# OS Fingerprinting and Tethering Detection in Mobile Networks

Yi-Chao Chen[†], Yong Liao[‡], Mario Baldi[‡], Sung-Ju Lee[‡], Lili Qiu[†]
[†]The University of Texas at Austin, [‡]Narus Inc.

## ABSTRACT

Fingerprinting the Operating System (OS) running on a device based on its traffic has several applications, such as NAT detection, policy enforcement in enterprise networks, and billing for shared access in mobile networks. In this paper, we propose to utilize several features in TCP/IP headers for OS identification, and use real traffic traces to evaluate the accuracy of fingerprinting. Our trace-driven study shows that several techniques that successfully fingerprint desktop OSes are not effective for fingerprinting mobile devices. Therefore, we propose new features for fingerprinting OSes on mobile devices. We also consider NAT/tethering detection, an important application of OS fingerprinting. We use the presence of multiple OSes from the same IP address along with TCP timestamp, clock frequency, and boot time to detect tethering. Evaluation shows that our approach effectively detects tethering and outperforms existing schemes.

## Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations – Network monitoring

## General Terms

Algorithms, Measurement, Performance

## Keywords

OS Fingerprint; Tethering Detection; TCP/IP

## 1. INTRODUCTION

Identifying the Operating System (OS) running on an end device based on its traffic is valuable in many contexts. For example, an enterprise network may restrict the usage of specific OSes for security reasons. Moreover, OS fingerprinting can be used to detect NAT/tethering (*i.e.*, multiple devices sharing the Internet connection of a mobile device), since the presence of multiple OSes sharing the same IP address is an indication of tethering, which may be prohibited by a wireless network due to resource usage concerns.

*IMC'14,* November 5–7, 2014, Vancouver, BC, Canada.
Copyright 2014 ACM 978-1-4503-3213-2/14/11 ...$15.00.
http://dx.doi.org/10.1145/2663716.2663745.

Motivated by this need, we examine a series of features in network traffic to understand their effectiveness in detecting the OS running on an end device. In particular, we consider IP Time-to-Live (TTL), IP ID monotonicity, TCP window size scale option, TCP timestamp, clock frequency, and boot time. Among them, TTL, IP ID, TCP timestamp option, and boot time have been considered in other contexts, including machine fingerprinting [2, 9, 29, 35, 39, 43]. We also propose several new features: the stability of the clock frequency, presence of TCP timestamp option, and the default set of TCP window size scale factors.

Next we apply OS fingerprinting to detect tethering using a simple probabilistic approach. It consists of two steps: (i) identifying the OS running on a device (*i.e.*, iOS, Android, or Windows) based on a combination of features, and (ii) determining if there is a tethering based on the number of OSes along with the number of distinct TTLs, TCP timestamp monotonicity, standard deviation of clock frequency, and standard deviation of boot time.

We evaluate our approach using traces we collected in 2013, as well as publicly available traces [15] collected during OSDI'06 [14] and SIGCOMM'08 [7]. We find that Apple iOS can be accurately identified, while Android and Windows are identified with $1.0$ precision (*i.e.*, the fraction of traffic our scheme detected as a given OS is indeed that OS) and $0.8$ recall (*i.e.*, the fraction of traffic from a given OS is correctly detected by our scheme). Tethering detection has $0.78 \sim 0.89$ recall when the target precision is $0.8$.

Our main contributions are as follows:

- We identify new features for OS fingerprinting, such as the presence of TCP timestamp, TCP window size scale factor, and standard deviation of clock frequency.
- We quantify the effectiveness of various individual features, including both new and previously proposed features. We show that clock skew, a feature proposed before for fingerprinting desktops in wired networks, does not work well in mobile networks. Existing work assumes that significant clock skew indicates different machines, but it is ineffective in a mobile context due to highly variable clock frequency in iOS devices and increased estimation error due to short transfers and unstable connectivity.
- We develop a probabilistic scheme that combines multiple features to detect OSes and tethering, and show it outperforms decision tree and linear regression.

## 2. TRACE DESCRIPTION

We use three packet traces in our study. The first two are WiFi packet traces captured during SIGCOMM'08 [7] and OSDI'06 [14], available from CRAWDAD [15]. We also collect our lab trace by setting up an AP in an office, recruiting users to use the AP for In-

**Table 1: Summary of the traces.**

| Trace | Time | Duration | # IPs | # pkts | # flows |
|---|---|---|---|---|---|
| OSDI06 Trace | 2006/11 | 1 day | 292 | 1,408K | 3,404 |
| SIGCOMM08 Trace | 2008/08 | 1 day | 223 | 1,107K | 2,586 |
| Lab Trace | 2013/10 | 2 hours | 56 | 193K | 741 |

ternet access, and capturing packet headers on the AP. Altogether, we had 14 captures from 4 different Android phone and tablet devices; 10 captures from iOS devices, including iPhone, iPod touch, and iPad; and 32 captures from laptops running Windows. Each capture lasts $10 \sim 30$ minutes. Table 1 summarizes the three traces.

## 3. OS FINGERPRINTING

We introduce a list of relevant features and describe how these features are used for OS fingerprinting. In this paper, we focus on identifying Windows, iOS, and Android, since recent market research [23,32] reports that Windows, iOS, and Android account for $12\%$, $43\%$, and $44\%$ of the laptop/phone/tablet OSes, respectively. Our methodology is general and can be easily extended to cover more OSes.

### 3.1 Features

We identify the following features for OS fingerprinting.

**IP Time-To-Live:** The TTL value in the IP header specifies the maximum number of hops a packet can traverse. Different OSes set different initial TTL values; Windows uses 64 or 128, while iOS and Android use 64 by default.

**IP ID Monotonicity:** The identification field in the IP header is primarily used in IP de-fragmentation. We observe that IP IDs in packets from Windows machines consistently increase monotonically over time. iOS devices always randomize the IP ID of each packet. Interestingly, some Android devices completely randomize the IP IDs, while others monotonically increase them for some time and periodically reset to random values.

**TCP Timestamp Option:** The TCP timestamp option [25] is used for measuring roundtrip time and protecting against wrapped sequence numbers. Most packets from Windows do not have TCP timestamp options, whereas packets from iOS and Android usually do [13].

**TCP Window Size Scale Option:** This option allows increasing TCP receiver window size beyond $65,535$ bytes. The scale value is negotiated during the TCP three-way handshake. Our traces reveal that the scale values vary across OSes: Windows uses $1$, $4$, or $256$; iOS uses $16$; and Android uses $2$, $4$, or $64$.

**Clock Frequency:** The clock frequency of a machine should be relatively stable. We observe that this holds true mostly for Windows and Android machines. Interestingly, the clock frequency of iOS varies over time. From the trace we collected from multiple iOS devices, we see that their clock frequency varies between 920Hz to 1000Hz. We suspect that iOS might dynamically adjust the clock frequency for power saving.

We estimate the clock frequency as follows. Let $t_1$ and $t_2$ denote the capturing time of two packets from the same device, and $T_1$ and $T_2$ be the logic timestamps (*e.g.*, TCP timestamp) embedded in those two packets. We compute clock frequency as $\frac{T_2-T_1}{t_2-t_1}$.

If the standard deviation of the estimated clock frequency from a flow is large (*i.e.*, $\geq 3$ in our evaluation), it implies the estimation frequency is unstable and is likely to be an iOS machine. Note that for OS detection, this heuristic may not be accurate since large standard deviation could be caused by multiple machines instead of one iOS machine. We will quantify the error through evaluation later.

**Table 2: Probability of identifying OS witch each feature. $thresh_{v1}$=0.05, $thresh_{v2}$=0.40, $thresh_t$=0.05, and $thresh_c$=3. $OS_a$ is Android; $OS_i$ is iOS; and $OS_w$ is Windows. We examine the distribution of values of each feature for different OSes and select the threshold yielding less than $10\%$ false positive.**

| **Feature** $f_i$ | $Pr(OS_a\|f_i)$ | $Pr(OS_i\|f_i)$ | $Pr(OS_w\|f_i)$ |
|---|---|---|---|
| TTL $= 128$ | 0 | 0 | 1 |
| TTL $\neq 128$ | 0.43 | 0.30 | 0.27 |
| IP ID monotonicity violation ratio $< thresh_{v1}$ | 0.18 | 0 | 0.82 |
| IP ID monotonicity violation ratio $\in [thresh_{v1}, thresh_{v2}]$ | 0.75 | 0 | 0.25 |
| IP ID monotonicity violation ratio $\geq thresh_{v2}$ | 0.22 | 0.78 | 0 |
| TCP TS Opt ratio $< thresh_t$ | 0 | 0 | 1 |
| TCP TS Opt ratio $\geq thresh_t$ | 0.59 | 0.41 | 0 |
| TCP WS Opt = 4 | 0.1 | 0 | 0.9 |
| TCP WS Opt = 16 | 0 | 1 | 0 |
| TCP WS Opt = 64 | 0.82 | 0 | 0.18 |
| TCP WS Opt = 256 | 0 | 0 | 1 |
| clock freq std $< thresh_c$ | 0.67 | 0.27 | 0.07 |
| clock freq std $\geq thresh_c$ | 0 | 1 | 0 |

Interestingly, this heuristic has better accuracy for tethering detection. When the large estimated frequency variation is due to multiple OSes instead of iOS, our scheme wrongly identifies iOS in the tethered traffic. However, because other OS fingerprinting features (*e.g.*IP TTL, TCP timestamp option, and TCP window size scale option) can still identify the correct OS, our scheme will identify iOS and non-iOS devices by combining results from all features. Since two or more OSes implies tethering usage, our scheme can still detect tethering correctly.

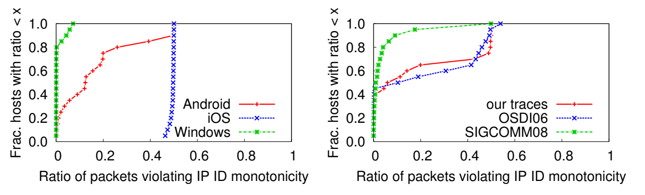
### 3.2 Using the Features

Exploiting each feature $f_i$ listed in Section 3.1 to detect OSes requires an important probability $Pr(OS_x \mid f_i)$, *i.e.*, the probability of being OS $x$ when feature $f_i$ is present. We use Bayes' rule to derive $Pr(OS_x \mid f_i) = \frac{Pr(f_i|OS_x)Pr(OS_x)}{Pr(f_i)}$, where $Pr(f_i \mid OS_x)$ and $Pr(f_i)$ are empirically computed using our lab trace; $Pr(OS_x)$ is the probability of $OS_x$ derived from the training traces. $Pr(f_i)$ denotes the fraction of IP addresses having feature $f_i$. An IP is considered to have feature $f_i$ if its feature value exceeds the corresponding threshold (*e.g.*, the fraction of packets with TCP timestamp exceeds a threshold).

We divide each trace into a training set and a testing set. Below we describe how we empirically determine the threshold for each feature using the training set. We later use the testing trace to evaluate the accuracy of our detection in Section 5.

**IP Time-To-Live:** From our training set, we learn that Android and iOS use 64 as the default TTL, while Windows uses 64 or 128. We can thus use the TTL value to identify Windows (*i.e.*, an initial TTL of 128) with high confidence. This rule accurately catches all Windows machines as shown in row 1 of Table 2. When the default TTL is not 128, it could be any OS. Because only a small number of packets from Windows set TTL other than 128 and there are more Android than iOS in the training set, the probability of being an Android is higher as shown in row 2 of Table 2.

**IP ID Monotonicity:** In our training set, we observe that Windows devices increase IP ID monotonically, iOS devices use random IP IDs, and their violation ratio (*i.e.*, the current packet's IP ID is smaller than the previous IP ID) is around $50\%$. There are



(a) Violation ratios across OSes (b) Violation ratios in the traces

**Figure 1: CDF of ratio of packets violating IP ID monotonicity.**

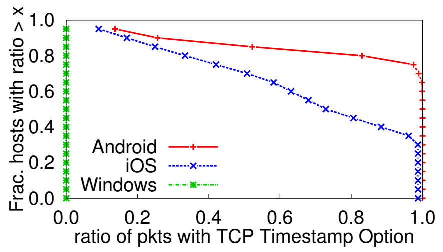


**Figure 2: CCDF of ratio of packets with the timestamp option.**

about 20% of Android devices randomizing the IP IDs. But 80% of Android devices have 40% or lower violation ratio. Figure 1(a) shows the ratio of packets violating IP ID monotonicity for three types of OSes in our training set.

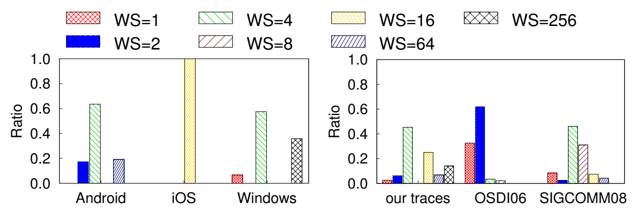
Rows 3∼5 in Table 2 show that requiring IP ID monotonicity identifies Windows machines fairly accurately while significant violations of IP ID monotonicity can be used to identify iOS. So we derive the following rule. When the violation ratio is less than 5%, most likely it is a Windows device. When the violation ratio is greater than 40%, it is iOS. When the violation ratio is in between (*e.g.*, $5\% \le ratio < 40\%$), it is likely to be Android.

Figure 1(b) shows that the violation ratios are smaller than 5% on 72% of machines in SIGCOMM'08 trace. This implies that a large number of machines are likely to be Windows. In our lab trace and OSDI'06 trace, violation ratios are smaller than 5% on 42% and 44% of machines, respectively. These machines are likely Windows machines, as well.

**TCP Timestamp Option:** As shown in Figure 2, the ratios of packets with TCP Timestamp option are more than 10% and 14% for iOS and Android devices, respectively. For Windows machines, the ratio is smaller than 5%. Hence, 5% is a good threshold to distinguish Windows machines from others. If the ratio of packets with TCP Timestamp option is smaller than 5%, we conclude that it is a Windows device. Rows 6 and 7 in Table 2 show the coverage of using the presence of TCP timestamp option for identifying the OS: it accurately identifies Windows. The accuracy for detecting Android and iOS is lower.

**TCP Window Size Scale Option:** The scale factor is determined by the maximum receiving buffer space and cannot be changed after the connection is opened. Figure 3(a) shows the scale factors selected by different OSes. We observe that only iOS uses 16; Windows and Android use 2, 4, 64 and 256. Figure 3(b) shows that no scale factor is set to 16 in OSDI'06 dataset. Since iOS was first released in 2006, it implies that 16 is unique for iOS devices. Hence we derive the following rule: a TCP window scale factor of 16 is iOS, 64 is Android, and 256 is Windows. Rows 8∼11 in Table 2 show it is fairly accurate for determining the OSes.

**Clock Frequency and Boot Time Estimation:** Figure 4 shows that the standard deviation of clock frequency in 90% of Windows machines is less than 1, and that of 90% of Android devices is less than 3. Therefore when the clock frequency exceeds 3, we conclude it is iOS. Row 13 in Table 2 shows that we identify all iOS correctly based on unstable clock frequency.



(a) Ratio in OSes (b) Ratio in data sets

**Figure 3: Ratio of selected TCP window scale option.**

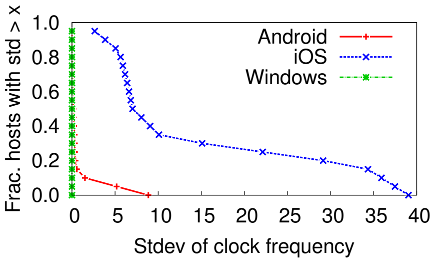


**Figure 4: CCDF of clock frequency std in OSes.**

## 3.3 Combining Features

So far, we have focused on using individual features. As we observed in Section 3.2, different features may work well in different scenarios. This motivates us to develop a technique to leverage multiple features to improve accuracy. We design a probability-based technique by applying the navïe Bayes classifier to effectively combine multiple features. Specifically, given the set of observed features $f_1 \sim f_k$, the probability of being $OS_x$ can be computed as Equation (1) if features $f_1 \sim f_k$ are independent.

$$Pr(OS_x \mid f_1, ..., f_k) = Pr(OS_x)\frac{Pr(f_i, ..., f_k \mid OS_x)}{Pr(f_i, ..., f_k)}$$
$$= Pr(OS_x)\frac{\prod_{i=1}^{k} Pr(f_i \mid OS_x)}{\prod_{i=1}^{k} Pr(f_i)}. \quad (1)$$

$Pr(OS_x)$ and $Pr(f_i|OS_x)$ are learned from the training traces. We then compute $Pr(f_i)$ based on all packets from an IP address in the testing trace and use Equation 1 to compute the probability that the IP uses $OS_x$. The OS is then identified as the one with the highest probability.

## 4. TETHERING DETECTION

The OS fingerprinting result can be used for tethering detection. To that end, we present a few more features specifically related to tethering detection and how tethering detection is conducted.

### 4.1 Features

If multiple types of OSes are detected from the traffic coming from an IP address, it is considered as tethering. In addition, the following features can also be used for tethering detection.

**Number of TTL Values:** If packets from an IP address has different TTL values, it is likely to be tethering.

**TCP Timestamp Monotonicity:** Packets generated by the same machine tend to monotonically increase TCP timestamp values, whereas packets from different machines usually have mixed TCP timestamp due to different clock offsets across machines.

**Clock Frequency:** If the standard deviation of clock frequency estimated using the packets from an IP address is too large, it is likely to be tethering.

**Boot Time:** Machine boot time can be inferred from TCP timestamp values in packets sent from that machine. Most OS implemen-

**Table 3: Probability of having tethering when the feature is observed (*i.e.*, $Pr(T|f_i)$), where all thresholds are derived from the training traces. $thresh_v = 0$, $thresh_c = 35$, and $thresh_b = 1455$.**

| Feature $f_i$ | lab trace | osdi06 | sigcomm08 |
|---|---|---|---|
| # distinct TTL $= 1$ | 0.33 | 0.24 | 0.24 |
| # distinct TTL $> 1$ | 0.96 | 0.95 | 0.92 |
| TCP TS monotonicity violation ratio $\leq thresh_v$ | 0.33 | 0.40 | 0.29 |
| TCP TS monotonicity violation ratio $> thresh_v$ | 1 | 1 | 1 |
| clock freq std $< thresh_c$ | 0.18 | 0.53 | 0.55 |
| clock freq std $\geq thresh_c$ | 1 | 1 | 0.67 |
| boot time std $< thresh_b$ | 0.1 | 0.36 | 0.66 |
| boot time std $\geq thresh_b$ | 1 | 1 | 0.88 |

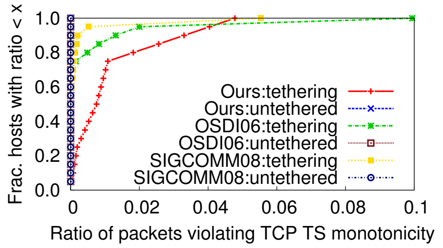


**Figure 5: CDF of ratio of packets that violate TCP timestamp monotonicity.**

tations use a random number as the starting value of TCP timestamps after booting. Hence the estimated boot time is not the real one. However, the value can still quite uniquely identify a machine because different machines have distinct boot times and distinct initial TCP timestamp values [39].

Note that TCP TS monotonicity, clock frequency, and the boot time are effective even when the tethered devices use the same OS, as the feature values vary across devices instead of OSes.

## 4.2 Using the Features

We describe how to use each feature for tethering detection. Similar to Section 3.2, we use the Bayes' rule to empirically compute $Pr(T|f_i)$ (*i.e.*, the probability of tethering under feature $f_i$) according to the training traces. To facilitate the empirical study, we simulate tethering activities in each trace by randomly mixing packets from different IPs and modifying the source IP address to make them look like from the same IP address. We assume that there is no tethering in the original traces. This assumption should hold in our lab trace due to the way in which they are collected. For OSDI'06 and SIGCOMM'08 traces, it is likely to be true since there is no reason for tethering when free WiFi is available (other literature [34] also makes a similar observation). For each trace, we select packets from 80% of the source IPs as the training traces to derive the threshold and use the remaining 20% as testing to quantify the accuracy of tethering detection in Section 5.3.

**IP TTL:** Based on the TTL analysis in Section 3.2, we conclude that there is tethering if the number of distinct TTLs in all packets from an IP address is more than one. From our training set, we find that this heuristic accurately identifies tethering: its coverage (*i.e.*, the fraction of traffic our scheme detected as tethering is indeed tethering) ranges from 0.92 to 0.96 in three different traces.

**TCP Timestamp Monotonicity:** Figure 5 shows the ratio of packets that violate TCP timestamp monotonicity. We see that untethered traffic have no violations, while 95% (our lab trace), 20% (OSDI), and 41% (SIGCOMM) of tethering machines have violation ratios larger than zero. Therefore, we conclude that the probability of tethering is one if the violation ratio of TCP timestamp monotonicity is larger than zero. When there is no violation of TCP timestamp monotonicity, the tethering probabilities are 0.33, 0.40, and 0.29 in our lab trace, OSDI'06 trace, and SIGCOMM'08 trace, respectively. The false negative cases are mainly due to two reasons. First, TCP timestamp option is not available in most of Windows devices. Second, sometimes flows from tethered devices do not overlap in time, and thus no violation is observed.

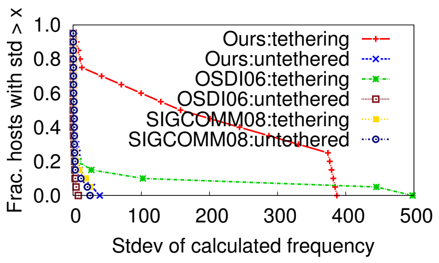


**Figure 6: CCDF of clock frequency standard deviation in tethering/untethered devices.**

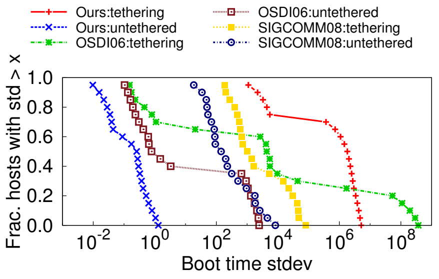


**Figure 7: CCDF of boot time std in tethering and untethering traffic.**

**Clock Frequency:** Figure 6 shows that the standard deviation in 90% of untethered traffic is smaller than 35 (lab trace), 9 (OSDI), and 12 (SIGCOMM). Therefore we conclude that there is tethering if the standard deviation is larger than 35.

**Boot Time:** Figure 7 shows standard deviation of estimated boot time. Using a large standard deviation of boot time can reliably detect tethering. When the standard deviation of boot time is larger than 1455, the probabilities of tethering are 1 for our lab trace and OSDI'06 trace, and 0.88 for the SIGCOMM'08 trace.

Table 3 is a summary of $Pr(T|f_i)$, *i.e.*, the probabilities of tethering we learned from our training data sets. Note that while the probabilities reported here may not always hold for other traces, the features we use and our methodology of deriving the thresholds for the features can be applied in general to other traces.

## 4.3 Combining the Features

We use two steps to compute tethering probability. First, we use features $f'_i$ for OS fingerprinting to derive the probability of having multiple OSes from an IP address, $Pr(multiOS|f'_1, ..., f'_k)$, as

$$Pr(multiOS|f'_1, .., f'_k)$$
$$= 1 - \sum_{x=1}^{m} Pr(OS_x|f'_1, ..., f'_k) \prod_{y=1..m}^{y \neq x} (1 - Pr(OS_y|f'_1, ..., f'_k)),$$

where $m$ is the number of different OSes, and probability $Pr(OS_x|f'_1, ..., f'_k)$ can be computed from Equation (1).

We then treat $Pr(multiOS|f'_1, .., f'_k)$ as one of the features for tethering detection, denoted as $g$, and use it along with the additional features presented in Section 4.2 to compute $Pr(T|f_1, ..., f_n, g)$ based on the Bayes' rule similar to Section 3.3.

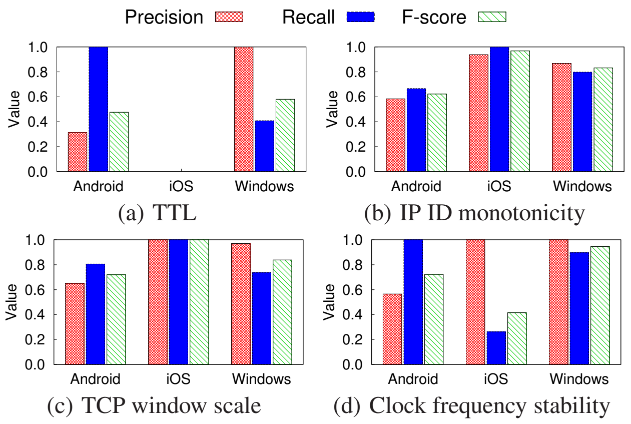


Figure 8: Accuracy of detecting OSes via individual features.

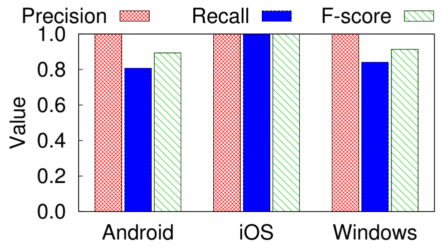


Figure 9: OS detection using combined method.

## 5. EVALUATION

### 5.1 Evaluation Metrics

We quantify the detection accuracy using three metrics: (i) *precision*, *i.e.*, the fraction of traffic our scheme detected as tethering (or have a given OS) is indeed tethering (or have that OS), (ii) *recall*, *i.e.*, the fraction of tethered traffic (or traffic from a given OS) are correctly detected by our scheme, and (iii) *F-score*, which is the harmonic mean of precision and recall (*i.e.*, $F-score = \frac{2}{1/\text{precision}+1/\text{recall}}$). For all three metrics, larger values indicate higher accuracy.

### 5.2 OS Detection Accuracy

We first evaluate how effective each individual feature is in detecting OSes using our lab trace, in which we have the ground truth on which OS generated each capture. We consider four OS specific features in this evaluation: TTL, IP ID monotonicity, TCP window scale, and clock frequency stability.

From Figure 8(a), we see that the precision of identifying Windows via TTL feature is high, because only Windows sets default TTL to 128. However, since Windows can also use 64 as its TTL, the recall is low. Android and iOS do not use 128 as the default TTL. When the TTL is not 128, the device is identified as Android because the probability of being Android is highest as shown in Table 2. Therefore, the recall of Android is one while recall and precision of iOS are both zero.

iOS has unique behaviors for IP ID monotonicity and TCP window scale, and thus we see from Figure 8(b) and 8(c) that both features identify iOS accurately. Although iOS devices are also distinguishable by having unstable clock frequency, some of the iOS packet captures in our lab trace are too short to reliably compute clock frequency. Therefore, we see from Figure 8(d) that the recall of identifying iOS via clock frequency is low.

The benefit of combining multiple features via our probability-based classifier is depicted in Figure 9. We see that our approach accurately identifies the OSes of most machines. The improvement is especially significant for identifying Android.

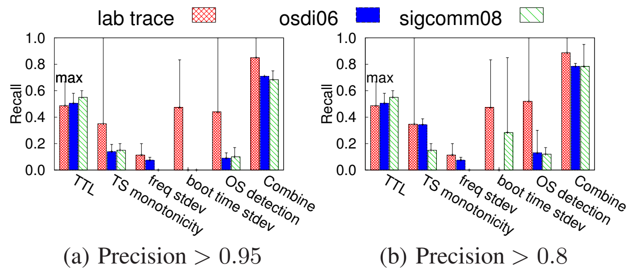


Figure 10: The recall of individual and combined technique when the precision is fixed to 0.95 and 0.8.

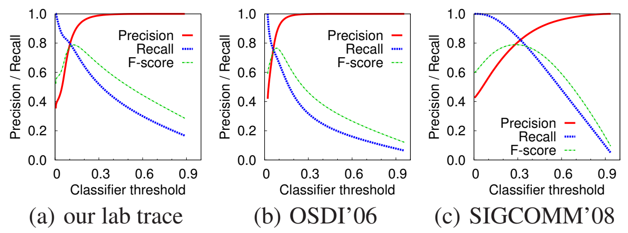


Figure 11: Average detection accuracy as the classifier threshold is varied in our probability-based classifier.

### 5.3 Tethering Detection Accuracy

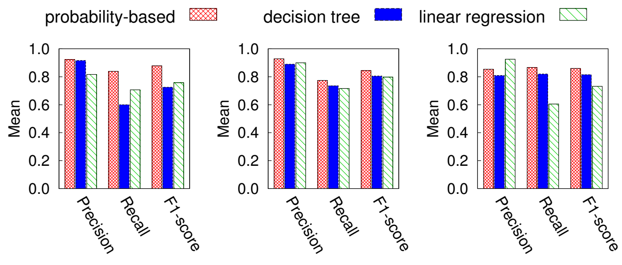
Next we evaluate the accuracy of applying the OS fingerprinting technique to tethering detection. Figure 10 shows combining multiple features using our probability-based classifier. We fix the targeted precision to be 0.95 and 0.80, and evaluate the average and maximal recall of detecting tethering via individual features and our probability-based classifier (depicted as "combine" in Figure 10).

Figure 10 provides a few interesting insights. First, our probability-based classifier consistently outperforms the schemes using individual features. Second, the clock frequency and boot time features are not effective in OSDI'06 and SIGCOMM'08 traces, because (i) the ratios of packets with TCP timestamps in two traces are small, and thus the clock frequency and boot time can be estimated in only a small number of devices, and (ii) the two traces have mostly short flows, which increase the error in clock frequency estimation. Moreover, we observe that OS detection is very effective in our traces but not in the other two. The main reason is that we do not have the ground truth on which OSes were used in OSDI'06 and SIGCOMM'08 traces and the probabilities learned from our lab trace may not work well for other traces.

The results shown in Figure 10 also demonstrate the trade-off between precision and recall in detecting tethering via our probability-based classifier. When we relax the target precision to 0.8, the recall of our probability-based classifier increases from 0.68∼0.85 to 0.78∼0.89. As we expect, the recall measurement will be higher when a lower precision is targeted.

In addition, Figure 11 shows the trade-off between precision and recall by varying the classification threshold. The probability-based classifier detects tethering when the probability is larger than a classification threshold. Determining the threshold itself is a challenging problem [12, 34]. A higher threshold implies higher confidence on whether the device is tethering. Therefore, when we increase the classifier threshold, the precision becomes higher while the recall becomes lower.

Next we compare our probability-based classifier with two well-known classifiers: linear regression and decision tree. In linear regression, each feature takes its actual value (*e.g.*, the number of distinct TTL, clock frequency standard deviation), and tethering



(a) our lab trace (b) OSDI'06 (c) SIGCOMM'08

**Figure 12: Comparison between decision tree and regression-based classifiers.**

is presented as a binary indicator. The linear regression classifier learns the weight of each feature from the training data such that the weighted sum of all features best approximates the binary tethering indicator. We estimate the weight by solving a linear inverse problem using min $L2$, which performs the least square fit. For the decision tree, we use an existing implementation from Weka [21]. For all three schemes, we select the classifier thresholds to maximize their F-scores. As we see from Figure 12, our probability-based classifier consistently outperforms decision tree by 5∼21% and linear regression by 6∼18% in the F-score measurement.

## 6. DISCUSSION

**Other Features:** Different OSes adopt different TCP congestion avoidance algorithms, referred to as *TCP flavors*. For example, OSX and iOS use New Reno [22] by default; Android and Linux use CUBIC [20] since kernel 2.6.19; Linux up to kernel 2.6.18 uses BIC [40]; Windows XP and earlier versions use New Reno; and Windows Server 2008 uses Compound TCP [38]. TCP flavors can be inferred by estimating the congestion window and how it changes in response to losses and RTT [26, 33]. It can be incorporated into our probability-based approach by adding another feature: $Pr(OS_x|flavor_y)$ from the training data. It is challenging to accurately infer TCP flavor from mobile network traces, because most flows are short and the throughput is usually limited by the lack of data to send [26] instead of congestion control algorithms.

The destination of the connection can also be used to identify OSes. For example, if a device connects to a Windows Update server, it is likely to be Windows. Similarly, connecting to Google Play or Apple App Store can also suggest Android and iOS devices.

Network Time Protocol (NTP) can also reveal the OS and tethering usage. The intervals between NTP messages vary from $64s$ to $1024s$ [3]. An interval less than $64s$ or changing dramatically can imply tethering. Besides, the default NTP servers are different across OSes and can be used to identify OSes (*e.g.*, `time.windows.com` or `time.apple.com`).

Tracking intervals between DNS queries sent from an IP address to the same hostname may be useful for tethering detection. NTP and DNS queries have not been considered for OS and device fingerprinting in the existing work. We will explore their effectiveness in the future work.

**Hiding the Tethering Usage:** Some tethering tools (*e.g.* tetherway [36], MyWi [1], PdaNet [4], etc.) camouflage the tethered traffic by changing packet headers, manipulating flow behaviors, or using VPN. The cost of camouflaging includes slowing down the traffic and consuming more power. Its cost will be higher as we identify more features.

In addition, some features are hard to be camouflaged, such as TCP flavors. Although we do not include this feature in our evaluation, our probability-based method is flexible and can easily incorporate new features.

**Evaluation with a Larger Data Set:** We apply our probability-based tethering detection method in a one week long campus trace. The trace includes $297,000$ flows collected from $12,600$ users at the beginning of 2013. We use the first day trace for training and trace of remaining days for testing. The tethering detection results are similar to those reported in Section 5.3: the precision is 0.86, the recall is 0.74, and the F-score is 0.8.

## 7. RELATED WORK

TCP/IP fingerprinting has been an active research area. Active probing of targeted system is adopted by [2,5,19,27,30,33,41,43]. Passive and hybrid schemes are studied as well [18, 26]. Inferring tethering via exploiting different TCP/IP fingerprinting features has been extensively studied [2, 9, 13, 28, 29, 31, 35, 39, 43]. Combining multiple features improves the inferring accuracy. The `p0f` tool [43] includes five features in TCP/IP header as its signatures in OS detection. The `Nmap` tool [2] uses a set of nine tests to detect different OSes from network packets. Further optimization techniques to combining multiple features are studied [10, 37]. Our work complements the previous efforts by (i) providing the first comprehensive quantitative study on the effectiveness of passive TCP/IP fingerprinting to OS and tethering detection; (ii) identifying new features for OS fingerprinting and tethering detection; and (iii) designing a method to effectively combine multiple features.

There are other techniques for detecting OSes or tethering activities, which utilize information in high level protocols, such as application layer features [11,24] and web browser fingerprints [8,16,17, 42]. Unlike TCP/IP fingerprinting, those techniques require Deep Packet Inspection (DPI). DPI not only has non-negligible overhead in packet processing, but also raises privacy concerns when adopted by service providers [6]. Besides, increasing adoption of encryption makes high level protocol information unavailable to use. Due to those practical issues, our study focuses on the features in TCP/IP headers.

## 8. CONCLUSION

This paper develops and evaluates a methodology that uses several features in network traffic for identifying the OS on the sending device. This OS fingerprinting can be used for detecting tethering and more generally deployment of network address translation. The proposed methodology includes a probabilistic approach to combine multiple features to enhance detection accuracy. We evaluate the effectiveness of individual features and find TTL, TCP timestamp, and TCP window size scale factor are more accurate, while clock frequency and boot time are less accurate. Furthermore, the proposed probabilistic approach significantly improves the accuracy over using individual features. It can detect iOS systems deterministically, and detect Android and Windows with 100% precision and 0.8 recall. The recall of tethering detection is 0.68∼0.85 when the target precision is 0.95, and 0.78∼0.89 when the target precision is 0.8.

## Acknowledgements

We are grateful to our shepherd Theophilus Benson and anonymous reviewers for their constructive feedback.

## 9. REFERENCES

[1] MyWi. http://intelliborn.com/mywi.html.

[2] Nmap Security Scanner. http://nmap.org/.

[3] ntpd(8) - Linux man page. http://linux.die.net/man/8/ntpd.

[4] PdaNet+. http://pdanet.co/.

[5] Snacktime: A Perl Solution for Remote OS Fingerprinting. http://www.planb-security.net/wp/snacktime.html.

[6] British Internet provider drops online tracking plans, 2009. http://goo.gl/FHtct2.

[7] CRAWDAD data set umd/sigcomm2008. Downloaded from http://crawdad.org/umd/sigcomm2008/, Mar. 2009.

[8] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: dusting the web for fingerprinters. In *Proc. of ACM CCS*, 2013.

[9] S. M. Bellovin. A technique for counting NATted hosts. In *Proc. of ACM IMW*, 2002.

[10] R. Beverly. A Robust Classifier for Passive TCP/IP Fingerprinting. In *Proc. of PAM*, 2004.

[11] J. Bi and J. Wu. Application Presence Information based Source Address Transition Detection for Edge Network Security and Management. *IJCSNS International Journal of Computer Science and Network Security*, 7(1), Jan. 2007.

[12] V. Brik, S. Banerjee, M. Gruteser, and S. Oh. Wireless device identification with radiometric signatures. In *Proc. of ACM MobiCom*, 2008.

[13] E. Bursztein. Time has something to tell us about network address translation. In *Proc. of NordSec*, 2007.

[14] R. Chandra, V. Padmanabhan, and M. Zhang. CRAWDAD data set microsoft/osdi2006 (v. 2007-05-23). Downloaded from http://crawdad.org/microsoft/osdi2006/, May 2007.

[15] CRAWDAD. http://crawdad.cs.dartmouth.edu.

[16] P. Eckersley. How unique is your web browser? In *Proc. of PETS*, 2010.

[17] E. Flood and J. Karlsson. Browser Fingerprinting. Master's thesis, University of Gothenburg, May 2012.

[18] F. Gagnon and B. Esfandiari. A Hybrid Approach to Operating System Discovery Based on Diagnosis. *Int. J. Netw. Manag.*, 21(2):106–119, Mar. 2011.

[19] L. G. Greenwald and T. J. Thomas. Toward undetected operating system fingerprinting. In *Proc. of USENIX WOOT*, 2007.

[20] S. Ha, I. Rhee, and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 42(5), July 2008.

[21] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[22] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno modification to TCP's fast recovery algorithm, 2012. RFC 6582: http://tools.ietf.org/html/rfc6582.

[23] Ingrid Lunden. Gartner: Device shipments break 2.4b units in 2014, tablets to overtake PC sales in 2015. http://goo.gl/1kUHwm.

[24] Y. Ishikawa, N. Yamai, K. Okayama, and M. Nakamura. An Identification Method of PCs behind NAT Router with Proxy Authentication on HTTP Communication. In *Proc. of IEEE SAINT*, 2011.

[25] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance, 1992. RFC 1323: http://tools.ietf.org/html/rfc1323.

[26] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring TCP Connection Characteristics through Passive Measurements. In *Proc. of IEEE INFOCOM*, 2004.

[27] A. Khakpour, J. Hulst, Z. Ge, A. Liu, D. Pei, and J. Wang. Firewall Fingerprinting. In *Proc. of IEEE INFOCOM*, 2012.

[28] T. Kohno, A. Broido, and K. Claffy. Remote Physical Device Fingerprinting. In *IEEE Symposium on Security and Privacy*, 2005.

[29] G. Maier, F. Schneider, and A. Feldmann. NAT Usage in Residential Broadband Networks. In *Proc. of PAM*, 2011.

[30] J. P. S. Medeiros, A. M. Brito, and P. S. M. Pires. A new method for recognizing operating systems of automation devices. In *Proc. of IEEE ETFA*, 2009.

[31] S. B. Moon, P. Skelly, and D. Towsley. Estimation and removal of clock skew from network delay measurements. In *Proc. of IEEE INFOCOM*, 1999.

[32] NetMarketShare. Operating system market share. http://www.netmarketshare.com/operating-system-market-share.aspx.

[33] J. Pahdye and S. Floyd. On Inferring TCP Behavior. In *Proc. of ACM SIGCOMM*, 2001.

[34] J. Pang, B. Greenstein, R. Gummadi, S. Seshan, and D. Wetherall. 802.11 user fingerprinting. In *Proc. of ACM MobiCom*, 2007.

[35] P. Phaal. Detecting NAT devices using sflow. http://www.sflow.org/detectNAT/.

[36] S. Schulz, A.-R. Sadeghi, M. Zhdanova, H. Mustafa, W. Xu, and V. Varadharajan. Tetherway: a framework for tethering camouflage. In *Proc. of ACM WISEC*, 2012.

[37] K. Straka and G. Manes. Passive Detection of NAT Routers and Client Counting. *Advances in Digital Forensics II*, 222:239–246, 2006.

[38] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound TCP approach for high-speed and long distance networks. In *Proc. of IEEE INFOCOM*, 2006.

[39] A. Tekeoglu, N. Altiparmak, and A. S. Tosun. Approximating the Number of Active Nodes Behind a NAT Device. In *Proc. of IEEE ICCCN*, 2011.

[40] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control for Fast, Long Distance Networks. In *Proc. of IEEE INFOCOM*, 2004.

[41] P. Yang, W. Luo, L. Xu, J. Deogun, and Y. Lu. TCP Congestion Avoidance Algorithm Identification. In *Proc. of IEEE ICDCS*, 2011.

[42] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *Proc. of NDSS*, 2012.

[43] M. Zalewski. P0f. http://lcamtuf.coredump.cx/p0f3/.