# lib•erate, (n) : A library for exposing (traffic-classification) rules and avoiding them efficiently

Fangfan Li
Northeastern University
li.fa@husky.neu.edu

Abbas Razaghpanah
Stony Brook University
arazaghpanah@cs.stonybrook.edu

Arash Molavi Kakhki
Northeastern University
arash@ccs.neu.edu

Arian Akhavan Niaki
Stony Brook University
sakhavanniak@cs.stonybrook.edu

David Choffnes
Northeastern University
choffnes@ccs.neu.edu

Phillipa Gill
University of Massachusetts Amherst
phillipa@cs.umass.edu

Alan Mislove
Northeastern University
amislove@ccs.neu.edu

## ABSTRACT

Middleboxes implement a variety of network management policies (e.g., prioritizing or blocking traffic) in their networks. While such policies can be beneficial (e.g., blocking malware) they also raise issues of network neutrality and freedom of speech when used for application-specific differentiation and censorship. There is a poor understanding of how such policies are implemented in practice, and how they can be evaded efficiently. As a result, most circumvention solutions are brittle, point solutions based on manual analysis.

This paper presents the design and implementation of *lib·erate*, a tool for automatically identifying middlebox policies, reverse-engineering their implementations, and adaptively deploying custom circumvention techniques. Unlike previous work, our approach is application-agnostic, can be deployed unilaterally (i.e., only at one endpoint) on unmodified applications via a linked library or transparent proxy, and can adapt to changes to classifiers at runtime. We implemented a *lib·erate* prototype as a transparent proxy and evaluate it both in a testbed environment and in operational networks that throttle or block traffic based on DPI-based classifier rules, and show that our approach is effective across a wide range of middlebox deployments.

## CCS CONCEPTS

• **Networks → Middle boxes / network appliances**;

## KEYWORDS

Network Neutrality, Traffic Differentiation

**ACM Reference Format:**
Fangfan Li, 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 *Proceedings of IMC '17, London, United Kingdom, November 1–3, 2017,* 14 pages.
https://doi.org/10.1145/3131365.3131376


*IMC '17, November 1–3, 2017, London, United Kingdom*

ACM ISBN 978-1-4503-5118-8/17/11…$15.00
https://doi.org/10.1145/3131365.3131376

## 1 INTRODUCTION

It is common for ISPs to use middleboxes to manage their network traffic. Recent work [32, 35] showed that such middleboxes are often implemented using deep packet inspection (DPI) on network traffic, relying on packet contents (e.g., `Host` headers, TLS Server Name Indication (SNI) extensions, or protocol-specific fields) to selectively apply policies to flows. While such policies may sometimes be beneficent, there are many scenarios in which they may cause harm to users and/or service providers, e.g., by violating net neutrality, implementing traffic shaping, or censoring content.

In response, researchers have proposed a number of solutions to obfuscate traffic in a way that evades classification. In general, these approaches entail encrypting/proxying traffic [3, 19], transforming flows to resemble different protocols [48, 49], or selectively modifying contents of fields that are targeted by classifiers [24]. However, developers of evasion techniques find themselves in an arms race with network providers, where the scales are tipped in favor of the provider. Designing evasion schemes generally involves one-off analyses based on a specific middlebox classifier (or suspected censorship trigger). Further, these schemes face deployment hurdles as they often require both client- and server-side changes. For example, the mobile app Signal recently deployed domain fronting [24] to avoid blocking in Egypt and UAE [36]. This required users to update their apps to evade censorship, with the obvious caveat that the app store might also be censored in the region. In contrast, a network provider can often neutralize a statically defined evasion technique once it is known (e.g., by blocking a fronted domain). To resolve this imbalance, there is a need for generalizable, adaptive, and non-invasive techniques to evade classifiers.

This paper presents the design and implementation of ***lib·erate***, a tool for identifying middlebox policies, reverse-engineering their implementations, and adaptively deploying custom circumvention techniques. Our key insight is that differentiation is necessarily implemented by middleboxes using *incomplete* models of end-to-end

communication at the network and transport layers. These models are necessarily incomplete because a middlebox does not have visibility into the state at endpoints; namely, a packet that traverses a middlebox may never reach the intended destination (and/or the corresponding application at the destination). Further, in practice we find that middlebox designers do not completely implement standards governing network traffic, nor do they necessarily inspect every packet of every flow (potentially for efficiency/cost reasons). With this insight, *lib·erate* conducts efficient, targeted network measurements to identify any network- and transport-layer inconsistencies between middlebox- and endpoint-views of end-to-end communication, and leverages this information to transform arbitrary network traffic such that it is *purposefully* misclassified (e.g., to avoid shaping or censorship).

At a high level, *lib·erate* operates in four automated phases. First, it conducts tests to determine whether an application's network traffic is differentiated by a middlebox (e.g., throttled, blocked). If so, it identifies which features of that network traffic that the middlebox uses to classify the application for differentiated treatment. Next, it uses this information to test candidate evasion techniques that are designed to exploit inconsistencies between the middlebox and end-to-end view of network traffic. Finally, *lib·erate* uses the lowest cost, working technique to evade differentiation for network traffic generated by the application.

Our approach is application agnostic, can be deployed unilaterally (i.e., only at one endpoint) on unmodified applications via a linked library or transparent proxy, and can adapt to changes to classifiers at runtime. Like any evasion technique, *lib·erate* does not end the cat-and-mouse game between evasion and countermeasures; rather, by automating evasion and adapting to changes in middlebox classifiers quickly, it makes countermeasures substantially more expensive for network providers.

The primary contributions of this paper are:

- An application-agnostic approach to identifying traffic-classification rules for differentiation.
- A taxonomy of evasion techniques that exploit inconsistencies between end-to-end and middlebox views of network flows.
- A library that identifies classification rules, finds the network location of the corresponding middlebox, and deploys low-cost, custom countermeasures without modifying applications.
- Public, open-source tools and datasets to allow others to incorporate and extend our work.[1]
- An empirical measurement of traffic classifiers and their susceptibility to evasion techniques.

We evaluate *lib·erate* in a testbed environment and in a variety of operational networks that throttle or block traffic using DPI-based classifier rules, and show that *lib·erate* is effective across a wide range of middlebox deployments (our testbed device, T-Mobile's Binge On middlebox, AT&T's Stream Saver middlebox, Sprint's unlimited data plans[2], the Great Firewall of China, and Iran's censor).[3] We further use *lib·erate* to characterize middlebox policies and implementations in these operational networks. Our key findings are as follows.

- Policies are implemented using classifiers that rely on searches for keywords in HTTP payloads, SNI fields, and protocol-specific fields. Some of the policies apply only to a small number of initial packets, while Iran's censoring devices inspect the entire flow. We found no evidence that UDP traffic was classified by any of the operational networks we tested, providing a surprisingly easy way to evade their policies (i.e., use UDP-based protocols).
- Middleboxes running the classifiers exhibit different, incomplete implementations of network and transport layers. Specifically, our testbed device does not check for a wide range of invalid packet header values, while the Great Firewall of China (GFC) does extensive packet validation. Iran and T-Mobile use middleboxes that only partially check for invalid packet headers. Further, we find that reordering of TCP segments can alter classification in all instances except for the GFC and AT&T (the latter uses a transparent HTTP proxy).
- Except for AT&T and Iran, all middleboxes in our experiments are vulnerable to misclassification using TTL-limited traffic that reaches the middlebox but not the server.
- The classifiers for Iran and AT&T only inspect port 80 for matching content; thus simply changing the server port can evade classification.
- Classifier results do not persist indefinitely, meaning that establishing a connection and pausing can evade middlebox policies. For some devices, flow contents are ignored after a fixed delay.
- If we can assume server-side support as well, we found that inserting even one packet carrying dummy traffic (that is ignored by the server) at the beginning of a flow evades classification in our testbed, T-Mobile, AT&T, and the GFC.

The paper is organized as follows. The next section discusses related work in the area of middlebox implementations and evasion. In Section 3 we detail the goals and approach taken in this work. Section 4 presents the design of *lib·erate* and Section 5 provides several key implementation details. We evaluate *lib·erate* in Section 6, both in terms of effectiveness at identifying policies and efficiency at evading the classifiers that implement them. We discuss open questions and future work in Section 7, and we conclude in Section 8.

## 2 RELATED WORK

We now describe related efforts that focus on identifying middlebox policies and classifiers, and developing strategies to evade them. Table 1 summarizes how *lib·erate* compares with related work on middlebox evasion.

**Identifying traffic-shaping middleboxes.** A substantial body of research focuses on identifying when ISPs implement policies that provide different service to different applications [43], and how these policies are implemented. Historically, previous work found that port-based differentiation led to BitTorrent shaping and blocking [20]. NetPolice [51], Bonafide [13] and NANO [44] use statistical analysis over different Internet paths to identify and isolate the location where traffic differentiation occurs. Likewise, Tracebox [18] reveals the existence of traffic-modifying middleboxes along a path, but does not focus on classification. Our work [32] used a VPN

[1] http://dd.meddle.mobi/liberate.html

[2] Sprint specifies that their unlimited plan includes "mobile optimized video, gaming & music streaming"

[3] An analysis of Verizon's DPI classifier appears in [15].

proxy to identify when networks selectively provide differential service (often shaping/throttling) for certain applications. In closely related work, we [35] develop techniques that reverse engineer modern DPI devices and find that they focus on keywords in HTTP headers and TLS handshakes.

**Network intrusion detection systems (NIDS).** NIDS devices seek to identify and block traffic that is potentially harmful to hosts in a network. A series of studies [34, 39, 40] identify network- and transport-layer techniques to evade NIDS devices, and countermeasures that defeat them. Interestingly, we find that several of their proposed defenses are *not* deployed by the middleboxes we studied. In addition, we developed several new evasion techniques not previously explored in prior work. As we discuss in Section 4, we leverage inferred information about how application-specific traffic classifiers are implemented, which admits new evasion techniques not previously considered in the NIDS context.

**Measuring Internet censorship.** Recent measurement research on censorship tends to focus on a single country (e.g., China [9–11, 16, 17, 38, 41, 45, 50, 52], or Iran [7, 8, 12]) or censorship technique (e.g., TCP RSTs [16, 17, 47, 50], or DNS filtering [10, 11, 21, 45]). Generalizing measurement tools and techniques between countries is challenging because they are tied to the specifics of how censorship is implemented.

**Evading (censorship) classifiers.** As online content controls and Internet censorship have evolved, so have the methods and techniques used to circumvent them. We briefly describe several key strategies below.

*VPNs and proxies.* These approaches avoid censorship by using proxies to access content on behalf of the user. These range from use of HTTP proxies and virtual private networks (VPNs) [3] to tunneling web browsing over Tor [19]. We found that using VPN proxies often prevented classifiers from providing differential service to video streaming traffic [32]. However, there is an active arms race between providers of proxies like Tor and censors who attempt to detect and block them.

*Covert channels.* These techniques modify censored traffic to resemble "innocuous" traffic by imitating or piggy-backing on other application-layer protocols [26, 29, 37]. This method has been used to hide Tor traffic from censors that classify and block Tor through protocol fingerprinting techniques [5]. However, recent work shows that many of these techniques can be identified based on discrepancies from the cover protocol [25, 28].

*Obfuscating traffic.* This circumvention approach avoids classification by "looking like nothing" (i.e., randomizing content and other properties of the protocol). ScrambleSuit [49] and obfs4 [48] use this technique to circumvent blocking of Tor. These techniques, however, rely on censors employing a black-listing approach where they only shape classified traffic. In practice, networks may apply a default differentiation policy for "unclassified" traffic (e.g., limited bandwidth [32] or disrupting connections), under the assumption that such traffic corresponds to malware, attacks, errors, or evasion.

*Domain fronting.* Domain fronting (e.g., meek [24]) attempts to evade classification by proxying targeted traffic using servers running existing popular services (e.g., exchanging Signal chat messages over Google servers [36]). It relies on the assumption that censors will be reluctant to block access to popular services. However, this assumption does not always hold, particularly in repressive regimes. For example, several fronting domains have been blocked in China [6] and the list continues to grow [2].

*Censorship fuzzing.* Khattak et al. [33] make the observation that censorship monitors use similar principles as network intrusion detection systems (NIDS), and leverage this to identify gaps in the Great Firewall of China (GFC) by conducting extensive probing. However, to the best of our knowledge the evasion techniques they identified require extensive client support and makes certain assumptions about the server, such as how it processes overlapping fragments and segments. In contrast, we focus only on unilaterally deployable evasion techniques that work on unmodified applications. A concurrently published study [46] investigated in detail how the GFC maintains state for network connections, and how this can be exploited to evade censorship. The authors find additional effective evasion techniques for the GFC, but their approach is not automated and it is unclear to what extent it generalizes beyond the GFC.

**Summary.** The process of detecting, reverse-engineering, and evading traffic classifiers has faced an uphill battle. Many tools require significant development effort, but are easily thwarted once detected by the censor. Some tools trade performance for additional stealthiness (e.g., Castle [26] provides 50-400 Bps of throughput) which makes them inappropriate for popular bandwidth-intensive services. Further, solutions that rely on infrastructure (e.g., Tor relays, meek proxies) require maintenance by a third party. The problem is further complicated by solutions that require software to be installed on both the client- and server-side of the connection or the use of a proxy to mediate interactions with existing servers. Thus, there is a need for an approach that automatically identifies when policies impact network traffic, determines how traffic is classified to be subject to these policies, and deploys circumvention strategies to evade them. We describe how *lib·erate* achieves this in the next section.

## 3 GOALS AND APPROACH

This section describes our goals and approach in the design and implementation of *lib·erate*.

### 3.1 Goals

Our primary goal is to provide a tool that enables *unmodified* network applications to *automatically, adaptively, and unilaterally* evade middleboxes that perform unwanted DPI-based differentiation. Unwanted differentiation can include throttling (which is prohibited in the U.S. at the time of writing [23], but still occurs) or content blocking [30]. *lib·erate* is designed as both a library that can be wrapped around existing socket libraries or as a local proxy service to lower the barrier to adoption. Further, *lib·erate* can be deployed unilaterally by the client or server (or deployed on both endpoints). *lib·erate* automatically identifies differentiation and determines the policy used to classify the flow. Based on this, *lib·erate*

| Method | Overhead per flow (n packets) | Client only | Application agnostic | Classifier evasion | | | | Validation in the wild |
|---|---|---|---|---|---|---|---|---|
| | | | | Rule detection | Splitting/ Reordering | Inert packet injection | Flushing | |
| VPN | $O(n)$ | × | ✓ | × | × | × | × | N/A |
| Covert channels[26, 29, 37] | $O(n)$ | × | × | × | × | × | × | × |
| Obfuscation[48] | $O(n)$ | × | × | × | × | × | × | ✓ |
| Domain fronting[24] | $O(1)$ | × | × | × | × | × | × | ✓ |
| C. Kreibich et al. [34] | $O(1)$ | ✓ | ✓ | × | × | ✓ | × | × |
| *lib·erate* | $O(1)$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

**Table 1: Comparison between *lib·erate* and other classifier evasion methods and studies.**

applies evasion techniques, and adapts them as needed when the classifier policy changes. Finally, our approach is designed to have sufficiently low overhead so as not to negatively impact end-to-end performance during evasion.

**Assumptions.** We make the following assumptions in the design and implementation of our system. First, we assume that differentiation is detectable, and is implemented on a middlebox that does not have access to client-side or server-side connection state. Substantial prior work details how to detect practices such as shaping, throttling, content modification, and blocking [20, 32, 51]; further, these practices are well known to be deployed on middleboxes that are not co-resident with clients and servers. We assume that these middleboxes use traffic classifiers to identify flows that receive differential treatment, so any differentiation that is not based on contents of network traffic (e.g., differentiation based on the client's IP address) is out of scope.

Our approach focuses on applications that use TCP or UDP on top of IP. We further assume that communication is between exactly two endpoints and at least one endpoint runs *lib·erate*.

**Non-goals.** We do not attempt to hide our differentiation/evasion tests, replay servers, or evasion techniques, and as such they can potentially be detected by the network implementing differentiation. However, by having an arsenal of evasion techniques we can quickly adapt to countermeasures. Our previous work discusses strategies for detecting adversarial treatment of test traffic and replay servers [32]. Further, we do not claim to provide any anonymity or privacy for activities where *lib·erate* helps to evade classifiers. *lib·erate* does not attempt to detect stealthy or randomized differentiation. Finally, we recognize that *lib·erate* can be used for both good (e.g., freedom of speech) or bad (e.g., evading NIDS) purposes, and do not condone (nor can we control) any use that might lead to harm.

## 3.2 Approach

We take an empirical approach to understanding and evading middlebox policies, using a systematic analysis of network and transport protocols to provide a general, effective, and efficient implementation. Our system entails the following steps.

**Differentiation detection.** We use general techniques for detecting differentiation in the form of shaping/policing, content modification, and/or blocking.

**Characterization.** We next reverse engineer the classifier rules that implement it (Section 4.2).

**Evasion evaluation.** We develop a taxonomy of application-agnostic evasion techniques that leverage a mismatch between a classifier's view of the flow state and the view from the end hosts (Section 4.3).

We also build a system that automatically deploys evasion techniques based on observed classifiers, both when an application first runs in a network and subsequently when any classification rules or policies change (Section 5).

**Evasion deployment.** To ascertain the effectiveness of our approach, we evaluate it both in a testbed environment equipped with a carrier-grade DPI traffic shaper, and against several differentiating middleboxes in operational networks (Section 6).

# 4 *LIB·ERATE* DESIGN

We now discuss the design of *lib·erate*. At a high level, our design entails two phases: (1) understanding the policies and traffic-classification rules employed by middleboxes in a network (if any), and (2) developing and applying application-agnostic techniques to evade the classifier. For (1), we extend previous work to provide a more general approach to detecting differentiating middlebox classification rules. For (2), we build a taxonomy of potential evasion techniques for TCP, UDP, and general IP traffic, broadly categorized as *inert packet insertion*, *payload splitting/reordering*, and *classification flushing*. These approaches leverage the fact that middleboxes have an incomplete view of end-to-end traffic, and may even have an incomplete implementation of transport and network layers. Further, by operating at the network and transport layers, our approach can be implemented as a networking library or proxy without needing application modifications.

Figure 1 presents an overview of the key components in *lib·erate*. First, we replay modified traces of application traffic to detect if differentiation occurs. If it does, we conduct a detailed, automated diagnosis of the specific traffic-classification matching fields that select the application for differentiation. After identifying the matching fields, we select from a suite of classifier-evasion techniques and iteratively try them until one succeeds or we have exhausted our approaches.[4] Finally, we use the most efficient, successful evasion technique for the application at runtime. Note that the first three steps require interacting with a replay server, but the last step (evasion) does not. Further, the result of the first three steps can be reused for subsequent flows from the classified application (until the classifier changes in a way that breaks evasion); thus, these steps are rarely executed in practice and represent a one-time cost for the lifetime of a classifier rule.

## 4.1 Differentiation Detection

For the initial phase of *lib·erate*, as depicted on the left of Fig. 1, *lib·erate* detects whether a network on the path between endpoints applies differentiation based on contents of an application flow. For

[4] In this study, we try all possible techniques to find successful ones.

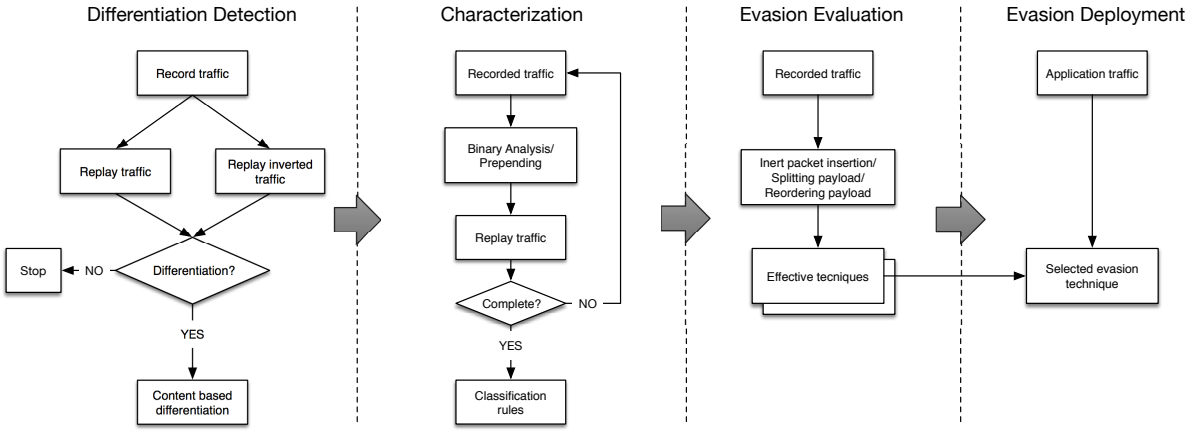



**Figure 1: Overview of *lib·erate*. In the first step, *lib·erate* detects DPI-based differentiation. If there is differentiation, *lib·erate* characterizes the classifier that implements it and tests evasion techniques based on the results. Last, *lib·erate* deploys an efficient evasion technique for applications at runtime.**

this step we use our previous work [31, 32], which detects differentiation in terms of bandwidth limitations, latency differences, content modification, blocking, and zero-rating.

This previous approach uses VPN proxies and randomized packet payloads to avoid classification, for comparison with unmodified application traffic that is exposed to DPI classification rules. However, in practice running VPN clients and servers across platforms is not easily deployable or scalable. Further, we find that randomized packet payloads are sometimes accidentally classified as a targeted application, leading to differentiation on traffic that was assumed not be exposed.

Thus, we validate the existence of deep-packet inspection (DPI) classifiers by sending traffic payloads with bits inverted from their original recorded payloads. This ensures that any bit patterns used for classification are systematically removed from the payload.

## 4.2 Characterization

After detecting differentiation, we use the binary analysis technique proposed in prior work [35] to identify the traffic-classification rules employed by the detected middlebox. Briefly, this technique conducts a recursive, binary search on payload contents to determine which bytes are responsible for classification, by "blinding" portions of the payload (i.e., inverting the bits) to prevent them from triggering classification. The end result is a list of packets and regions of payload that triggered classification ("Characterization" in Fig. 1). We call these portions of payloads *matching fields* used by classifier rules.

We extend this previous work in the following ways. First, we adapt the technique to additionally detect UDP-based classification rules. Second, we develop a new strategy to efficiently detect whether "match-and-forget" rules apply.[5] Specifically, we prepend an increasing number of packets with random payloads before a packet that triggered classification; if these flows always experience differentiation then we have high confidence that the classifier matches on all packets in a flow. On the other hand, if we detect that $i$ packets contain matching fields and prepending $j$ packets changes the classification result, we can confirm that the classifier checks at most $(i + j - 1)$ packets for matching fields.

**Characterization efficiency.** Even with the optimization described in the previous paragraphs, characterization may still require tens of minutes to complete. While we can reduce the number of tests (and thus the time to complete), there is a trade-off between time and accuracy.

An alternative approach to reduce runtimes is to distribute disjoint subsets of the tests among multiple users in the same network, and aggregate the results. These test results can be stored in a well known public location (e.g., a server or a DHT) so that all users can identify the matching rules without running additional tests. However, the drawback of this approach is that an adversary could see the detected rules and adapt accordingly.

*lib·erate* must run the characterization step whenever an application's classification rule changes. In our experience, these rules change relatively infrequently compared to the time required to characterize classification rules, and they change only in response to applications using different protocols (e.g., HTTPS instead of HTTP) or different domain names and/or content delivery providers. To detect a change in classification rule, *lib·erate* repeats the test for differentiation. If differentiation occurs even when using a previously successful evasion technique, then *lib·erate* assumes that matching rules have changed, and repeats the characterization and evasion steps.

**Characterization countermeasures.** An adversary wishing to evade our characterization step could selectively choose not to differentiate against our replay servers or our tests. We can detect the former using previously unseen replay servers. If countermeasures that rely on identifying our servers do not work, it will be difficult to differentiate against our tests without collateral damage. For example, an adversary could attempt to frustrate characterization tests by changing classification rules during the analysis. Doing so, however, requires differentiating traffic that does not match targeted rules, e.g., acting on traffic containing "net" instead of "netflix." Such an approach could be costly for the network if

[5] Previously, we found that most classifiers made final classification decisions within a small number of packets [35] (e.g., the first one or two); however, in this study we found two middleboxes that do not.

it affects legitimate, popular traffic (e.g., all traffic containing the term "net").

### 4.3 Evasion Evaluation

In the second phase of our design, we apply a suite of classifier-evasion techniques to identify those that prevent the classifier from applying an unwanted policy ("Evasion Evaluation" in Fig. 1). This section describes how we generated this suite of techniques by creating a taxonomy of transport- and network-layer modifications to an application's network traffic.

Recall that we assume that a classifier identifies applications based on the payloads of network traffic. Figure 2(a) depicts the packet/time diagram of communication between a client and server that traverses a middlebox classifier. The boxes on the left indicate the type of packet being exchanged, while the shaded boxes under "Classifier" indicate what the classifier has detected at each point in time. In this case, there is no attempt at evasion and the classifier correctly classifies the flow as HTTP traffic for application *B*.

**Strawman: payload modification.** We begin with a simple strawman and explain why it is not sufficient for our purposes. Namely, we could leverage the specific classifier rules identified in the previous phase to avoid classification by removing any matching content from an application's flow (e.g., by deleting them entirely or replacing them with random bytes). While this is likely to evade classification, it will also potentially break end-to-end functionality by generating traffic that the other endpoint does not know how to interpret. For example, if a classifier matches on the contents of the HTTP `Host` header and that field is replaced with random bytes, a server hosting multiple domains would not be able to tell which one should handle the request.

**Key insight.** Previous work showed that middlebox classifiers use incomplete views of end-to-end communication at the network-, transport-, and application-layers to match on rules, e.g., by inspecting packets with errors [34], or by looking at a small number of packets [31, 34] or HTTP/TLS fields [31]. Our key insight is that we can leverage knowledge of classifier implementations to evade them in new and systematic ways.

To this end, we developed a taxonomy of classifier evasion techniques that fall into four high-level categories: sending traffic that is processed by a classifier but never reaches the application-layer endpoint (*inert packet insertion*), modifying the size and order in which portions of payload are revealed to the classifier (*payload splitting and reordering*), and causing the middlebox to flush the classification state for the flow (*classification flushing*).

**Inert packet insertion.** Figures 2(b) and 2(c) illustrate the inert packet insertion technique for a flow belonging to traffic class *B*. After completing a traditional three-way handshake, in Fig. 2(b) *lib·erate* generates a valid application-layer request for traffic class *A* that is wrapped in a TCP/IP packet that is rejected by the server (in this case due to an invalid IP protocol value). Similarly, in Fig. 2(c) *lib·erate* generates a valid packet for traffic class *A* with a TTL that expires before it reaches the server. In both cases, the classifier is not aware that the packet is either not delivered to, or rejected by, the server, and classifies the flow into class *A*. Assuming a "match and forget" classifier, subsequent traffic for *B* will be treated the same as class *A*.

We exhaustively analyzed TCP, UDP, and IP packet fields for opportunities to create inert packets, which produced a set of techniques listed in Table 3 (leftmost columns, upper portion). Most of the techniques rely on setting certain bits to invalid value, while the techniques using header options are motivated by the observation that middleboxes may treat header options differently [27] than end hosts. While we believe our list is comprehensive, it will likely grow as new protocols become available. Instead of relying on completeness, our approach is designed to leverage any additional techniques should they arise.

**Splitting and/or reordering payload.** Figures 2(d) and 2(e) illustrate the payload splitting and reordering techniques, respectively, for evading classification. Here, we exploit cases where the middlebox relies on a fixed number of packets to classify, or where it has an incomplete implementation of the TCP/IP stack that improperly handles segments and/or fragments. Unlike the previous technique, all packets that arrive at the server are valid and payloads are delivered to the application-layer endpoint.

In Fig. 2(d), *lib·erate* modifies the original flow by splitting TCP segments across $k$ IP packets, evading a classifier that inspects only the first $k-1$ (or fewer) packets. In Fig. 2(e) *lib·erate* reorders the transmission of segments, evading a classifier that makes decisions based on matching fields in the first $k$ packets.

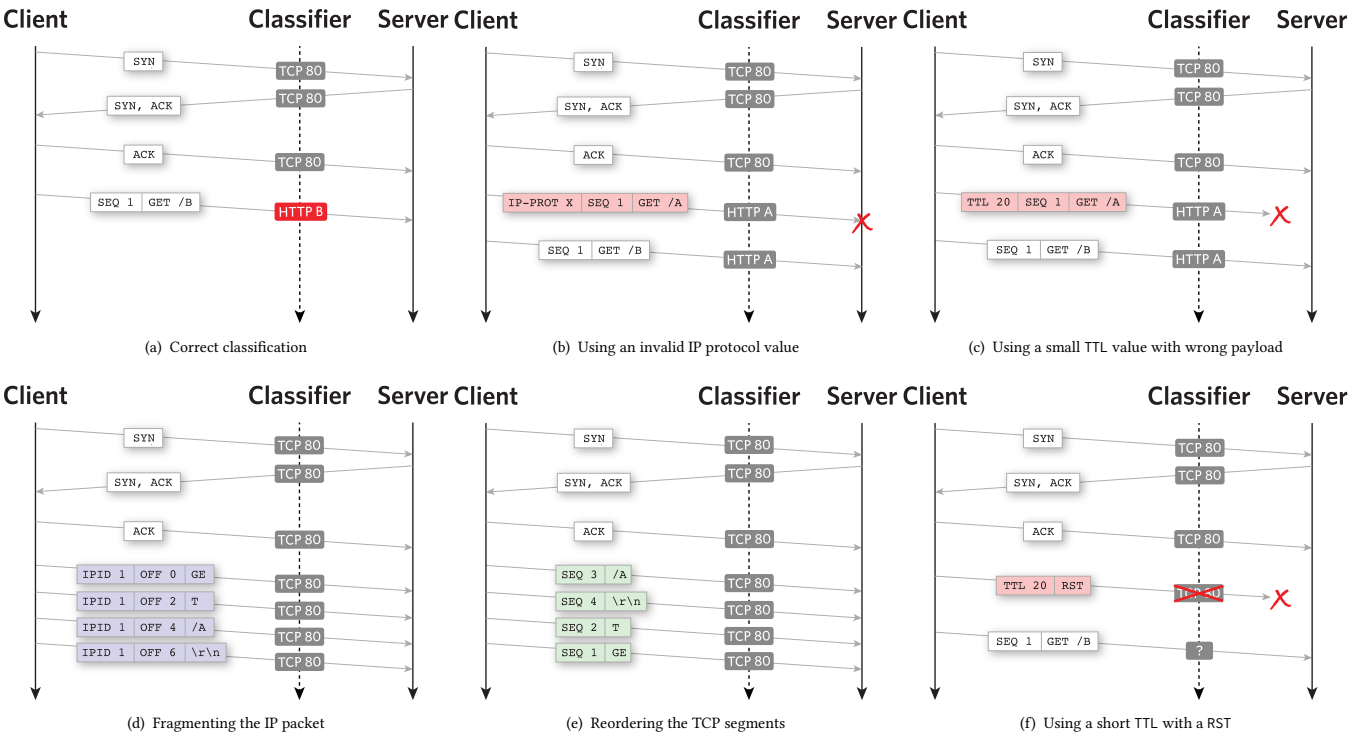
We identified five techniques that use splitting/reordering to evade classification, listed in Table 3 (leftmost columns, lower portion). Again, we believe this list is comprehensive; regardless, *lib·erate* can incorporate additional ordering techniques for new protocols as they are discovered.

**Classification flushing.** We found empirically that several middleboxes do not retain their classification results indefinitely; namely, these results may be flushed after certain packet-based events or timeouts. We investigate this (out of band) by inserting increasingly large delays before matching packets (which may timeout faster), between matching packets and subsequent packets, or by sending inert RST/FIN packets (Fig. 2(f)). We additionally investigate the impact of using this technique at different times of day (where resources to maintain state may become more scarce during busy hours).

**Comparison with NIDS attacks.** In their 2001 paper, Kreibich et al. [34] describe a suite of attacks that evade NIDS devices along with an approach called *norm* that normalizes attack traffic to neutralize it. Like their work, we also propose techniques that use invalid transport- and network-layer packets to confuse middleboxes. Interestingly, we find that few defenses identified by *norm* are adopted by the operationally deployed middleboxes that we tested. Note that our inert injection techniques are inspired by the NIDS attacks and are well known.

In addition, we propose new evasion techniques that use *valid* packets to target specific implementations of traffic classifiers (and their weaknesses). These include payload splitting, payload reordering, and classification flushing, which prior work does not address.

**Completeness.** We cannot claim that the evasion techniques presented in this paper are complete, nor that they are sufficient




(a) Correct classification (b) Using an invalid IP protocol value (c) Using a small TTL value with wrong payload

(d) Fragmenting the IP packet (e) Reordering the TCP segments (f) Using a short TTL with a RST

**Figure 2: Diagrams of example techniques that *lib·erate* uses to evade classification.**

to evade differentiation in every network. Our approach, however, can incorporate new techniques as they are discovered (e.g., state-based attacks [46]), and identify cases where middleboxes prevent evasion (e.g., a connection-terminating TCP proxy). The latter is an inherent limitation of a unilateral approach to evasion, and in future work we will investigate how bilateral control can enable evasion even in such cases.

**Evasion countermeasures.** We now discuss how a network might deploy countermeasures against our evasion techniques. For one, a network could detect and filter *lib·erate*'s inert packets, much like Kreibich et al. proposed [34]. Doing so would render this class of techniques ineffective.

Countermeasures for splitting/reordering packets, as well as classification flushing, require a middlebox to reassemble packets and maintain state for longer durations than is done today. While it is feasible to do so, we believe it is expensive (as it requires more state and processing) and that engineering such solutions will become only more costly as connection volumes continue to increase.

Our TTL-limited technique can be defeated if the middlebox normalizes the TTL to a large value or if it knows how many hops remain to the destination. However, the former could have unintended side-effects (e.g., amplifying load during transient loops), and the latter is difficult to determine accurately due to dynamics of routing.

In summary, all of our evasion techniques are susceptible to countermeasures and we believe this to be intrinsic to unilateral evasion techniques. As part of future work, we are exploring low-cost bilateral evasion techniques that are significantly less susceptible to countermeasures. In the meantime, we find that today's DPI devices are all susceptible to multiple evasion strategies.

### 4.4 Evasion Deployment

After identifying effective techniques for an application, *lib·erate* deploys the most efficient, successful evasion technique. Namely, *lib·erate* (which can be configured as a proxy or library built into an application) intercepts the application's network traffic and modifies it using the selected evasion technique.

## 5 IMPLEMENTATION

We built a prototype *lib·erate* system as shown in Figure 3. The system works in three phases; in the first phase, application-generated traffic exchanged between the application's client and server is recorded for controlled tests.[6] In the second phase, *lib·erate* uses the recorded traffic with a replay server to identify differentiation, reveal classification rules, and evade classification (as described in Section 4). The replay server is deployed such that the path from the client traverses the classifier; in some cases it can even be deployed in the same data center as the application server. The last phase entails modifying application-generated traffic in-flight using one of the evasion techniques identified in the previous steps. The *lib·erate* system is implemented in Python, currently with 4,800 lines of code (2,500 of which come from the system developed in [32]).

[6] Alternatively, we can provide built-in traces that are distributed with the tool.

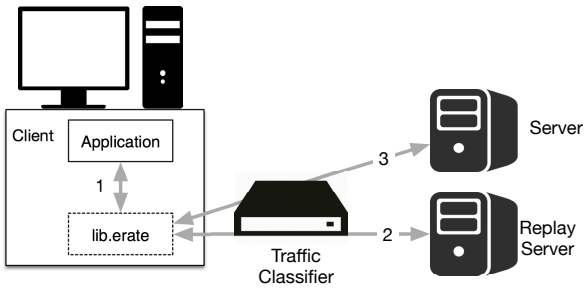



**Figure 3: Diagram of *lib·erate* implementation. An unmodified client application generates network traffic that is recorded for classifier identification and evasion (1). *lib·erate* then uses this traffic to conduct controlled experiments that identify classification rules and evasion techniques (2). After identifying effective techniques, *lib·erate* applies the most efficient working technique to live application-generated traffic (3).**

## 5.1 Identifying policies and classifier fields

We use previous work [35] to identify the existence of differentiation. A key difference from the previous approach is that we invert each bit instead of using a VPN proxy or randomized payloads as the "control" traffic that should not be subject to differentiation. We do this because the bit inversion operation ensures that the replay is completely different from the recorded trace at the bit level. Second, it is deterministic, which allows us to identify portions of payloads that match classification rules without interference from random bytes that might also match classification rules.[7]

We also extend our previous work [35] to identify not only matching rules on content, but also ones that depend on the position of the matching payload within the flow. We do so by inserting random payloads before each matching field, both ones that are in the same packets as a matching field, and ones that add one or more packets before the matching field. Our hypothesis (based on our previous work [35]) is that some classifiers inspect a fixed number of packets or bytes in a flow before reaching a final classification decision. To test this hypothesis, we first append random bytes in increments of one MTU until we observe a change in classification. Assuming this change occurs after $k$ packets were sent, then we append $k$ 1-byte packets to the flow instead of $k$ MTU-sized packets to see if the change in classification depends only on packet count instead of bytes. If so, we conclude there is a fixed packet-based limit; else, we conclude that the limit is no more than $k * MTU$ bytes. To avoid iterating forever, we use a tunable maximum threshold of packets (based on our observations, 10) to prepend before concluding that the classifier will inspect all packets in a flow.

## 5.2 Evading classification

To implement our evasion approaches, we needed to address the following key challenges.

**Determining the middlebox location.** With TTL-limited packets (first and last rows in Table 3), we send a valid packet that traverses the middlebox but times out before reaching the server. This requires *lib·erate* to first identify where the middlebox is located (in terms of number of TLL-decrementing hops) relative to the host generating the packet. We implement an approach similar to traceroute and Tracebox [18], using a series of probes starting with TTL=0 and incrementing the TTL on subsequent probes until we observe a response indicating that the TTL-limited flow was classified. In the case of censorship, the inert packet often generates a response from the middlebox such as a block page or RST packet. To detect shaping and zero rating, we use the approaches in our previous work [32, 35].

**Do invalid inert packets reach the middlebox?** Most of our inert packet insertion techniques rely on sending packets that are not compliant with TCP/IP protocols and thus may (but not must) be dropped by any intermediate hop between endpoints. To test whether packets are dropped in the network, we send inert packets to our *lib·erate* replay server: if they reach the server, then we know that they also reached the middlebox. In the case where they do not reach our replay server, they may still have traversed the middlebox before being dropped. In this case, we can test whether any subsequent (valid) packets in the flow experience differentiation. If so, then the middlebox classified traffic based on the inert packet. Otherwise, the inert packet is either ignored by the middlebox or never reaches it.

**Determining how to split/reorder payloads.** The number of possible ways to split and reorder payloads grows combinatorially with the size of the flow being tested. Thus, it is infeasible to test all combinations, so we use a heuristic approach. First, we split the packet into at most $n$ packets, where any matching fields are split across packet boundaries. We currently use a conservative threshold of $n = 10$ based on our empirical observations that packet-limited classifiers inspected no more than 5 packets. Likewise, we test whether classifiers support IP fragmentation and reassembly by splitting each packet into $m$ fragments (currently $m = 2$). Our values of $n$ and $m$ were chosen empirically to improve *lib·erate* efficiency; we can further adapt/tune this parameter based on empirical observations in deployment.

For reordering packets and fragments, we explore all combinations of order starting with *reversing* the initial $n$-packet, and stop when we find a reordering that evades classification. In our experiments, we found that this reveals effective packet-order evasion techniques after just one try (when focusing only on those techniques that use packet-limited classifiers).

**Efficient evasion testing.** Our taxonomy of evasion techniques allows us to efficiently rule out tests based on observed classification behavior. For example, if *lib·erate* finds that a classifier inspects *all* packets instead of the first $n$ packets, then we know that inert packet insertions are unlikely to evade the classifier and we can eliminate those tests from our suite. Here, *lib·erate* uses only reordering tests in this case. When *lib·erate* identifies "match and forget" classifiers, it instead tests the more efficient inert packet insertion techniques first. When deciding the order in which to test techniques that have not been ruled out, *lib·erate* tests evasion techniques that were effective in our study first, based on the assumption that such classifier implementations are also deployed elsewhere.

[7] This approach can be detected by middleboxes, so we fall back to randomization if bit inversion fails to reveal correct matching rules.

| Technique | Description | Overhead |
|---|---|---|
| Inert packet insertion | Inject packet that either does not reach the server, or reaches but is dropped. | $k$ packets |
| Payload splitting | Divide a flow's payload into packets of different sizes from the original. | $k$*40 bytes[1] |
| Payload reordering | Reorder packets relative to the original flow. | $k$*40 bytes[1] |
| Classification flushing | Cause a classifier to flush its classification result, leaving a flow unclassified. | $t$ seconds or 1 packet[2] |

**Table 2: High-level evasion techniques in *lib·erate*.** [1]**Plus nominal overhead for reassembling.** [2]**When flushing with inert RST.**

## 5.3 Performance of *lib·erate*

The primary overheads of *lib·erate* are the characterization of the classifier for a specific application, and the techniques used to evade it. The (relatively expensive) characterization step need only be conducted once for each classifier rule, and can be skipped as long as the rule remains in place. In our experiments, this step ranges from 10–35 minutes and 300 KB (Web pages) to 140 MB (video streams) of data. This cost in general should be incurred rarely, and the results from one user in a network can be shared with another.

The remaining cost comes from the evasion technique deployed when running an application. These costs are relatively low, both in terms of packets and delay (Table 2). When performing injection, splitting or reordering, *lib·erate* overhead consists of $k$ extra inert packets (up to 5 MTU-sized packets) or adding a nominal amount of processing time for the server to reassemble the $k$ segments. In practice, we find that $k$ is always less than 5; when using *lib·erate* on a video streaming connection, this can represent small fractions of a percent of data overhead. When using *lib·erate* on test applications, we did not notice any qualitative negative impact on performance.

When using the classification flushing approach, the overhead is either sending one inert RST packet or waiting for $t$ seconds. The effects of the former are nearly immediate, while for the latter we found empirically that $t$ ranges from 40 seconds to 240 seconds (for the GFC, see §6.5).

# 6 EVALUATION

We evaluate our approach both in a testbed environment and "in the wild" in operational networks known to use classifiers to impose policies on network traffic. In our testbed environment, which contains a carrier-grade DPI device, we can evaluate our approaches and directly view their impact on classification. We use this to evaluate the efficiency of *lib·erate* for each of its phases, and to characterize the DPI device itself and the effectiveness of our evasion techniques. The experiments in operational networks demonstrate that our approach works well outside our testbed, and that it can reveal effective evasion techniques previously unknown publicly.

## 6.1 Testbed experiments

Our testbed consists of a client connected to a server via a DPI middlebox from a top-5 industry leading DPI middlebox vendor and router. We record popular TCP and UDP content that is classified by the DPI device, then use these traces to replay traffic with and without *lib·erate*. The middlebox shows the result of classification immediately after traffic flows through it.

For each application, we evaluate the following. First, we determine how many rounds of measurement are required to reverse engineer the classifier for the application, and use this information to calculate the runtime and data consumption for these tests. Next, we evaluate the effectiveness of every evasion technique and the efficiency for identifying them. Last, we determine how long classification of flows persists on the DPI device. We omit any tests for identifying differentiation, since this was covered in previous work [32].

**Efficiency of classifier analysis.** Identifying the matching rules used for classification requires repeated rounds where different portions of replay traffic are blinded via bit inversion. We use our testbed (with ground truth classification results) to determine the number of rounds, the time required to identify all matching fields, and cost in terms of bytes. We focus on two categories: HTTP traffic (which covers several streaming applications including Amazon Prime Video, Spotify, ESPN streaming video) and UDP traffic.

For HTTP traffic in our experiments, *lib·erate* needs at most 70 replay rounds to identify all the matching fields. This takes no more than 10 minutes (given a 5-second test time) and could be hastened by using multiple parallel connections. For UDP traffic, we focus on Skype. We find that *lib·erate* can successfully identify all matching fields in the first six packets, using 115 replays. In our testbed, we needed less than 2 KB of data for each replay round for both TCP and UDP traffic, since we can check the classification results immediately. We discuss the costs for each operational network in each case study in the next sections.

**Classifier matching fields.** Much like what we found previously [35], the matching fields in HTTP/S traffic typically contain human-readable text such as hostnames, content type, and application names in user agent strings. Unlike previous work, we also identify UDP-based matching rules. For Skype, we find that it uses bytes that are not human-readable. After manual analysis of the matching rules that *lib·erate* identified, we found that the classifier focused on STUN packets and used matching fields that correspond to standard STUN message types and attributes. In this case, it focused on identifying the attribute `MS-SERVICE-QUALITY`[8] (i.e., `0x8055` as the attribute type) in the first packet from the client. We also found that prepending one packet with one byte of payload changes the classification result, which suggests that the classifier only inspects packets at certain position in the flow.

**Evading classification** We now characterize how well our evasion techniques work in our testbed. We summarize our findings below, and Table 3 shows the detailed results of both inert packet insertion techniques and payload splitting/reordering. There are two columns for each test environment, where a green check mark (✓) in the left column (**CC?**) indicates that *lib·erate* can evade the classifier using the technique specified at the beginning of the row. A green check mark (✓) in the right column (**RS?**) indicates that the inserted or modified packets reach the server (i.e., we know that it traversed the classifier). The red crosses (×) indicate that classification was not changed (**CC?**) or that the packet did not reach the server (**RS?**).

These tables cover our testbed and the case studies that follow; the results for our testbed are in the group of columns labeled "Testbed." For example, the first row indicates that *lib·erate* evades

[8] The MS in the attribute name refers to Microsoft, which owns Skype.

| Prot. | Technique | Testbed CC? | Testbed RS? | T-Mobile CC? | T-Mobile RS? | China CC? | China RS? | Iran CC? | Iran RS? | AT&T — | Server Response Lin. | Server Response Mac | Server Response Win. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Inert packet insertion* | | | | | | | | | | *Dropped by OS?* | | |
| IP | Lower TTL to only reach classifier | ✓ | × | ✓ | × | ✓ | × | ×[3] | × | × | – | – | – |
| IP | Invalid Version | × | × | × | × | × | × | × | × | × | ✓ | ✓ | ✓ |
| IP | Invalid Header Length | × | × | × | × | × | × | × | × | × | ✓ | ✓ | ✓ |
| IP | Total Length longer than payload | ✓ | × | × | × | × | × | × | × | × | ✓ | ✓ | ✓ |
| IP | Total Length shorter than payload | × | × | × | × | × | × | × | × | × | ✓ | ✓ | ✓ |
| IP | Wrong Protocol | ✓[1] | ✓ | × | ✓ | × | ✓ | × | × | × | ✓ | ✓ | ✓ |
| IP | Wrong Checksum | ✓ | × | × | × | × | × | × | × | × | ✓ | ✓ | ✓ |
| IP | Invalid Options | ✓ | ✓ | ✓ | × | × | × | ×[3] | × | × | × | × | ✓ |
| IP | Deprecated Options | ✓ | ✓ | ✓ | × | × | × | ×[3] | × | × | × | × | × |
| TCP | Wrong Sequence Number | ✓ | ✓ | × | × | × | ✓ | ×[3] | × | × | ✓ | ✓ | ✓ |
| TCP | Wrong Checksum | ✓ | ✓ | × | × | ✓ | ✓[4] | ×[3] | × | × | ✓ | ✓ | ✓ |
| TCP | ACK flag not set | ✓ | × | × | × | ✓ | ✓ | ×[3] | × | × | ✓ | ✓ | ✓ |
| TCP | Invalid Data Offset | × | ✓ | × | × | × | ✓ | × | × | × | ✓ | ✓ | ✓ |
| TCP | Invalid flag combination | ✓ | ✓ | × | × | × | ✓ | ×[3] | × | × | ✓ | ✓ | ×[6] |
| UDP | Invalid Checksum | ✓ | ✓ | – | × | – | ✓ | – | ✓ | × | ✓ | ✓ | ✓ |
| UDP | Length longer than payload | ✓ | ✓ | – | × | – | × | – | ✓ | × | ✓ | ✓ | ✓ |
| UDP | Length shorter than payload | ✓ | ✓ | – | × | – | × | – | ✓ | × | ✓[5] | ✓ | ✓ |
| | *Payload splitting* | | | | | | | | | | *Delivered by OS?* | | |
| IP | Break packet into fragments | ✓ | ✓[2] | × | ✓[2] | × | ✓[2] | × | × | × | ✓ | ✓ | ✓ |
| TCP | Break packet into segments | ✓ | ✓ | ✓ | ✓ | × | ✓ | ✓ | ✓ | × | ✓ | ✓ | ✓ |
| | *Payload reordering* | | | | | | | | | | *Delivered by OS?* | | |
| IP | Fragmented packet, out-of-order | ✓ | ✓[2] | × | ✓[2] | × | ✓[2] | × | × | × | ✓ | ✓ | ✓ |
| TCP | Segmented packet, out-of-order | ✓ | ✓ | ✓ | ✓ | × | ✓ | ✓ | ✓ | × | ✓ | ✓ | ✓ |
| UDP | UDP packets out-of-order | ✓ | ✓ | – | ✓ | – | ✓ | – | ✓ | × | ✓ | ✓ | ✓ |
| | *Classification flushing* | | | | | | | | | | *Delivered by OS?* | | |
| IP | Pause for $t$ sec. (after match) | ✓ | ✓ | × | ✓ | × | ✓ | × | ✓ | × | ✓ | ✓ | ✓ |
| IP | Pause for $t$ sec. (before match) | ✓ | ✓ | × | ✓ | ✓[7] | ✓ | × | ✓ | × | ✓ | ✓ | ✓ |
| | | | | | | | | | | | *Dropped by OS?* | | |
| TCP | TTL-limited RST packet (a) | ✓ | × | ✓ | × | × | × | × | × | × | ✓ | ✓ | ✓ |
| TCP | TTL-limited RST packet (b) | ✓ | × | ✓ | × | ✓ | × | × | × | × | ✓ | ✓ | ✓ |

**Table 3: Effectiveness of *lib·erate*'s evasion techniques, showing whether the technique Changes Classification (CC?), whether the packet Reaches the Server (RS?), and whether various OSes drop or deliver the packet. For CC?, a ✓ means that the technique allows *lib·erate* to change classification, a × means that it does not. For RS?, a ✓ means that the packet that was inserted or modified does reach the server, a × means that it does not, and a ✓ means the packet that arrives at the server is different than what was sent (e.g., the IP fragments are re-assembled). For the "Server Response" columns, a ✓ means that the evasion technique does not impact the transport- or application-layer integrity; a × means that it might.**
**[1]The classifier yields different classification results for TCP and UDP. [2]The fragmented packets are reassembled before reaching the server. [3]Evasion fails, but an inert packet with blocked content causes the connection to be blocked. [4]The TCP checksum is corrected before arriving at the server. [5]The server reads the content up to the specified length. [6]The server sends a RST packet in response. [7]The interval depends on the time of the day.**

the classifier by inserting a packet with a low TTL, and the inserted packet does not reach the server.

Focusing on the **CC?** column under "Testbed," a wide range of inert packet insertion techniques successfully evade classification. The exceptions fall into a small number of categories that include specifying length fields that are too small and using invalid protocol/version values.

We find that payload splitting/reordering is always effective in our testbed. Taken to an extreme, we can evade classification even when the only modification is that the first packet contains only one byte of payload. Further, this works for both UDP and TCP.

To identify opportunities for evasion via classification flushing, we determine how long each classification result persisted in the middlebox. We found that our testbed middlebox uses a timeout value of 120 seconds for all classification results. In addition, for TCP connections, the timeout is reduced to 10 seconds after the classifier sees a RST packet for that connection.

In addition to classification results, we test whether packets are properly handled at the server without side effects, for each popular OS (Linux, MacOS, Windows). The rightmost columns in Table 3 present the results, where a green check mark (✓) indicates behavior that enables unilateral evasion. For the inert packet injection techniques, this means that the server *drops* inert packets; for payload splitting/reordering it means that the server *accepts* packets and delivers them to the application. A red cross (×) indicates a server response that might prevent an application from working due to unexpected payload.

## 6.2 T-Mobile US (TMUS)

In the US, T-Mobile sells a cellular service plan that by default includes "Binge On" and "Music Freedom," which provide zero-rated[9] video streaming and zero-rated music streaming, respectively. We ran *lib·erate* over TMUS using recorded traffic from a set of applications that are classified under one of these programs. This comprises HTTP or HTTPS traffic from YouTube[10], Amazon Prime Video and Spotify.

**Efficiency of classifier analysis.** *lib·erate* needs between 80 and 95 rounds for classification. Unlike in the testbed where we have

[9]i.e., the data from using these services does not count against the subscribers monthly data quota.
[10]YouTube flows using QUIC (an application-layer transport layer built atop UDP) are not classified or zero rated by T-Mobile, so we use only TCP flows when evaluating YouTube.

immediate access to the classification results, we use our account's data-usage counter after each replay to determine whether the traffic is zero rated (i.e., classified). The counter may either be slightly out of date, or include data from background traffic, and we found via manual analysis that using at least 200 KB of data (approximately 15 s of test time) for each replay eliminates the risk of false inference.

Altogether, our tests took 23 minutes and 18 MB of data in total. Note that these experiments can be conducted in advance and in the background (e.g., when the device is not using the network) and the results can be cached for future use in real-time evasion.

**Classifier matching fields.** Similar to our testbed findings, TMUS classifies applications by matching on text strings in certain fields such as host headers or the SNI field (e.g., cloudfront.net in the Host header, and .googlevideo.com in the SNI field during TLS handshake). For both HTTP and HTTPS traffic, prepending one packet with one byte of (dummy) data changes classification, which suggests that payload splitting or reordering would be effective evasion strategies.

**Evading classification.** We now characterize how well our evasion techniques work in T-Mobile's network (second group of columns in Table 3). The **CC?** column indicates that three inert packet insertion techniques evade classification in TMUS, one uses TTL-limited probes and the other two use invalid IP Options. For TTL-limited probes, we found that using an inert packet with TTL = 3 is sufficient to evade the classifier. For invalid IP Options, we found that while it is effective for evading the classifier, unfortunately these packets *are not* dropped by the server for most OSes. In general, such packets are not actually inert; however, we found that these invalid packets were dropped somewhere between the TMUS classifier and the server, thus enabling evasion without side effects.

The table also shows that payload splitting/reordering across multiple packets is effective, while IP fragmentation is not. Without reordering, evasion requires the payload of the matching packet to be split across five or more packets; however reversing the order in which packets are sent is always effective, even with as few as two packets. We believe that this is because the TMUS classifier reassembles TCP segments only if the first payload-carrying packet in the flow begins with `GET`, and if so, it uses only the first five packets in the flow to search for matching fields.

We surprisingly found that TMUS does not classify UDP traffic for any of the applications we tested. An implication is that YouTube traffic that uses QUIC is not throttled or zero rated.

Regarding classification flushing, we found that the classification result in TMUS applies to a flow for more than 240 s (the largest interval we tested) even if there are no packets exchanged during the interval, and the classification result is flushed immediately after the classifier sees a RST packet in the flow.

When replaying a 10 MB Amazon Prime Video trace, the average throughput is 1.48 Mbps and peak throughput is 4.8 Mbps without *lib·erate*. With *lib·erate* evasion enabled, the average throughput is 4.1 Mbps and the peak throughput is 11.2 Mbps.

## 6.3 AT&T

The AT&T Stream Saver [1] service is an opt-out program that limits streaming video to "DVD quality" without zero-rating. We ran *lib·erate* over AT&T using recorded traffic from a set of video streaming applications, including HTTPS traffic (YouTube, Amazon Video) and HTTP traffic (NBCSports). We found that replayed HTTP traffic is throttled to 1.5 Mbps, while Stream Saver did not interfere with HTTPS traffic when running our experiments. Further analysis indicates that is because AT&T runs a transparent web proxy and did not inspect TLS traffic at the time of our tests.

**Efficiency of classifier analysis.** *lib·erate* identifies the matching fields in HTTP request with 71 replays. Unlike Binge On, Stream Saver classifies video content only on port 80. The only signal for differentiation is throughput (classified video content is throttled to 1.5 Mbps), each round of test consumes around 2 MB of data and 30 seconds.

**Classifier matching fields.** The matching contents in packets from client to server are standard HTTP parameters such as 'GET', 'HTTP/1.1'. We found the server-to-client packets are also used for classification; namely, the keyword `Content-Type: video`. We also find that prepending one MTU-sized packet can break the classification.

**Evading classification.** None of the evasion techniques is effective for Stream Saver, because they deploy a transparent HTTP proxy that terminates TCP connections and thus serves as an endpoint in addition to being a middlebox. Although *lib·erate* fails to evade Stream Saver, the fact that only traffic on port 80 is differentiated in our tests makes evading it even more straightforward—by simply redirecting traffic to a different port (e.g., using TLS on 443).

## 6.4 Sprint

Sprint offers "mobile optimized video, music streaming, and gaming" as part of their unlimited plans. We tested whether this is implemented using DPI or header-space analysis, and found no evidence that it is.

Specifically, we tested different IP addresses, ports, traffic to popular video streaming services, replays of those flows to our servers, both in the original form and with bit inversion. We found no pattern to which flows received relatively more or less bandwidth. We repeated the tests on a SIM card using a limited-data plan, and saw similar results.

## 6.5 The Great Firewall of China

We now turn to the so-called Great Firewall of China (GFC), a term referring to the system that the Chinese government uses to regulate use of the Internet in Mainland China. The GFC implements policies for blocking certain applications and websites, so we use *lib·erate* to determine if this blocking can be evaded.

Our tests use a client in China and servers in the US. We record HTTP traffic from a client located in the US visiting a website that is blocked in China.[11] We then replay the trace from China using our US replay server and confirm it is blocked by 3–5 RST packets. Thus, we use the presence of spurious RST packets to indicate that traffic has been classified by the GFC.

**Efficiency of classifier analysis.** *lib·erate* identifies the matching fields in Web traffic with 86 replays, each of which consists of

[11] We use http://www.economist.com and avoid HTTPS traffic, which is typically blocked via active probing [22].

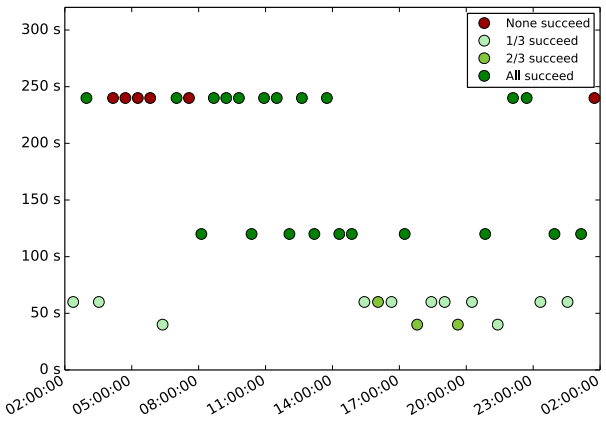



**Figure 4: Successful evasion intervals vary during the day.**

4 KB of data. These tests take less than 15 minutes and consume less than 400 KB of data. Note that detecting blocking is extremely efficient because the signal (RST packets) is clear and immediate. The keywords we identified include `GET` and `economist.com` in the Host header.

**Classifier matching fields.** Similar to other classifiers, the GFC classifier determines what to block based on text strings in the HTTP host header. In addition, we found that unlike other classifiers in our tests, the GFC blocks *all traffic toward a server, even uncensored content* after it blocks two replays for that server and port. This behavior can potentially lead to inconsistent results when determining matching fields, so in practice we use different server ports for each replay. Our subsequent analysis confirmed that doing so leads to correct results when identifying matching fields. In addition, we find that prepending a replay with one byte of (dummy) traffic causes classification to fail, similar to what we found for TMUS.

**Evading classification.** The third column group in Table 3 shows the results for our evasion techniques against the GFC. In short, using a TTL-limited probe, an invalid TCP checksum, and packets without an ACK flag set, were all effective. *lib·erate*'s TTL-probing technique found that using a TTL of 10 leads to misclassification without reaching the server when sent from our vantage point. Interestingly, prepending a replay with dummy traffic does evade a classifier, indicating that having server-side support can provide additional evasion opportunities.

Similar to TMUS, the GFC does not classify UDP traffic for any of the applications we tested. One implication is that users can view otherwise censored content on YouTube simply by using the QUIC protocol.

Sending a RST before the matching packet can evade censorship; however, unlike TMUS, sending a RST packet *after* being classified has no observable effect on classification. We also found that adding delays before the matching packet can evade classification, and the minimum delay changed over the course of the day. To quantify this, we tested delays ranging from 10 to 240 seconds and repeated tests six times per hour over the course of two days. We plot the results in Figure 4 for one of those days (the other was similar), where the $x$-axis represents time of day and the $y$-axis represents the delay interval. A green dot indicates the minimum delay that succeeded in evasion, while a red dot represents the case where even our longest delay interval did not flush the classification result.

We see time-of-day effects, where traditional busy hours permit shorter delays (likely due to classification results being flushed due to scarce resources), while during quiet hours even long delays do not work. Note that when shorter delay intervals (i.e., $\leq 60$ seconds) succeed, they typically work only for a subset of tests. To the best of our knowledge, we are the first to observe the opportunity for delay-based evasion to circumvent censorship based on faster connection-state flushing during busy hours. Note, however, that we found that adding delays *after* sending a matching GET request never evades classification.

## 6.6 Iran

Our final case study focuses on Iran, where certain websites are also blocked at a national level. Similar to the scenario for testing the GFC, we placed a replay client in Iran, a replay server is in the US, and we use a trace of web traffic recorded from the US that we confirmed is blocked in Iran. The signal for blocking in Iran is that the client receives an unsolicited "`HTTP/1.1 403 Forbidden`" response in addition to two RST packets.

**Efficiency of classifier analysis.** For the tested website, *lib·erate* successfully identifies the matching fields used by the classifier with 75 replays, which takes about 10 minutes and consumes 300 KB of data.

**Classifier matching fields.** Similar to the GFC, the Iranian classifier uses keyword matches in the HTTP Host header (e.g., facebook.com). Aside from that, we found substantial differences when compared to other classifiers in our study.

For one, we found that prepending packets does not appear to change classification results, suggesting that the classifier checks every packet in a flow instead of using a "match and forget" policy. Specifically, we found no difference in classification result when appending any number of 1,400 B packets, up to 1,000 packets.

Another observation is that traffic traversing the classifier is *not blocked* if the server port is not the standard HTTP port 80 (e.g., traffic blocked on port 80 is not blocked if the HTTP server uses port 8080), suggesting that the classifier uses both port-specific and content-specific rules. As a result, *lib·erate* must use only port 80 to identify the classifier's matching fields in Iran.

**Evading classification.** Unsurprisingly, inert packet insertion techniques do not work for evading the classifier in Iran, due to the fact that the classifier inspects every packet in a flow. That said, we found that traffic *without any blocked content* will be blocked if it is preceded with an inert packet containing a blocked payload—an example of misclassification. Though the TTL-limited probe technique was not effective for evasion, we nonetheless found that the classifier is eight hops away from our client in our experiments.

Interestingly, we can successfully evade classification simply by using payload splitting; namely, by splitting the payload of an IP packet's matching field across two packets (with or without reordering). We suspect this occurs because the classifier in Iran uses a per-packet classification implementation (i.e., it makes classification

decisions on each packet independently) instead of doing classification on a stream of bytes that spans multiple packets. We also found that IP fragments were dropped before reaching our server when testing from Iran, whereas IP fragments were reassembled and delivered to our server in all other tested environments.

Due to the fact that the Iran classifier inspects *every* packet, we did not conduct an analysis of classification flushing and thus omit it from this section.

## 7 DISCUSSION

This section discusses limitations, future work, and other issues.

**Arms race.** *lib·erate* does not end the arms race between networks that deploy differentiation and those who seek to evade it; rather, it provides efficient, unilateral evasion in a way that we expect will make the arms race substantially more expensive for networks. In addition, despite the fact that techniques to prevent some of our evasion techniques have been available for more than a decade, our results show that few are actually deployed.

**Impact of filtering.** We found that many of the inert packets that worked in our testbed were dropped in every operational network we tested. This is likely due to routers and/or firewalls that drop malformed packets. To maximize the potential for evasion, operators of *lib·erate* deployments should disable such filters on *lib·erate* traffic to the extent possible.

**IPv6.** This paper focused exclusively on IPv4 traffic. While we did not test IPv6 traffic in this study, we later observed Verizon throttled video traffic on prepaid SIMs (similar to what was reported by [14]) both over IPv4 and IPv6. Based on this observation, we will extend *lib·erate* to evade differentiation over IPv6.

**Deployment location.** This paper focuses primarily on a client deployment, because doing so makes it easy to use *lib·erate* by running replay servers at arbitrary locations via cloud services. We also support server-only deployments, but they require control over clients in affected networks for replay traffic. Some platforms, such as ICLab and OONI [42], provide such an ability but the scale of such systems is in general small.

**Detection and bidirectional *lib·erate*.** It is possible to detect our approaches and adapt to them using devices that more extensively keep track of state, or that permit some level of collateral damage. *lib·erate* in part addresses this by selecting from a suite of evasion techniques, each of which requires a different type of countermeasure. To further enhance *lib·erate*'s resilience to countermeasures, we are investigating techniques that modify application-layer properties of packets in arbitrary ways. This requires support on the server side, and coordination between *lib·erate* clients and servers. Such an approach can be combined with our current design to provide evasion techniques that are not only resilient to adaptation, but that can incorporate payload-modification strategies that are not publicly known by the differentiating ISP *a priori*.

**Masquerading.** This paper focuses on how *lib·erate* evades differentiation. However, in some cases users may want to masquerade as a type of differentiated traffic (e.g., if it is zero rated and/or receives better performance). Our framework supports masquerading as long as users supply traffic to place in inert packets, and we are implementing this as future work.

**Lawfulness and terms of service.** *lib·erate* can be used as a vehicle for enabling free speech where it is suppressed, but it also can be used to violate terms of service and or local laws—in fact, it can do all of these at the same time. We do not solve these issues; rather, we provide a technical solution for users that want to ensure that their network traffic is treated neutrally and we do not condone uses that can lead to harm.

**Ethics.** With the exception of experiments in Iran, all tests were conducted by one of the authors. For the tests in Iran, we used a virtual private server acquired by an ICLab [4] partner who consented to allowing its use for censorship measurements. The ICLab project that the partner participates in has IRB approval.

## 8 CONCLUSION

In this paper, we designed and implemented *lib·erate*, a tool that allows *unmodified* network applications to *automatically, adaptively, and unilaterally* evade middleboxes that provide them with unwanted differential service (e.g., blocking or shaping). Our approach systematically leverages inconsistencies between the end-to-end view of network flows and the view only from middleboxes, to evade policies applied to flows matching traffic-classification rules. We showed that *lib·erate* has reasonably low data-consumption overhead during the out-of-band classifier-detection phase, and that afterward it evades classifiers with negligible overhead. We then used our approach to characterize and evade several middlebox policies, both in our testbed and in nation-scale operational networks. In future work, we will enhance *lib·erate*'s robustness by incorporating application-layer modification.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Roya Ensafi, for their valuable feedback. This work was funded in part by the National Science Foundation (CNS-1617728, CNS-1700657, CNS-1350720) and a Google Research Award. Any opinions, findings, and conclusions or recommendations expressed in this comment are those of the authors and do not necessarily reflect the views of the NSF or Google.

## REFERENCES

[1] 2017. AT&T Stream Saver. https://www.att.com/offers/streamsaver.html. (April 2017).
[2] 2017. Censorship of cloudfront.net in China. https://en.greatfire.org/search/all/cloudfront.net. (August 2017).
[3] 2017. Hola - Free VPN, Secure Browsing, Unrestricted Access. http://hola.org/. (April 2017).
[4] 2017. ICLab. https://iclab.org/. (April 2017).
[5] 2017. Tor Pluggable Transports. https://www.torproject.org/docs/pluggable-transports.html.en. (April 2017).
[6] 2017. Wiki Meek. https://trac.torproject.org/projects/tor/wiki/doc/meek. (April 2017).
[7] Collin Anderson. 2012. The Hidden Internet of Iran: Private Address Allocations on a National Network. *CoRR* abs/1209.6398 (2012). http://arxiv.org/abs/1209.6398
[8] Collin Anderson. 2013. Dimming the Internet: Detecting Throttling as a Mechanism of Censorship in Iran. *CoRR* abs/1306.4361 (2013). http://arxiv.org/abs/1306.4361
[9] Daniel Anderson. 2012. Splinternet Behind the Great Firewall of China. *Queue* 10, 11 (2012), 40.
[10] Anonymous. 2012. The collateral damage of internet censorship by dns injection. *ACM SIGCOMM Computer Communication Review* 42, 3 (2012).
[11] Anonymous. 2014. Towards a Comprehensive Picture of the Great Firewall's DNS Censorship. In *Proc. of USENIX FOCI*. https://www.usenix.org/conference/foci14/workshop-program/presentation/anonymous
[12] Simurgh Aryan, Homa Aryan, and J. Alex Halderman. 2013. Internet Censorship in Iran: A First Look. In *Proc. of USENIX FOCI*.

[13] V. Bashko, N. Melnikov, A. Sehgal, and J. Schonwalder. 2013. BonaFide: A traffic shaping detection tool for mobile networks. In *IFIP/IEEE International Symposium on Integrated Network Management (IM2013)*.

[14] Jon Brodkin. 2017. Verizon accused of throttling Netflix and Youtube, admits to 'video optimization'. urlhttps://arstechnica.com/information-technology/2017/07/verizon-wireless-apparently-throttles-streaming-video-to-10mbps/. (July 2017).

[15] David Choffnes, Phillipa Gill, and Alan Mislove. 2017. An Empirical Evaluation of Deployed DPI Middleboxes and Their Implications for Policymakers. In *Proc. of TPRC*.

[16] R. Clayton, S. Murdoch, and R. Watson. 2006. Ignoring the great firewall of China. In *Proc. of PETs*.

[17] Jedidiah R. Crandall, Daniel Zinn, Michael Byrd, Earl Barr, and Rich East. 2007. ConceptDoppler: A Weather Tracker for Internet Censorship. In *Proc. of ACM CCS*. http://doi.acm.org/10.1145/1315245.1315290

[18] Gregory Detal, Benjamin Hesmans, Olivier Bonaventure, Yves Vanaubel, and Benoit Donnet. 2013. Revealing Middlebox Interference with Tracebox. In *Proc. of IMC*.

[19] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-generation Onion Router. In *Proc. of USENIX Security*.

[20] Marcel Dischinger, Massimiliano Marcon, Saikat Guha, Krishna P. Gummadi, Ratul Mahajan, and Stefan Saroiu. 2010. Glasnost: Enabling End Users to Detect Traffic Differentiation. In *Proc. of USENIX NSDI*.

[21] Haixin Duan, Nicholas Weaver, Zongxu Zhao, Meng Hu, Jinjin Liang, Jian Jiang, Kang Liz, and Vern Paxson. 2012. Hold-On: Protecting Against On-Path DNS Poisoning. In *SATIN'12*.

[22] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. 2015. Examining How the Great Firewall Discovers Hidden Circumvention Servers. In *Internet Measurement Conference*. ACM.

[23] FCC. 2015. Protecting and Promoting the Open Internet. https://www.federalregister.gov/articles/2015/04/13/2015-07841/protecting-and-promoting-the-open-internet. (April 2015).

[24] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. 2015. Blocking-resistant communication through domain fronting. *Proc. of PETs* (2015).

[25] John Geddes, Max Schuchard, and Nicholas Hopper. 2013. Cover Your ACKs: Pitfalls of Covert Channel Censorship Circumvention. In *Proc. of ACM CCS*. https://doi.org/10.1145/2508859.2516742

[26] B. Hahn, R. Nithyanand, P. Gill, and R. Johnson. 2016. Games without Frontiers: Investigating Video Games as a Covert Channel. In *Proc. of IEEE Euro S&P*. https://doi.org/10.1109/EuroSP.2016.17

[27] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. 2011. Is It Still Possible to Extend TCP?. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC '11)*. ACM, New York, NY, USA, 181–194. https://doi.org/10.1145/2068816.2068834

[28] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. 2013. The Parrot Is Dead: Observing Unobservable Network Communications. In *Proc. of IEEE S&P*. https://doi.org/10.1109/SP.2013.14

[29] Amir Houmansadr, Thomas Riedl, Nikita Borisov, and Andrew Singer. 2013. I Want my Voice to be Heard: IP over Voice-over-IP for Unobservable Censorship Circumvention. In *Proc. of NDSS*.

[30] Jill Jermyn and Nicholas Weaver. 2017. Autosonda: Discovering Rules and Triggers of Censorship Devices. In *7th USENIX Workshop on Free and Open Communications on the Internet (FOCI 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/foci17/workshop-program/presentation/jermyn

[31] Arash Molavi Kakhki, Fangfan Li, David R. Choffnes, Ethan Katz-Bassett, and Alan Mislove. 2016. BingeOn Under the Microscope: Understanding T-Mobile's Zero-Rating Implementation. In *Proc. of SIGCOMM Workshop on Internet QoE*.

[32] Arash Molavi Kakhki, Abbas Razaghpanah, Anke Li, Hyungjoon Koo, Rajesh Golani, David R. Choffnes, Phillipa Gill, and Alan Mislove. 2015. Identifying Traffic Differentiation in Mobile Networks. In *Proc. of IMC*. https://doi.org/10.1145/2815675.2815691

[33] Sheharbano Khattak, Mobin Javed, Philip D Anderson, and Vern Paxson. 2013. Towards Illuminating a Censorship Monitor's Model to Facilitate Evasion.. In *FOCI*.

[34] Christian Kreibich, Mark Handley, and V Paxson. 2001. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium*, Vol. 2001.

[35] Fangfan Li, Arash Molavi Kakhki, David Choffnes, Phillipa Gill, and Alan Mislove. 2016. Classifiers Unclassified: An Efficient Approach to Revealing IP-Traffic Classification Rules. In *Proc. of IMC*.

[36] Moxie Marlinspike. 2016. Doodles, stickers, and censorship circumvention for Signal Android. https://whispersystems.org/blog/doodles-stickers-censorship/. (December 2016).

[37] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. 2012. SkypeMorph: Protocol Obfuscation for Tor Bridges. In *Proc. of ACM CCS*. https://doi.org/10.1145/2382196.2382210

[38] J.C. Park and J.R. Crandall. 2010. Empirical study of a national-scale distributed intrusion detection system: Backbone-level filtering of HTML responses in China. In *Proc. of ICDCS*.

[39] Vern Paxson. 1999. Bro: a system for detecting network intruders in real-time. In *Computer networks*, Vol. 31. Elsevier, 2435–2463.

[40] Thomas H Ptacek and Timothy N Newsham. 1998. *Insertion, evasion, and denial of service: Eluding network intrusion detection*. Technical Report. DTIC Document.

[41] Jie Qiu, Tian Zhihong, Ye Jianwei, and Tang Shuofei. 2011. Design, implementation and optimization of network access control system based on routing diffusion. In *Proc. of Information Technology and Artificial Intelligence Conference (ITAIC)*.

[42] Abbas Razaghpanah, Anke Li, Arturo Filastò, Rishab Nithyanand, Vasilis Ververis, Will Scott, and Phillipa Gill. 2016. Exploring the Design Space of Longitudinal Censorship Measurement Platforms. *CoRR* abs/1606.01979 (2016). http://arxiv.org/abs/1606.01979

[43] Peter Svensson. 2007. Comcast Blocks Some Internet Traffic. http://www.washingtonpost.com/wp-dyn/content/article/2007/10/19/AR2007101900842.html. (October 2007).

[44] Mukarram Bin Tariq, Murtaza Motiwala, Nick Feamster, and Mostafa Ammar. 2009. Detecting Network Neutrality Violations with Causal Inference. In *CoNEXT*.

[45] ViewDNS.info. 2011. DNS Cache Poisoning in the People's Republic of China. (2011). http://viewdns.info/research/dns-cache-poisoning-in-the-peoples-republic-of-china/.

[46] 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 *Proc. IMC*.

[47] N. Weaver, R. Sommer, and V. Paxson. 2009. Detecting forged TCP reset packets. In *Proc. of NDSS*.

[48] Philipp Winter and Yawning Angel. 2014. obfs4 (the obfourscator). https://gitweb.torproject.org/pluggable-transports/obfs4.git/tree/doc/obfs4-spec.txt. (May 2014).

[49] Philipp Winter, Tobias Pulls, and Juergen Fuss. 2013. ScrambleSuit: A Polymorphic Network Protocol to Circumvent Censorship. In *Proc. of Workshop on Privacy in the Electronic Society*. http://www.cs.kau.se/philwint/scramblesuit/wpes2013.pdf

[50] X. Xu, Z. Mao, and J. Halderman. 2011. Internet censorship in China: Where does the filtering occur?. In *Passive and Active Measurement*.

[51] Ying Zhang, Z. Morley Mao, and Ming Zhang. 2009. Detecting Traffic Differentiation in Backbone ISPs with NetPolice. In *Proc. of IMC*.

[52] J. Zittrain and B. Edelman. 2003. Internet filtering in China. *IEEE Internet Computing* 7, 2 (2003), 70–77.