Manipulation Strategy Taxonomy

When comparing strategies, it is useful to define a taxonomic rank; we use species, subspecies, and variants. The highest level classification is *species*, a broad class of strategies classified by the type of weakness it exploits in a censor implementation. *TCB Teardown* is an example of one such species; if the censor did not prematurely teardown TCBs, all the strategies in this species would cease to function. Within each species, different *subspecies* represent unique ways to exploit the weakness that defines the strategy. For example, injecting an insertion TCP `RST` packet would comprise one subspecies within the *TCB Teardown* species; injecting a TCP `FIN` would comprise another. Within each subspecies, we further record *variants*, unique strategies that leverage the same attack vector, but do so slightly differently: corrupting the checksum field `RST/ACK` packet is one variant of the RST subspecies of the *TCB Teardown* species; corrupting the `ack` field is another.

We refer to specific individuals as *extinct* if they once worked against a censor but are no longer effective (less than 5% success rate). As we show in §5, multiple species discussed in prior work, including *TCB Creation*, *Data Reassembly* and *Traffic Misclassification* are currently extinct. That formerly successful approaches could, after a few years, become ineffective lends further motivation for a technique that can quickly *learn* new strategies.

## 3 Geneva DESIGN

Geneva's goal is to automate the process of discovering new censorship evasion strategies. In this section, we describe its genetic algorithm-based design in terms of its building blocks and how it composes and evolves them over time. We begin by providing a high-level overview of the approach.

### 3.1 Overview and Challenges

Genetic algorithms [7] 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.

### 3.2 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 modes 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* [41] (characterized by sending a `SYN/ACK` before the three-way handshake) and *RST-Drop* [6]. (Unfortunately, as we will see in Section 5, 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 [21, 33, 41] 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. [21], and those that (2) paused for 40–240 seconds, as was done by lib·erate [33]. These are not fundamental limitations: one could easily extend Geneva to support HTTP manipulation or sleeping through `tamper` actions. For this paper, 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 §4, 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.

## 3.3 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(`$A_1$`,` $A_2$`)` → `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[2]: 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 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.

[2]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.

**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 §5, 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* [14]. 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 5, 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.

### 3.4 Implementation

We implemented Geneva in approximately 6,000 lines of Python. Geneva runs strictly at the client, and uses NetfilterQueue [30] 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 §5, 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 [1, 21, 33, 41, 44], root privilege is necessary for most of their packet manipulations. However, we demonstrate in §5 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 [21, 33], we speculate that Geneva would also fare well, but this is beyond the scope of this paper.

## 4 VALIDATION

In this section, we validate Geneva's design by investigating whether it can re-derive strategies found from prior work [33, 41]. 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 [3, 29, 33, 41]. 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 in the appendix.

We implemented a Dockerized [27] 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 [21, 33, 41], 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 [33, 41], perform state exhaustion [21, 41], or perform HTTP-specific tweaks [21] (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` [41]. 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 [16, 33, 41]. While training in the lab, Geneva identified 9 now-patched bugs in scapy [35], a bug in Docker for Mac [27], and a bug in NetfilterQueue [30].

All of the discovered strategies require only 1–2 action trees in the outbound forest to express; besides the initial strategy of dropping inbound `RST`s, 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.

# 5 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 1 lists the success rates, descriptions, and taxonomy of all strategies and strategy variants Geneva found against these censors.

## 5.1 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 [41, 49], 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.

## 5.2 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" [41]: 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 [5]. 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 [21, 33, 41], 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 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 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 [33, 41], 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%.

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

**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)-| \/
```

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

| 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.**