Classifiers Unclassified: An Efficient Approach to Revealing IP Traffic Classification Rules

Fangfan Li* Arash Molavi Kakhki* David Choffnes* Phillipa Gill† Alan Mislove*
*Northeastern University †Stony Brook University

ABSTRACT

A variety of network management practices, from bandwidth management to zero-rating, use policies that apply selectively to different categories of Internet traffic (e.g., video, P2P, VoIP). These policies are implemented by middleboxes that must, in real time, assign traffic to a category using a classifier. Despite their important implications for network management, billing, and net neutrality, little is known about classifier implementations because middlebox vendors use proprietary, closed-source hardware and software.

In this paper, we develop a general, efficient methodology for revealing classifiers' matching rules without needing to explore all permutations of flow sizes and contents in our testbed environment. We then use it to explore implementations of two other carrier-grade middleboxes (one of which is currently deployed in T-Mobile). Using packet traces from more than 1,000,000 requests from 300 users, we find that all the devices we test use simple keyword-based matching rules on the first two packets of HTTP/S traffic and small fractions of payload contents instead of stateful matching rules during an entire flow. Our analysis shows that different vendors use different matching rules, but all generally focus on a small number of HTTP, TLS, or content headers. Last, we explore the potential for misclassification based on observed matching rules and discuss implications for subversion and net neutrality violations.

1. INTRODUCTION

Today's IP networks commonly use middleboxes to perform management tasks that include bandwidth management [6], protecting users from malicious traffic [11], performance optimization [18, 20], and zero-rating [3]. While previous work has revealed the existence of middleboxes and their policies using black-box methods [2, 5, 7, 10, 12–14, 16, 17, 19, 21, 22], there is little work that investigates *how* these middleboxes determine which traffic is subject to a policy. In this work, we present the first general approach for doing so, and use this to characterize three carrier-grade middleboxes.

IMC 2016, November 14 - 16, 2016, Santa Monica, CA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4526-2/16/11…$15.00
DOI: `http://dx.doi.org/10.1145/2987443.2987464`

To facilitate network management, numerous vendors provide middleboxes that allow operators to specify high-level policies for traffic management (e.g., block malicious traffic, prioritize VoIP) without needing to know the details of how to implement them. In general, these policies include a *match rule* (or classification rule) that identifies a class of traffic, and an *action* that specifies what should be done to this class of traffic. An important challenge for any middlebox is how to develop matching rules that reliably identify a traffic class—often in real time so that it can apply a policy to it. While certain types of classification are straightforward (e.g., identifying DNS traffic), accurately classifying traffic into classes such as video, voice, and Web is challenging due to confounding factors such as SSL encryption, ubiquitous HTTP transport, and the absence of standard content encodings. Because these devices are expensive, sold under restrictive license agreements, and deployed in ways that are not transparent to users or researchers, little is known about how middlebox classifiers work and their implications.

In this paper, we are the first to identify and characterize the *classification rules* for HTTP(S) traffic implemented in today's carrier-grade middleboxes. This allows us to understand how rules are deployed, their impact on topics such as network neutrality, and how they can be subverted. Further, our general algorithm for identifying classification rules can facilitate auditing and analysis of future middleboxes and their policies by users, policymakers, and regulators. Our key contributions follow.

First, we develop a general methodology for identifying the matching rules used by a classifier. To address the potential combinatorial explosion of tests required to uncover them, we propose the notion of *Franken-Flows*, i.e., flows combining features of multiple, different packet traces generated by applications that are subject to classification. Doing so allows us to focus only on traffic that is likely to trigger matching rules.

Second, we conduct a detailed study of the classification rules used by devices in a controlled setting and in the wild. These include a carrier-grade packet shaper and an IPS device in our lab, as well as a third system that enforces T-Mobile's BingeOn service. Through traffic-replay and analysis, we find that the devices all analyze a small number of TCP, HTTP, and TLS fields (e.g., Host and SNI) to classify network traffic in our test suite, and *do not* use a fixed set of ports and IP addresses.

Third, we find that the devices use simple text-based matching in HTTP and TLS fields, indicating that their accuracy is limited by the specificity of the string-matching patterns that they use to match in HTTP headers or TLS handshakes. We show that these strings can lead to misclassification, both in terms of false positives and false negatives.

Fourth, we find that the devices exhibit a "match-and-forget" policy where an entire flow is classified by *the first rule that matches*, even if later packets match a different rule that would lead to more accurate classification. Further, when keywords for multiple classes appear in the same field in the same packet (e.g., the `Host` header contains `facebook.youtube.com`), the devices assign it to a single class using deterministic matching-rule priorities. These simple matching priorities are easily exploited to allow one class of traffic to masquerade as another and thus evade or subvert network policies.

2. METHODOLOGY

Our approach for identifying middlebox classification rules is to use a device under our control as ground truth for developing and validating our detection methodology, then to use our methodology to study other middleboxes. Similar to our previous work [9], we use an air-gapped testbed consisting of a client, server, and a middlebox between them (Fig. 1). In contrast to our prior work, which focused on identifying when differentiation occurs (e.g., shaping), this study focuses on identifying precisely what content in network traffic triggers classification that may lead to differentiation. The server also spoofs as an Internet gateway, allowing us to use arbitrary IP addresses in our traffic and capture all communication from all devices in the testbed.

The middleboxes in our possession allow us to log in, but do not reveal the exact classification rules that they use. However, we can access a user interface that indicates in real time the class of traffic for each flow that traverses it. This allows us to test hypotheses for classification rules, by sending carefully crafted traffic through the device to see how it is classified.

A naïve approach to hypothesis testing is to try every combination of packet sizes and packet payloads to determine which ones trigger classification. However,

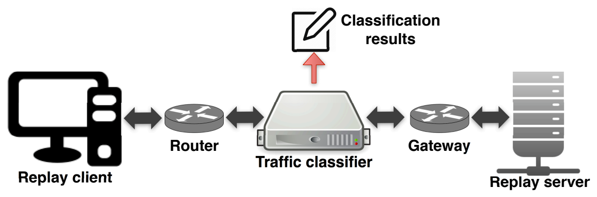

Figure 1: Testbed for controlled experiments with classifiers.

this is infeasible due to combinatorial explosion in the number of tests required to complete the analysis.

Instead, we leverage the fact that 1) a set of targeted applications potentially detected by the device is known a priori,[1] and 2) application-generated traffic already contains content that triggers rules. Using the first observation, we focus only on traffic from applications that are likely to be classified. The second observation allows us to use application-generated traffic as baseline that we then modify to efficiently search for the exact properties that trigger classification. Note that this approach extends to devices in our testbed and those in the wild, so long as we have both the network traffic trace to test, and a way to tell that traffic has been classified (e.g., rate limiting or zero-rating).

2.1 Dataset

Our approach requires us to send application-generated traffic through our testbed. While this may sound trivial, most interesting applications are closed-source and often require interaction with third-party servers to run; as a result, they cannot run in our air-gapped testbed. Instead, we use the record/replay system developed by Molavi et al. [9], which allows us to replay arbitrary network traces gathered from applications outside our testbed.

To obtain a set of applications representative of those users interact with, we leverage data from the ReCon project [15], which boasts more than 300 users of iOS and Android devices.[2] This provides us with 1,179,618 HTTP `GET` requests, in which there are 20,129 unique host headers, 8,701 unique `User-Agent` strings, and 685 distinct `Content-Type` headers [1]. We also extracted 1,727 unique Server Name Indication (SNI) fields from 51,985 HTTPS TLS handshakes.

Note that we only collect TCP traffic in our dataset, as the vast majority of flows and applications in our dataset use HTTP/S. Understanding UDP-based classification is part of ongoing work.

2.2 Identifying Matching Fields

The first step in understanding matching rules is determining which portions of network flows contain content that matches. Instead of permuting each bit in a

[1]In our testbed, we have ground truth information about all applications the device claims to identify. Outside of our testbed, the applications may be ones that an ISP publicly admits to targeting (e.g., T-Mobile's Binge On) or those that an observer simply suspects are targeted.

[2]This data is collected as part of ReCon's IRB-approved study.

network flow to observe its effect on classification, we take a more efficient approach that exploits the fact that our recorded traffic already matches rules.

In the base case (nothing is known about matching rules), we conduct a binary search where we replace half of the flow with random bytes and observe its effect on classification. Our assumption is that random bytes are very unlikely to match any classification rules.[3] If the traffic is no longer classified as the recorded application, we then identify a more specific region in which the method will be applied on—namely, the half of bytes that triggered the change. To do this, we first revert the bytes in that region back to their original content, then repeat the process of changing one half of the bytes at a time in that region. If both halves triggered a change, then we identify both halves as triggering the matching rules. Once we identify portions of network flows that trigger matching rules, we conduct more extensive tests by modifying each byte of a packet, one at a time, until we find the set of bytes that affect classification. These bytes comprise the field(s) used for matching. We intentionally avoid exhaustively evaluating all combinations of bits, as this is combinatorially infeasible to test at scale and is not the goal of our study.

Using this approach, we found that our packet-shaping device uses HTTP and TLS-handshake fields in their matching rules, and only for the first packet in each direction.[4] With this observation, we can more efficiently identify matching fields by using the known structure of HTTP and TLS packets and permuting only the corresponding fields. For example, when running this test on Netflix traffic, we find that the `Host` header triggers classification. For Pandora, we find that the `User-Agent` field is used. In addition, we omit any portion of network traffic that does not trigger classification, leading to substantially shorter replays.

2.3 Revealing Classification Rules

After identifying regions of network traffic that trigger matching rules, our next step is identifying the specific matching rule. For this work, we assume that matching rules take the form of regular expressions. While matching rules in general could be arbitrary and not based on text, our extensive analysis of three devices found no counterexamples to our assumption. Without loss of generality, we make an additional simplifying assumption that matching rules take the form of one or more basic regular expressions. While we could adapt our methodology to reveal more complex and diverse matching rules, we found no need to based on the three devices we tested with HTTP/S traffic.

[3]We validated this assumption by running 1,000 flows, each with different random bytes (from 100 to 1000 bytes) on port 80, all of which were classified as the same generic "HTTP" class.

[4]To validate this, we tried splitting the first packet into two packets, each with different subsets of bytes, and found that only the first packet was ever used for classification.

Algorithm 1 Isolating fields used in matching rules.

```
Inputs:
  $T$: the original trace
  $R$: classification
  $L$: list of potential matching fields
Output:
  $M$: set of combinations of matching fields

  function LocateMatchingFields($T, R, L$)
      Initialize number of fields $k$ to 1
      $M$ = Set()
      while $k \leq$ Length($L$) do
          for $F$ in combination($k$,$L$) do
              $T'$ = copy($T$)
              for $field$ in $F$ do
                  $T'$ = permute($field$)
              $R'$ = classification($T'$)
              if $R' \neq R$ then
                  $M$.add($F$)
          $k$ += 1
      Return $M$
```

We assume that a matching rule applies to a field, i.e., a region of a packet identified in the previous step. However, this alone does not tell us the matching rule because it is not specific. For example, if the field contains `Host: prefix.netflix.com`, it does not necessarily imply that the matching rule is "First find the `Host` field, then find any string that contains *netflix*" (e.g., the rule could also be "find any `Host` field that ends with *netflix.com*").

To reveal the precise matching rule, we conduct tests that randomize and otherwise alter subsets of content in each field. More generally, our approach is to permute content in matching fields to test hypotheses about the minimal set of content in matching fields that reliably triggers classification for a specific application (see Algorithm 1). This entails replacing content with random bytes, as well as adding and removing content. Because these generated flows include traffic from multiple sources of content, we refer to them as *FrankenFlows*.

2.3.1 Building a FrankenFlow

To build a *FrankenFlow*, we start with a *base flow* that is associated only with the application-layer protocol, i.e., HTTP or HTTPS. To do so, we build a flow that contains a dummy value for every known matching field for a classifier (revealed by the analysis in §2.2), where each field's value contains random content. For example, a base flow for HTTP traffic is a valid `GET` request and response pair where all the HTTP header values are replaced with random content:

```
GET {random text} HTTP/1.1
Host: {random text}
User-Agent: {random text}
...
```

Likewise, a base flow for HTTPS content is a TLS handshake with any SNI and server-certificate fields replaced with random content. We then automatically generate new *FrankenFlows* by replacing dummy values with payloads from matching fields in observed network traces.

2.3.2 Extracting Matching Rules

After isolating fields used for classification, we further randomize substrings of matching fields to isolate the portions of field triggering matching rules. Based on the resulting tests, we construct a minimal multi-level matching rule (i.e., a rule that contains the minimum number of fields to trigger a match) that is consistent with our observations.

For example, consider `qq.com` downloads. First, we observe that it is an HTTP `GET` request, a feature shared by many applications. Second, we find that the `Host` field is used for matching. Next, we modify the `Host` field by adding, removing, and replacing field values with random strings. During this process, we find that if `dl.xyz.qq.com` is replaced with `random.qq.random`, it is classified as QQ, and if replaced with `dl.xyz.random`, then it is classified as generic "HTTP". Last, if we replace the string with `random.dl.random.qq.random` or `random.qq.random.dl.random`, it is classified as QQ Download (a different class from "QQ"). We therefore infer that classifier rule is: 1) `HTTP GET` request, 2) Host header contains `*dl*.qq.*` or `*.qq.*dl*`.

To determine the impact of keyword location on classification, for each line in the HTTP header we randomly added, moved, removed, and replaced bytes both inside and outside of the identified keyword region. These tests reveal rules that match on simple keywords anywhere in a field, only at the beginning of a field, or only at the end of a field.

2.3.3 Rule Prioritization

Our methodology also reveals how matching rules are prioritized when a flow matches multiple rules simultaneously. To determine the priority of matching rules for different matching fields, we simply build all combinations of *FrankenFlows* with different matching field content, and determine the relative priority of each rule by inspecting the classification result. For example, if a *FrankenFlow* contains `Host: netflix.com` and `User-Agent: Pandora...` and is classified as Netflix regardless of the order the fields appear in the flow, then we conclude that Netflix's rule has a higher matching priority than Pandora's. Otherwise we infer that order determines matching priority.

For the case of multiple matches in the *same* field (e.g., `Host: netflix.youtube.com`), we explore all combinations of matching strings, by concatenating each matching string into one field value in a *FrankenFlow*. When the classifier selects one application that matches, we then determine whether the position of the corresponding matching string matters (e.g., was the application selected because the matching field was the first to appear in the *FrankenFlow*?).

Specifically, we move the matching string to the end of the concatenated string. If the classification result changes, then order also impacted rule matching (e.g., there is a tie-breaker based on position); otherwise, we know that order is not a factor. Once we determine the impact of position on classification for an application, we remove its string from the *FrankenFlow* and repeat these steps for the application that is classified next.

2.4 Efficiency

An important property for our methodology is efficiency, both in terms of time and data usage required to learn matching rules. We now evaluate this, and discuss an optimization that improves efficiency.

In addition to *FrankenFlow* optimizations, we leverage the observation that classification occurs using only the first two packets exchanged between client and server for HTTP/S traffic for the three devices we study. As a result, we need only conduct tests using only the first packet sent by a client and server. Note that if our assumption is incorrect (which is not the case for any devices we tested), we simply continue to replay larger fractions of flows until we are able to identify classification rules.

This approach significantly reduces the data and time needed for each test. For example, the size of a typical streaming video traffic replay is reduced from 30 MB to just 2 KB in our testbed (where we have immediate, ground-truth classification results). In our testbed, we tested all combinations of *FrankenFlows* using a single client and server in 14 hours.

In contrast, the naïve approach of permuting every bit of two 1460 B packets (one for client request and one for server response) would require 2^n tests, where n is the number of bits (23,200). Suffice it to say this number is enormous. Even if we were to permute only 80 bits of a field value, it would require 1.2x10^{24} tests, requiring $\approx 10^{16}$ years if each test takes one second.

Outside our testbed, we detect T-Mobile's Binge On by checking for zero-rated traffic against our data plan, as done previously [8]. Here, we use 10 KB *FrankenFlows*, to avoid attributing data charges to background traffic. Identifying the matching rules for an application in Binge On required on average 400 KB of data.

3. EXPERIMENTAL RESULTS

We now present results from our detailed look into the traffic-shaping device in our testbed, as it contained a large number of matching rules and provided us with ground-truth classification results.[5] In the next section, we summarize results from two other devices.

Table 1 lists the application categories our device detected and the number of matching rules we triggered in

[5]This device vendor is consistently identified as one of the key companies in the global DPI market [4].

Application category	Number of examples
Streaming Applications	33
Web Applications	32
File Transfer	10
VoIP	9
Instant Messaging	7
Games	5
Mail	5
Security	2
P2P	1

Table 1: Categories of applications detected by our device using test traffic gathered from user traces. Our traces provide good coverage across a variety of application types.

Header	Example Value	Application
`URI`	site.js?h={...}-**nbcsports**-com	NBC Sports
`Host`	Host: www.**netflix**.com	Netflix
`User-Agent`	User-Agent: **Pandora** 5.0.1 {...}	Pandora
`Content-Type`	Content-Type: video/**quicktime**	QuickTime

Table 2: HTTP matching fields and examples of applications classified by them. Matching keywords are in bold font.

each category. The device identified 104 different classes of traffic, covering 9 of the 13 categories it supports.[6]

3.1 Matching Fields

In this section, we identify the fields used for matching rules in the packet shaper. Recall that our *FrankenFlows* are generated using HTTP/S traffic, so our results only apply to these protocols.

HTTP. We find that this device first identifies HTTP traffic by looking for a request that starts with `GET`, then checks for application-specific content in the following headers: `URI`, `Host`, `User-Agent` (in the GET request) and `Content-Type` (in the GET response). Table 2 shows examples of matching fields. We note that in many cases, it is trivial to modify these fields to avoid classification, indicating that matching fields are not particularly resilient to adversarial behavior. Further, we will show in the next subsection that many of these rules can lead to false positives.

HTTPS. When applications use HTTPS, the encrypted TLS connection prevents the device from inspecting HTTP headers. In this case, the device identifies the corresponding application by matching text *anywhere* in the first two packets of the TLS handshake. These include strings in the TLS Server Name Indication (SNI) extension, and in the server-supplied SSL certificate, which contains fields such as the `Common Name` and `Subject Alternate Names` (SAN) list.

Interestingly, the classifier does not parse fields in the TLS handshake: our *FrankenFlows* were classified as an application even if the TLS handshake contained invalid data, so long as a keyword in a matching rule appeared in the packet payload. Similar to the case of HTTP, this would allow misclassification, e.g., by putting a keyword in the SAN list that matches a different application.

[6]The remaining 4 categories are for intranet traffic.

3.2 Matching Rules

Our analysis revealed that the HTTP matching rules used by the shaper can be described by a series of keyword matches on text in matching fields.[7] Examples of different matching rules include: `facebook` (Facebook), `.spotify.` (Spotify), `music.qq` (QQ Music), `instagram.com` (Instagram), `storage.live.com` (SkyDrive), and `itunes.apple` (iTunes). For HTTPS traffic, the matching rules are simply text strings. Examples include `netflix`, `facebook` and `cloudfront`.

None of the `Host`-header-based matching rules specify *where* in the matching field they occur. For example, `netflix.youtube.com` and `youtube.netflix.com` would both be classified as Netflix. In the case of matches of the `User-Agent` string, we find that 28 cases match only at the beginning of the string (e.g., Viber, Pandora, Pinterest, WhatsApp) and 4 cases can match anywhere in the string (e.g., YouTube, Twitter). We could find no general pattern for when location of keywords played a role in matching rules.

Interestingly, these matching rules are surprisingly brittle. For example, we already identified how Netflix and YouTube can be misidentified. Further, we did not find any restriction on whether a string appears at the end of a line, something that would avoid this case. As another example, we found that traffic from the "Galaxy Wars Multiplayer" app is mistakenly classified as BBC's iPlayer because the keyword `iplayer` appears in the HTTP header.

3.3 Priority Rules

We characterized how the device classifies a flow when *multiple rules* match a single flow. Such "tie-breaker" cases (which we found to be consistent over time) provide additional insight into the accuracy of a classifier and its resilience to subversion.

We tested all combinations of text in matching fields to identify how the shaper prioritizes them. Figure 2 depicts a decision process that captures our observations for HTTP traffic. The device first examines the content in the `GET` request, then it checks the `Content-Type` headers from the server response only if the request does not match any rule.

Importantly, the device exhibits a "match-and-forget" policy where an entire flow is classified by *the highest priority matching rule* even if later packets match a different rule that would lead to more accurate classification. For example, if the URI contains `/user/youtube` and the `Host` contains `facebook.com`, the flow is classified as YouTube instead of Facebook. Further, the priority of different matching rules in the `GET` request depends both on the field and the content. For

[7]Note that we cannot confirm whether this is the actual matching implementation; rather, we can only say that our observations suggest this is the case.

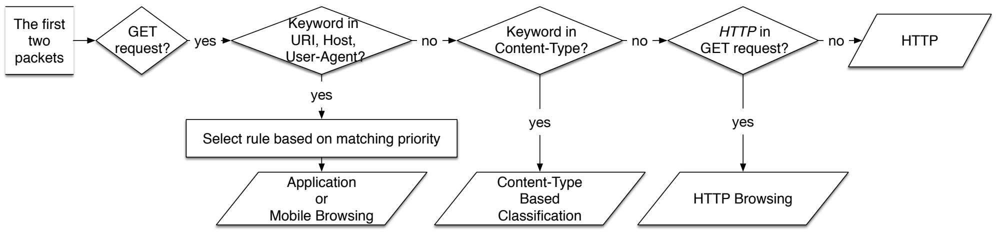
The first two packets
GET request?
yes
Keyword in URI, Host, User-Agent?
no
Keyword in Content-Type?
no
HTTP in GET request?
no
HTTP
yes
Select rule based on matching priority
Application or Mobile Browsing
yes
Content-Type Based Classification
yes
HTTP Browsing

Figure 2: Observed decision process for applying matching rules to HTTP traffic.

example, the *FrankenFlow* with `Host: www.hulu.com` and `User-Agent: Pandora` is classified depending on which header appears first in the GET request (i.e., if Hulu appears first, it is classified as Hulu). In contrast, the *FrankenFlow* with `Host: facebook.com` and `User-Agent: Pandora` is classified as Pandora regardless of the order of headers in the GET request.

Our results for classification-rule priorities for all 87 matching keywords in HTTP traffic are summarized in Table 3. We find six distinct priority groups, with only two groups containing more than one service. For example, if the URI matches NBC Sports, the classifier will label the entire flow as NBC Sports, even if the strings for YouTube, Netflix, or iPlayer also appear in the flow. When multiple matching strings from the third priority group appear in the same flow, the device uses yet another set of tie-breaking priorities, listed in Table 4. For example, if Netflix and Facebook appear in the Host header, then the flow is classified as Netflix regardless of where in the host field the string *netflix* appears (because Netflix is in a higher priority group). For matching rules in the same tie-breaking group, e.g., Netflix and LinkedIn, the flow is classified as the *first* keyword to appear in the Host string. We found no particular reason for the priority order found on this device.

The device classifies HTTPS traffic with multiple matching rules similarly to HTTP. For example, after matching on a TLS `Client Hello` packet, the device ignores matching rules in the server response.

4. ADDITIONAL CASE STUDIES

We also tested our methodology against two other traffic shaping devices: a carrier-grade IPS device in our possession and a deployed device in T-Mobile's network.

Carrier-grade IPS Device (Testbed). Our IPS device uses coarse-grained classification, e.g., it identifies "streaming video" traffic, but does not identify specific applications. We tested all our recorded streaming video traces against the device, and found that it does not classify any of them as video traffic except for a Netflix trace from 2014 (newer Netflix traces are not classified). In this case, the device matched on the string `ftypmp4` (presumably matching on "file type mp4") in the HTTP payload. Interestingly, this is the only case we found so far that inspect packet content beyond HTTP headers. We suspect that the poor matching results are due to the device using outdated rules, in addition to the fact that the device's primary purpose is detecting security threats (and not applications).

T-Mobile's Binge On Service T-Mobile's "Binge On" service allows opt-in subscribers to stream video content from participating providers without counting those bytes against their data plan (i.e., such content is zero-rated). In previous work, Molavi et al. [8] used adhoc techniques to show that T-Mobile identifies Binge-On-eligible content by inspecting the values in HTTP `Content-Type` and `Host` headers in addition to some fields of the TLS handshake. The ground truth signal that traffic is classified as Binge On is that its data is zero-rated (based on our account's data-usage counter). We use this information, along with the methodology in Section 2, to revisit Binge On—where we have no a priori knowledge about the vendor or their rules, and describe new findings below.

As an example of our findings, our analysis revealed that Netflix, which was previously identified using HTTP traffic, has since moved to HTTPS connections. Our analysis shows that the T-Mobile classifier checks the SNI field for the string `nflx` and the contents of the Common Name in the server certificate in the

Priority	Matching Field(s)	Example Apps	#cases
1	URI	NBC Sports	1
2	URI	YouTube	1
3	Host header, User-Agent	Netflix, ESPN, Pandora	80
4	Host header	Facebook	1
5	URI	iPlayer	1
6	User-agent	Apple Service, Android Content, Mobile browsing	3

Table 3: When a flow triggers multiple matching rules, the packet shaper classifies the flow according to the rule with highest priority. Tie-breaking priorities are listed in Table 4.

Tiebreaking Priority	Example Apps
1	NetFlix, LinkedIn, Skype, SkyDrive, Symantec, Yahoo Video, Gmail
2	Youtube, Instagram, Tango, AmazonCloud, Kaspersky, EA Games, Wechat
3	Facebook, Yahoo Mail, Zynga

Table 4: When a flow matches multiple rules in the same priority group in Table 3, the classifier picks the one with the highest tie-breaking priority. If the flow matches multiple rules of the same priority, the classifier picks the first matching string to appear in the flow.

TLS handshake for the value `nflxvideo.net`. These highly specific matching fields are surprising, since it seems likely that such domain names change often and require significant manual maintenance to ensure reliable classification. Interestingly, we further find that unlike the packet-shaping device in our testbed, T-Mobile's classifier parses the TLS handshake and will not properly detect HTTPS traffic if the TLS packets are malformed or the value is not in a matching field (SNI or Common Name).

We investigated priorities when a packet matches multiple applications and found that T-Mobile matches only on the last `Host` header if multiple are present. Interestingly, this leads to zero-rated traffic if the last header matches a BingeOn participant, even if other headers do not. If the first header is, for example, a Google App Engine domain, Web site content is returned without error even though there are multiple `Host` headers. This provides a way to zero-rate arbitrary HTTP traffic beyond was was found previously [8].

5. DISCUSSION AND CONCLUSION

In this paper, we presented an efficient approach for identifying the matching rules used in traffic shapers for applying policies such as packet shaping, security, and zero-rating. We showed that using existing application traffic facilitates efficient and reliable discovery of matching rules. We applied this approach to several devices and found that their approach to classifying the applications that we tested was surprisingly simple and generally based on matching text in a small number of HTTP and HTTPS fields.

Our work on revealing matching rules provides a solid framework for researchers and regulators to audit implementations of policies in today's middlebox deployments, understand their impact on issues such as network performance and net neutrality, and understand security implications. As part of our future work, we are investigating other deployments and extending our analysis to UDP traffic. We expect to further refine our methodology as we encounter new policies and classifier implementations not covered by our current approach.

Acknowledgements

We thank the anonymous reviewers and our shepherd Renata Teixeira for their helpful feedback. This work is supported in part by a Google Research Award, and NSF awards CNS-1617728, CNS-1350720, and CNS-1518845.

6. REFERENCES

[1] Complete MIME types list. http://www.freeformatter.com/mime-types-list.html.

[2] Neubot – the network neutrality bot. `http://www.neubot.org`.

[3] T-Mobile BingeOn. `http://www.t-mobile.com/offer/binge-on-streaming-video.html`.

[4] Global DPI market 2014-2018: Key vendors are allot communications, cisco, procera networks and sandvine. `http://www.prnewswire.com/news-releases/global-dpi-market-2014-2018-key-vendors-are-allot-communications-cisco-procera-networks-and-sandvine-275106991.html`, September 2014.

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

[6] M. Dischinger, M. Marcon, S. Guha, K. P. Gummadi, R. Mahajan, and S. Saroiu. Glasnost: Enabling end users to detect traffic differentiation. In *Proc. of USENIX NSDI*, 2010.

[7] FCC announces "Measuring Mobile America" program. `http://www.fcc.gov/document/fcc-announces-measuring-mobile-america-program`.

[8] A. M. Kakhki, F. Li, D. R. Choffnes, E. Katz-Bassett, and A. Mislove. BingeOn under the microscope: Understanding t-mobile's zero-rating implementation. In *Proc. of SIGCOMM Workshop on Internet QoE*, 2016.

[9] A. M. Kakhki, A. Razaghpanah, A. Li, H. Koo, R. Golani, D. R. Choffnes, P. Gill, and A. Mislove. Identifying traffic differentiation in mobile networks. In *Proc. of IMC*, 2015.

[10] P. Kanuparthy and C. Dovrolis. ShaperProbe: end-to-end detection of ISP traffic shaping using active methods. In *Proc. of IMC*, 2011.

[11] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the edge network. In *Proc. of IMC*, 2010.

[12] R. Mahajan, M. Zhang, L. Poole, and V. Pai. Uncovering performance differences among backbone ISPs with Netdiff. In *Proc. of USENIX NSDI*, 2008.

[13] Measurement Lab Consortium. ISP interconnection and its impact on consumer internet performance. `http://www.measurementlab.net/blog/2014_interconnection_report`, October 2014.

[14] A. Nikravesh, H. Yao, S. Xu, D. R. Choffnes, and Z. M. Mao. Mobilyzer: An open platform for controllable mobile network measurements. In *Proc. of MobiSys*, 2015.

[15] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. R. Choffnes. ReCon: Revealing and controlling privacy leaks in mobile network traffic. In *Proc. of MobiSys*, 2016.

[16] Switzerland network testing tool. `https://www.eff.org/pages/switzerland-network-testing-tool`.

[17] M. B. Tariq, M. Motiwala, N. Feamster, and M. Ammar. Detecting network neutrality violations with causal inference. In *CoNEXT*, 2009.

[18] N. Weaver, C. Kreibich, M. Dam, and V. Paxson. Here Be Web Proxies. In *Proc. PAM*, 2014.

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

[20] X. Xu, Y. Jiang, T. Flach, E. Katz-Bassett, D. Choffnes, and R. Govindan. Investigating transparent web proxies in cellular networks. In *Proc. PAM*, 2015.

[21] Y. Zhang, Z. M. Mao, and M. Zhang. Detecting Traffic Differentiation in Backbone ISPs with NetPolice. In *Proc. of IMC*, 2009.

[22] Z. Zhang, O. Mara, and K. Argyraki. Network neutrality inference. In *Proc. of ACM SIGCOMM*, 2014.