AppClassifier: Automated App Inference on Encrypted Traffic via Meta Data Analysis

Chong Xiang*, Qingrong Chen*, Minhui Xue†, and Haojin Zhu*
*Shanghai Jiao Tong University, Shanghai, China
†Macquarie University, Australia
Email: danco2015@sjtu.edu.cn, chenqingrong@sjtu.edu.cn, minhuixue@gmail.com, zhuhaojin@gmail.com

***Abstract*—As smart phones gradually become the dominant network traffic generators, app traffic analysis methods have gained great interests for network management and targeted advertisement. Specifically, previous works have shown that the scalability of app inference via traffic meta-data has the edge over traditional payload based analysis. However, such works mainly considered the ideal inference scenario where only one app is running on the client's device, without any background traffic noise interfered. In this paper, we extend the research to a more practical scenario, by assuming that multiple apps simultaneously run on a smart phone in the noisy background with complex traffic generated by the operating system. To that end, we propose AppClassifier, an Android app fingerprinting scheme for real-time app inference. We first leverage the observed differences in packet size distributions and traffic sequential behaviors to boost inference accuracy for noise-free traffic analysis. We then propose novel heuristic based methods to re-correct mislabeled traffic flows to realize real-time traffic inference. As a result, AppClassifier achieves inference accuracy of 82.3% for noise-free traffic analysis and reduces error rate from 66.7% to 36.4% for real-time traffic inference.**

I. Introduction

With the rapid development of the smart phone market, applications (apps) are replacing traditional browsers as the predominant network traffic generators. According to the latest research, in 2018, 52.2 percent of all website traffic was generated through mobile phones [1], up from 50.3 percent in the previous year. The mobile traffic was contributed by more than 2.8 million Android apps and 2.2 million Apple apps. The explosive increase in app usage makes smart phones desirable app fingerprinting targets for any individual or organization that aims to identify the presence of targeted apps on users' smart phones.

App fingerprinting is regarded as an important analysis tool as obtaining data about the types and usage patterns of apps running within a network is valuable. For example, knowing most popular installed apps and their throughput and latency requirements, network administrators can optimize the network deployment to improve the quality of the experience (QoE) of the users [2]. Furthermore, app usage information which has a strong correlation with the user's demographic information can be used for user profiling and facilitating market research and targeted advertisements for vendors [3].

Though the hosts of TCP/IP traffic on traditional personal computer can be easily identified by IP addresses and port numbers, app fingerprinting on smart phones is regarded as a greater challenge. First, payload encryption protocols, such as HTTPS which is widely adopted, make it less likely to identify app traffic by simply inspecting the traffic payload. Second, IP addresses based app fingerprinting may be frustrated by the widely adoption of content distribution networks (CDNs) and network address translations (NATs) which allow multiple apps to send or receive data to (and from) the same IP address or IP address range. Due to the above reasons, automated fingerprinting and identification of Android apps based on the network meta-data has been receiving increasing attention [2], [4]. The information in meta-data is usually well-formatted, thus making it feasible to automatically process and analysis network traffic. This leads to a significant advantage over traditional traffic analysis [5].

The existing meta-data based app fingerprinting approaches [2], [4] work well in an ideal setting, in which network traces are generated by launching one single app at a time. Collecting network traces in such ideal situation allows the automatic feature extraction and supports various supervised learning algorithms to achieve an accurate app identification. However, in a real-world application setting, the existing solutions may not work well when multiple apps are triggered simultaneously and their pair-wise traffic interferes. Considering the fact that the Android framework also introduces background noises, this problem is becoming even more challenging. Therefore, a novel app fingerprinting scheme which can deal with noisy traffic is highly desirable in a real-world setting.

In this study, we present AppClassifier, a novel app fingerprinting scheme for real-time app inference. AppClassifier is driven by the following two observations. *First*, the traffic from different apps tends to have different distributions on meta data (e.g., packet sizes), which results in the diversified statistical features. *Second*, different apps tend to have very different goals in the use of online services, which leads to the different sequential behaviors (or click-streams). This further enhanced the dissimilarity of the features of the different app network flows. Motivated by these two observations, we first introduce a Random Forest method [6] to learn the decision boundary among different apps in statistical feature space. Then, we capture the different sequential behaviors of various apps by introducing a novel Markov based model. Finally, to minimize the impact of background noises introduced by multiple apps, we propose a novel heuristic based method to re-correct mislabeled flows, which is inspired by the fact that the foreground traffic volume is dominant during a certain time period. Our experimental evaluation shows that AppClassifier is able to achieve an accuracy of 82.3% for the noise-free traffic inference, and reduce error rate from 66.7% to 36.4% in a real-world practical setting where an user can use multiple apps during a ten-minute time period.

In summary, our contributions are listed as follows:

- We extend our research to a more practical setting where multiple apps simultaneously run on a smart phone in the noisy background with complex traffic generated by the operating system.
- By empirical observations, we identify the apps' distinguishable statistical features and sequential behaviors.
- We propose AppClassifier, a three-phase app inference system, which
  - Phase 1: Uses Random Forest to classify statistical features.
  - Phase 2: Incorporates a novel Markov Chain to model sequential behaviors.
  - Phase 3: Finally leverages novel heuristic methods to identify and re-correct mislabeled traffic flows to realize real-time app inference.

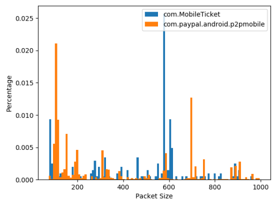
The remainder of this paper is organized as follows. We first survey related work in Section II. In Section III, we provide concrete examples about different app traffic patterns. We then propose the problem statement and system overview in Section IV. The details of system design are presented in Section V and Section VI. Finally, we show experimental evaluation in Section VII and conclude the paper in Section VIII.

II. Related Work

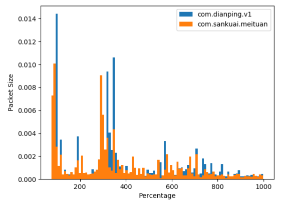
In this section, we review previous work related to traffic meta-data analysis. Most research mainly utilizes the traffic packet sizes to generate app fingerprints.

Korczyński et al. [7] developed a Markov chain model to fingerprint application traffic flows conveyed in SSL/TLS sessions. They took different protocol types in each session, which are available in meta-data, as different states in Markov model and use statistical methods to generate transition matrices. With this model, they managed to distinguish most traffic generated by 12 apps. Alan et al. [8] investigated apps from their launchtime network traffic. By collecting the first 64 packet sizes, they achieved an accuracy of 88% on 1,595 popular Android applications. Since launchtime traffic accounts for only a small portion of application generated traffic, this work suffers from its generality. A recent work [9] used end-to-end deep learning framework to automatically learn the traffic features. They collected millions of Tor network traces and achieved 96% success rate for a closed world of 100 websites.

The most similar work with ours is conducted by Taylor et al. [2]. They divided network traffic into units of flows, generated statistical features of packet sizes in each flow,

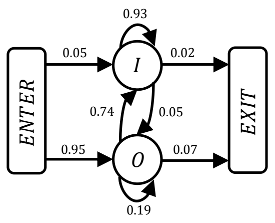


(a) Distinguishable distribution difference between MobileTicket and Paypal

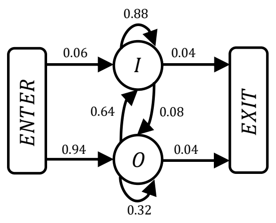


(b) Indistinguishable distribution difference between Dianping and Meituan

Fig. 1: Packet size histograms for different apps



(a) Dianping



(b) Meituan

Fig. 2: Simple transition diagrams for Dianping and Meituan

and explored several machine learning classifiers to make inference. Their further work [4] showed different devices, app versions, and time periods will severely decrease inference accuracy since app logic would change through different devices, app versions and time periods.

Although works mentioned above are argued to have high inference accuracy, they only tested their methods in the ideal setting where no background traffic noise is considered. Our work, however, mainly focuses on a real-time inference scenario and tries to address traffic noise in the wild.

III. Inferring apps via Click Stream Traffic

In this section, we perform a simple traffic analysis on the network traffic collected from two app pairs.

App Level Statistical Features. Figure 1a shows the histograms for traffic packet sizes of two different apps, `com.MobileTicket` (MobileTicket) and `com.paypal.android.p2pmobile` (Paypal). We can see noticeable differences between the two distributions. Such a difference may result from differences in app functions and communication logic. For example, traffic of video apps is likely to be dominated by large size packets due to its nature of sending great volumes of data, while chat apps tend to have traffic packet of smaller sizes since instant messages only require several bytes' space. To exploit the difference in statistical distributions, we generate the statistical features like moments and percentiles, and take them as the fingerprints for different apps.

Sequential Behavior based Features. Though noticeable distribution difference can be observed in Figure 1a, some apps may have similar distributions as shown in Figure 1b. To better distinguish between app pairs such as `com.dianping.v1` (Dianping) and `com.sankuai.meituan` (Meituan), we resort to different sequential behaviors of traffic flows. An app may have many implicit running states representing different logical functions. For example, current traffic packet directions can tell whether an app is receiving data from others or sending requests to servers. During running time, different apps can transit from states to states with different sequential manners due to different code logic and such difference can serve as fingerprints for apps. We empirically calculate the transition probability between two simple states (Incoming and Outgoing) and plot transition diagrams in Figure 2 (details about the diagrams are covered in Section V-C). As shown in Figure 2, given current packet state, I (incoming) or O (outgoing), the probabilistic distributions of direction of next packet have noticeable difference between Dianping and Meituan whose statistical distributions are quite similar. So this observation motivates us to include sequential behavior based features to boost inference performances.

Summary. As shown in Figure 1a, the network traffic of some apps (such as MobileTicket and Paypal) shows a significant difference in terms of their meta data (e.g., packet size distributions). In this case, it is reasonable to leverage the statistical features to generate the fingerprints for them. In some cases, as shown in Figure 1b and Figure 2, some apps (such as Dianping and Meituan) have similar statistical distributions, which may make it less likely to classify the apps just based on their app level statistic features. However, it is observed that traffic sequential behaviors may be distinct among these apps. Thus, we incorporate sequential features with statistical features and propose a novel app fingerprinting method.

IV. Problem Statement and System Overview

A. Problem Statement

We assume that we have access to a public Wi-Fi hotspot with which the user is connected and we are only able to passively eavesdrop the network-level traffic with meta-data, such as IP addresses, host ports, and timestamps. We argue that this assumption is reasonable because people tend to use available public Wi-Fi due to their limited mobile traffic budget. It should be noted that we cannot take the IP addresses and ports as the fingerprints of the apps due to the prevalence of CDNs and NATs which make IP addresses and ports unreliable [2]. Overall, we can only use IP addresses and ports to split traffic into units of flows and exploit meta-data like timestamps, traffic direction, and network packet sizes to generate fingerprints for different apps.

B. System Overview

As shown in Figure 3, our system mainly has four different components: traffic pre-processing, statistical features generating, sequential behavior modeling and real-time traffic re-correcting. We give a brief introduction in this section and detail on approach design in Section V and Section VI

Traffic Pre-processing. To conduct inference, we first split raw network traffic into finer units, which we call traffic flows. A flow is a subset of traffic packets appearing within a certain time threshold with the same IP addresses and network ports. Our inference is mainly conducted with respect to flows.

Statistical Feature Generation. As shown in Section III, we can exploit the difference in packet size distributions to perform app fingerprinting. To achieve that, we calculate statistical features of traffic distribution and use the Random Forest machine learning algorithm [6] to learn the decision boundary among different apps.

Sequential Behavior Modeling. Inspired by the observation in Section III, we seek ways to model the different sequential behaviors to generate app fingerprints. In our approach, we choose the Markov Chain Model [10] to depict the traffic sequence features. Also, we will incorporate the Markov Chain with Random Forest Classifier to improve inference performance.

Real-time Traffic Re-correcting. As we will show in Section VI, directly deploying aforementioned methods to real-time traffic will result in high error rates. Thus, we conduct re-correcting before our system outputs its prediction. We propose novel heuristic based rules to identify and re-correct mislabeled traffic flows. Finally, we will output the app a user is using in each second.

V. App inference with noise-free traffic

In the next two sections, we provide details about our system design. Traffic pre-processing and feature generation are covered in this section and the next section mainly focuses on real-time inference methods.

To perform real-time inference, we should first design a system which yields good performance in noise-free traffic inference. After traffic pre-processing, we first generate statistical features. Then we incorporate a novel Markov Chain to model traffic sequential behaviors to improve inference accuracy. Note that in this section, all the traffic is collected in the setting where only one app is running at a time and no traffic noise is introduced.

A. Traffic Pre-processing

To make inference on the app a user is using, we first filter out network error packets (e.g., TCP retransmission packets) and useless packets which do not carry any information about the app (e.g., ACK frames). Then we split raw traffic into bursts and flows as AppSanner [2] does. According to AppScanner, a burst is a set of traffic which satisfies the condition that the most recent packet occurs within a time threshold and a flow is a subset of a burst whose packets share the same destination IP addresses and port numbers. After traffic pre-processing, we get traffic flows which are sequences of packet sizes and timestamps. Note that our goal in noise-free traffic inference is to label each flow to its corresponding app.

B. Statistical Feature Generation

Since different apps tend to yield different distributions of packets sizes, as shown in Section III, we can generate features depicting different distributions. The most straightforward

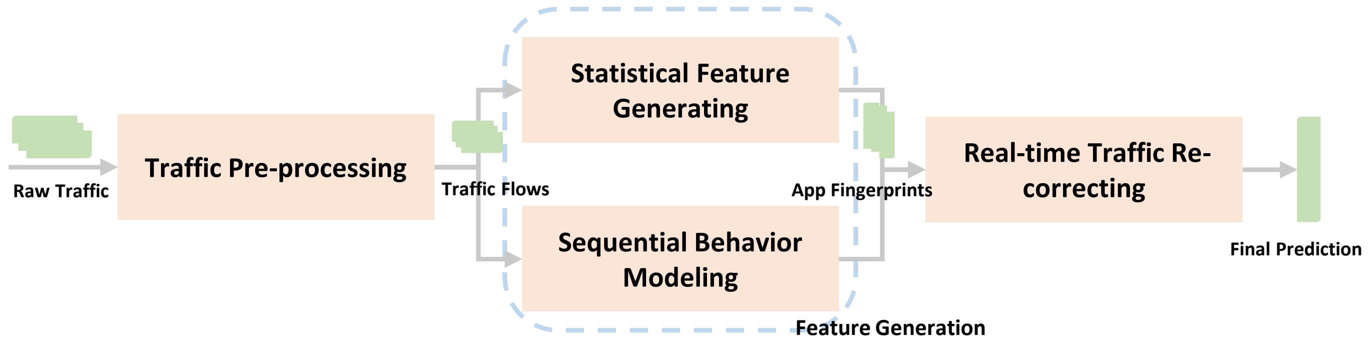


Fig. 3: System overview

way is to calculate the statistical quantities of packet size sequence as features. We adopt statistical features proposed by AppScaner [2] as well as some related features which show empirical benefits. Those features mainly include different orders of moments and different percentiles. The top 35 statistical features and their corresponding importance scores is provided in Table II (see Appendix).

After generating these features, we can feed them to a machine learning model to learn the decision boundary of each app. According to the existing researches [6], Random Forest outperform other algorithms in [2], and thus we implement a one-vs-all Random Forest model to perform multi-class classification.

C. Sequential Behavior Modeling

As shown in Section III, apps with similar statistical features may have a significant difference in sequential behaviors. Thus, we take apps' sequential behaviors into consideration to improve system performance. There are many existing methods to model a time sequence, such as HMM [10] and RNN [11]. In our system, we choose to incorporate a first-order homogeneous Markov Chain due to its simplicity and effectiveness for capturing the sequential behavior of users on apps.

Similar to [7], we use a discrete time variable X_t, which takes values $s_i \in \{-M, \dots, -1, 0, 1, \dots, M\}$, to denote the t^{th} state of traffic sequence. Here, s_i refers to the size of a packet and M denotes the possible maximum packet size. The s_i with a positive value means the packet is sent by the smartphone while a negative value means the packet is sent by the sever. We assume the Markov Chain variable X_t is first-order (1) and homogeneous (2). Formally, our assumption can be formulated as equations followed:

$$\begin{aligned} P(X_t = s_t | X_{t-1} = s_{i-1}, &X_{t-2} = s_{t-2}, \dots, X_1 = s_1) \\ &= P(X_t = s_t | X_{t-1} = s_{i-1}), \end{aligned} \tag{1}$$

$$P(X_i = s_i | X_{t-1} = s_{t-1}) = P(X_t = j | X_{t-1} = i) = p_{i \to j}. \tag{2}$$

Considering that there may be too many possible states (i.e., packet sizes) in the aforementioned Markov model, which may incur a great computation overhead. We reduce the number of states by dividing the possible packet size domain into intervals with the same length. In this case, we take the interval a packet size belongs to as the state of that packet. Formally, X_t takes values $s_i \in \{\lfloor -M/l \rfloor, \dots, -1, 0, 1, \dots, \lfloor M/l \rfloor\}$, where l is a parameter denoting the length of each interval. In our experiment, M and l are set to be 1600 and 5, respectively.

Besides normal states of packet sizes, we also include two special states, ENTER (shorthand EN) and EXIT (shorthand EX), to provide more information about the flows in Markov Chain. The ENTER probability distribution represents the probabilities that a flow starts at different packet sizes. Similarly, the EXIT probability distribution represents the probabilities a flow ends at different sizes. As shown in Fig 2, each flow begins at ENTER state, transitions among different packet size states and finally ends at EXIT state. Combining the ENTER and EXIT probability distribution with the normal transition matrix, we can generate all the probability needed in our inference. Using these probabilities, we can calculate the possibility of the occurrence of flow $\{X_1, \dots, X_T\}$ using Equation (3):

$$P(\{X_1, \dots, X_T\}) = q_{EN \to s_1} \times \prod_{t=2}^{T} p_{s_{t-1} \to s_t} \times p_{s_T \to EX}. \tag{3}$$

To generate the transition matrices for Markov Chain, we use Maximum Likelihood Principle and calculate the probability of all possible transitions observed in the training dataset. Note that we have to pay attention to the possible situations that some possible transitions may not be observed in our training set, which lead to a zero transition probability. To handle with this problem, we use Laplace smooth method when calculating the empirical probability. Formally, we calculate the transition matrix elements using the formula:

$$p_{i \to j} = \frac{k_{\text{times}}^{ij} + a}{T_{\text{total}} + a \cdot n},$$

where k_{times}^{ij} refers to the time a transition from state i to state j observed in our training data, a is a self-chosen parameter as the degree of smooth process, T_{total} refers the total number of transition in the dataset and n refers to the number of possible states. Note that i and j can also be the special states ENTER and EXIT. In our experiment, we set a to be 0.1.

We generate one matrix for one specific app, and we will get N different matrices after our training, where N is the number of apps in the training set. After getting the transition matrices, we can perform app inferences on the test data: given an unlabeled flow, we calculate the probability of the occurrence of this flow using N different transition matrices. N probabilities will be obtained and we will pick out the highest probability to label the flow as its corresponding app.

D. Combination of Statistical Features and Sequential Behaviors

As discussed in Section V-B and Section V-C, we can take either statistical features or sequential behaviors as the fingerprints for apps. Our observations also show that traffic flows with similar statistical features may differ greatly in sequential behaviors. Therefore, to make the fingerprint more robust, we propose to combine these two types of traffic features. We first calculate flows' statistical features and then use Markov Chain Model to calculate the probabilities the flow belongs to N different apps. After that, we feed statistical features as well as N probabilities to the Random Forest classifier to train a model considering both statistical features and sequential behaviors. We will demonstrate the effectiveness of this method on app classification via extensive experiments.

VI. Real-time App Inference

In the previous section, we have presented the details of the inference methods in the noise-free traffic setting. In this section, we will discuss the difference between app inference in an ideal setting where no noise is involved and a real-time setting where background traffic is present and show how to perform a real-time app inference using novel heuristic re-correcting methods.

A. The Challenges in Real-time Traffic Inference

Unlike the previous works where all the traffic flows are collected in the ideal setting where only one app is running at a time and no noise is introduced, real-time inference has to directly deal with the raw network traffic which contains a lot of background noises. Though in most cases only one app is running at the front, background apps and Android framework can also generate noise traffic. To show how these background noises interfere with app inference, we run a simple experiment where a user randomly switches the apps for every five minutes. A 250-minute experiment shows that directly using state of art inference methods will label 49% flows as apps which have never been used in the experiment. Therefore, a new method is highly desirable for real-time app inferences.

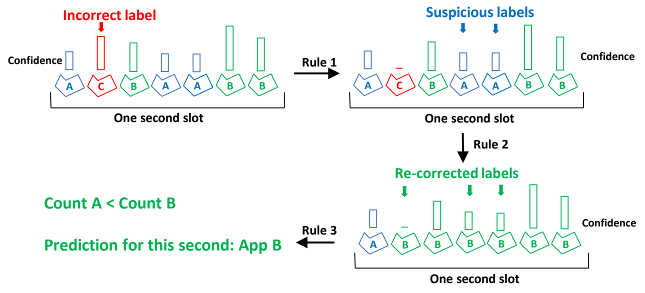


Fig. 4: An example of real-time traffic re-correcting. Firstly, the confidence of Flow C is reduced to zero according to Rule (i). Secondly, Flows A in the middle are regarded as suspicious ones according to Rule (ii) and then both Flows A and C are re-corrected as flow B. Lastly, the system output B since B's traffic volume is dominant in this time slot. Note that without re-correcting rules, the system will fail to determine the running app for this second.

B. Dealing with Background Traffic Noise

To overcome the challenges in real-time inferences, we propose a novel heuristic scheme based on the flow re-correcting rules to identify and re-correct mislabeled flows.

Our re-correcting rules mainly stem from the observations that the background traffic volume is usually significantly smaller than that generated by the front app and that users are not likely to switch apps too frequently. From these observations, we have the priors knowledge that, in most cases, the current flow captured is likely to belong to the same app as the previous flow and that the flows whose labels appear frequently within a time period are likely to be correctly labeled.

Before giving the details of our re-correcting methods, it is important to point out that the aforementioned models will return app labels and classification confidence as their outputs. We will exploit the information provided by these outputs to perform traffic re-correcting. Besides, since we assume the user only uses one app at one time, we only output the app that the user is using during each fixed-length time slot (each second in our experiment). Also note that the traffic flows have been sorted according to their time stamps before our label re-correcting. Our re-correcting rules are demonstrated in Figure 4 and detailed as followed:

(i) If no flow in a flow's neighborhood is labeled with the same app, we assume this flow is incorrectly labeled. Formally, if the i^{th} flow is labeled as app A, we examine the $(i-k)^{th}$ to the $(i+k)^{th}$ flows, where k is a self-chosen parameter, to find out if any flows in this interval is labeled with app A. If not, we assume that the i^{th} is incorrectly labeled and reduce its confidence into zero. We set k to be 6 for experiments.

(ii) If a flow is labeled with low confidence while one of its neighbors is confidently labeled to a different app, we assume the former flow is incorrectly labeled and should be re-corrected to the high-confidence label. Formally, for any two different flows f_i and f_j at the i^{th} and j^{th} position respectively, if the following inequation (4) is satisfied, we suppose the i^{th} label is suspicious and replace the label of f_i with the label of f_j.

$$\frac{C_j - C_i}{|j - i|} > T_{\text{thre}}, \tag{4}$$

where C_i and C_j refer to the classification confidence values of the f_i and f_j and T_{thre} is a self-chosen parameter indicating the degree of re-correcting (we choose 0.1 for experiments). Note that in Rule **(i)**, we already reduce the confidence values of mislabeled flows to zero. Therefore, those mislabeled flows identified by Rule **(i)** can also be re-corrected by Rule **(ii)**.
(iii) We only output the label of apps whose traffic volume is dominant within a specific time slot. Formally, In a time slot of t seconds, we calculate the traffic volume for different apps. We only retain flows labeled to the app with the largest traffic volume and finally output the app label. We set t to be 1 for experiments.

In our later evaluation, we will demonstrate that our filter method can receive good results.

VII. Evaluation

In this section, we implement all the methods proposed in previous sections and perform a comprehensive evaluation on our system performance.

A. Traffic Collection

We use a Google Nexus smartphone running on Android 6.0.1 as our experiment device. We randomly download 48 most popular apps from China Android market and 17 apps from Google Play Store. The smartphone is connected with a laptop hotspot so that we can capture all the network traffic using Wireshark [12].

To gain good performance in the real-time inference, we have to make sure that the training traffic flows are similar with those in the real-time inference scenarios. Unfortunately, our experiment finds that automated UI fuzzing tools such as Monkey [13] cannot simulate human's behavior because their can only execute random operations. So we choose to run apps manually instead. Though our system need manual operation, we would argue that the scalability won't be a problem because each app requires only ten-minute operations to generate enough training traffic. We believe such cost of time is trivial compared with the potential benefits obtained by meta-data analysis. Besides, we can also resort to on-demand workforce marketplace like Amazon Mechanical Turk [14] to obtain adequate traffic data.

B. Noise-free Traffic Inference

We first implement our system in an ideal setting where no traffic noise is introduced to show the performance improvement with our newly introduced methods. We run each apps for ten minutes to generate enough traffic flows for inference. 63,258 flows from 65 apps are collected and randomly split into training set and test set by a proportion of 8 to 2. Experiment results for noise-free traffic inference are shown in Figure 5a. As shown in Figure 5a, both statistical features and sequential behaviors can serve as fingerprints for different apps, achieving inference accuracy of 80.0% and 73.4%, respectively. As expected, our combined method yields the highest accuracy of 82.3%, which justify our attempt to consider both distribution statistical features and sequential behaviors. This improvement in the ideal setting will also help implement real-time traffic inference.

C. Real-time Traffic Inference

To simulate real-time inference scenarios, we randomly choose five different apps and switch among these apps during running time. In the meanwhile, we also record the time of app switching. We eavesdrop the network traffic to make inference about which apps is being used in each second.

We perform 25 times real-time inference experiments, each lasts for around 10 minutes, and use the re-correcting rules detailed in Section VI to identify and re-correct mislabeled flows. To evaluate the re-correcting performance, we calculate the error rate for the app inference in each second. The results of our system for different running times are shown in Table I. Comparing the performance before and after re-correcting, we can see our re-correcting rules work quite well and correct more than half of the incorrect flows.

TABLE I: Real-time inference error rate under different inference methods and running times

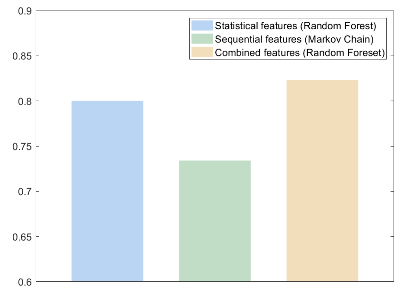
Inference method / **Running time**	Without re-correcting	With re-correcting
4 minutes	63.9%	33.8%
6 minutes	63.8%	33.2%
8 minutes	64.5%	35.1%
10 minutes	64.6%	34.9%

To better understand how re-correcting rules works, we visualize inference results with and without first two rules of re-correcting in Figure 5b and Figure 5c respectively. The figures demonstrate the cumulative flow packet sizes changes in ten minutes and different curves represent flow changes of different app labels. We only plot the curves for apps with top 10 total packet sizes in the figures. As shown in the figures, inference result of our original system without re-correcting is a little noisy and has many incorrect labels while after applying re-correcting rules, few incorrectly labeled flow is presented and we can clearly identify the app the user is using during each second.

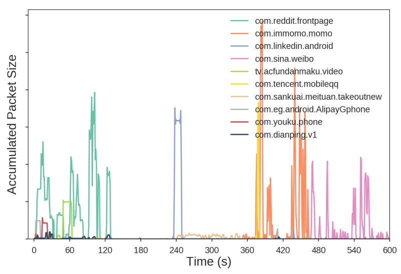
VIII. Discussion and Conclusion

One challenge we fail to overcome in this paper is the impact of different devices as mentioned in [4]. The apps' logic changes over time and hinders inference accuracy. Potential improvements also include implementing other methods, such as reinforcement machine learning or second-order Markov chain to gain an even higher accuracy. We would leave this to the future avenue.

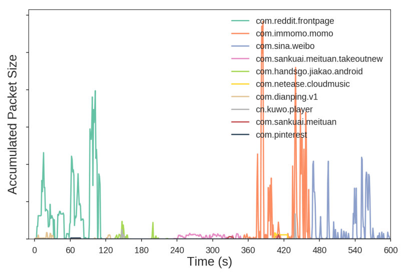
This paper attempts to explore the feasibility to perform real-time inference on running apps via traffic meta-data in



(a) Experimental results of noise-free traffic inference



(b) Real-time inference without re-correcting



(c) Real-time inference with re-correcting

Fig. 5: Experimental results for APPCLASSIFIER

the wild. In doing so, we first explore different traffic patterns, and then leverage both different statistical features and traffic sequential behaviors to fingerprint different apps. Using newly proposed methods, we boost app inference accuracy for noise-free traffic analysis. Furthermore, we examine real-time app inference where traffic from different apps and Android framework can interfere. To address this, we also propose heuristic based re-correcting rules to identify and re-correct mislabeled flow caused by background traffic noise. Our experiment has shown that the re-correcting rules perform well in real-time scenarios.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China (No. 61672350, U1401253).

REFERENCES

[1] "Share of mobile phone website traffic worldwide," https://www.statista.com/statistics/241462/global-mobile-phone-website-traffic-share/, 2018.
[2] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 439–454.
[3] E. Malmi and I. Weber, "You are what apps you use: Demographic prediction based on user's apps." in *ICWSM*, 2016, pp. 635–638.
[4] V. F. Taylor, R. Spolaor, I. Martinovic *et al.*, "Robust smartphone app identification via encrypted network traffic analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 1, pp. 63–78, Jan 2018.
[5] H. Li, H. Zhu, and D. Ma, "Demographic information inference through meta-data analysis of wi-fi traffic," *IEEE Transactions on Mobile Computing*, vol. 17, no. 5, pp. 1033–1047, 2018.
[6] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
[7] M. Korczyński and A. Duda, "Markov chain fingerprinting to classify encrypted traffic," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. IEEE, 2014, pp. 781–789.
[8] H. F. Alan and J. Kaur, "Can android applications be identified using only tcp/ip headers of their launch time traffic?" in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, 2016, pp. 61–66.
[9] V. Rimmer, D. Preuveneers, M. Juarez, T. Van Goethem, and W. Joosen, "Automated website fingerprinting through deep learning," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.
[10] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.
[11] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.
[12] "Wireshark," https://www.wireshark.org/.
[13] "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey.html.
[14] "Amazon mechanical turk," https://www.mturk.com/.

APPENDIX

TABLE II: Statistical features and importance scores

Rank	**Feature**	**Score**
1	Overall Maximum	0.041522
2	Outgoing Maximum	0.040859
3	Position in FlowFirst or not	0.038741
4	Outgoing 90% Percentile	0.035219
5	Outgoing 80% Percentile	0.030555
6	Outgoing 70% Percentile	0.030147
7	Outgoing 60% Percentile	0.030048
8	Outgoing Minimum	0.029528
9	Outgoing 50% Percentile	0.028771
10	Outgoing 10% Percentile	0.028614
11	Outgoing 20% Percentile	0.027693
12	Outgoing 30% Percentile	0.027122
13	Outgoing 40% Percentile	0.026321
14	Outgoing Mean	0.026216
15	Overall Minimum	0.022138
16	Incoming Maximum	0.021378
17	Incoming Minimum	0.020818
18	Incoming 10% Percentile	0.017286
19	Incoming 90% Percentile	0.014936
20	Incoming 20% Percentile	0.014398
21	Overall 10% Percentile	0.014271
22	Incoming Mean	0.013302
23	Outgoing Median Absolute Deviation	0.013164
24	Outgoing Standard Deviation	0.013092
25	Outgoing Variance	0.012890
26	Overall Standard Deviation	0.012879
27	Overall Variance	0.012520
28	Elapsed Time	0.012038
29	Incoming 30% Percentile	0.011949
30	Incoming 40% Percentile	0.011603
31	Incoming 80% Percentile	0.011387
32	Incoming Median Absolute Deviation	0.011111
33	Incoming Kurtosis	0.010954
34	Overall Mean	0.010916
35	Overall Median Absolute Deviation	0.010873