Automatic Generation of Mobile App Signatures from Traffic Observations

Qiang Xu†, Yong Liao‡, Stanislav Miskovic‡, Z. Morley Mao†, Mario Baldi‡, Antonio Nucci‡, Thomas Andrews†
†University of Michigan, ‡Symantec, Inc.

***Abstract*—There are network management, traffic engineering, and security practices adopted in today's networking that rely on the knowledge about what applications' traffic is passing through the networks. These practices might fail with mobile apps whose identity remains hidden in generic HTTP traffic. The main reason is that unlike traditional applications, most mobile apps do not use specific protocols or IP ports with distinctive features. Many enterprises and service providers are in a great need of regaining control over their networks that increasingly carry mobile traffic. In this paper we propose FLOWR, a system that automatically identifies mobile apps by continually learning the apps' distinguishing features via traffic analysis. FLOWR focuses solely on key-value pairs in HTTP headers and intelligently identifies the pairs suitable for app signatures. Our system employs a custom supervised learning approach that leverages a very limited knowledge of app-signature seeds and autonomously grows its capacity for app identification. The approach is motivated by a simple but effective hypothesis that unknown app-identifying features should co-occur with the known signatures. Our experimental results show a significant growth in flow identification coverage provided by FLOWR. Specifically, we show that FLOWR can achieve identification of 86–95% of flows related to their generating apps.**

I. Introduction

The rapid adoption of mobile devices has dramatically changed the access to various networking services: instead of using web browsers, mobile users increasingly choose mobile applications (simply "apps") as preferred interfaces to the Internet [1]. Consequently, many stakeholders are becoming interested in app identification. For example, app market providers would benefit from knowing real app usage and promoting the apps accordingly; users can be offered content based on the interests inferred from the app usage; network operators may optimize their resources for apps; enterprise departments can restrict the access to selected apps, thus enabling BYOD (bring your own device) without compromising the security of corporate networks. The problem is that apps are very difficult to identify, appearing just as a generic HTTP traffic.

Traditional approaches to identification of desktop applications and corresponding protocols (e.g., email, news, and VoIP) [2]–[8] and for discerning P2P traffic [9] are too coarse-grained for mobile apps. Such approaches cannot differentiate apps that communicate through generic application-layer protocols (e.g., HTTP) or contact the same content servers (e.g., cloud services and content delivery networks). State-of-the-art solutions exist to generate signatures for small numbers of mobile apps either through user studies or app emulation [10]–[13]. However, such approaches do not come even close to classifying individual traffic flows generated by hundreds of thousands of potential apps in real time. An effective approach to real-time app identification at scale would thus have to address the following challenges.

- **Similarity.** HTTP is the protocol of choice for many app developers to implement communications with remote servers [14]. Moreover, it is common for several apps to contact the same servers due to the widespread use of content delivery networks (CDN) and cloud services. Hence, conventional approaches based on hostnames and transport ports fail.
- **Scale.** Hundreds of thousands of apps being available in app markets make any manual or small-scale efforts to app identification not scalable. Also, relying heavily on supervised learning to generate app signatures is impractical, because even providing a sufficient training set is a challenge. Moreover, complicated stateful signatures based on information gathered across multiple app flows do not scale well when performing signature matching.
- **Coverage.** As observed in previous research [14], the top 5K apps contribute 98% of the mobile traffic volume. However, the known sophisticated signature generation methods for these apps [15] do not apply to less popular apps. Identifying these apps is however crucial because high app-identification coverage is necessary for security applications as well as for handling the highly volatile nature of app popularity [16].

To address these challenges, we are guided by several common behaviors of mobile apps. First, at least some flows generated by the apps should include some sort of *distinctive information*. For example, some of the apps may be related to a specific web service. In other cases, developers may include specific and a priori unknown information in the apps that communicate with CDNs. This would serve as means to distinguish the apps. It is up to the app-identifying algorithms to discover such information automatically. Secondly, app identification can benefit from temporal and topological structure of the traffic, e.g., flows repeatedly observed from the same devices within short time intervals are likely to come from the same apps.

Starting from the above observations, we developed a unique system that automatically discovers app signatures via on-line traffic analysis. Our solution is based on examining

This research was conducted under the Narus Fellow Research Program.

ties between the unclassified flows and a very small set of seeds known to originate specific apps. When the occurrence likelihood of these ties is sufficiently high, the features extracted from the unclassified flows can be promoted to new app signatures. We implemented this methodology in FLOWR (FLOW Recognition). FLOWR is a self-learning system requiring minimal supervised training. Once provided with a small set of initial app signatures, the system can operate completely autonomously and grow its signature knowledge. Consequently, the identification coverage grows consistently over time.

As a result, FLOWR scales automatically to the size and growth of entire app markets, each containing hundreds of thousands of apps. Moreover, the stateless nature of our signatures supports app identification at real time: Using an off-the-shelf machine, we achieved comprehensive identification at speeds of up to 5 Gb/s of input traffic. We also devised a methodology that evaluates false positives with a very limited ground truth provided by a priori known app identifiers (seeds). In a 6 day 10 billion flow trace from a nationwide cellular network, FLOWR was capable of identifying 86–95% of flows related to the signature seeds with "tolerable" false positives. In contrast, only 3% of such flows could be identified without FLOWR.

The rest of this paper is organized as follows. Section II uses a few examples to illustrate the problem and motivate the development of FLOWR. The methodology is presented in Section III. Section IV discusses some of the limitations of FLOWR in its current form. Tuning of important parameters of FLOWR is addressed in Section V. FLOWR is evaluated in Section VI and related work is surveyed in Section VII. Section VIII concludes this paper.

II. Motivating Examples

We use several examples to illustrate the app identification problem. We start by showing that mobile apps in general have seed *identifiers* that are uniquely related to them. For instance, popular Facebook Android app has an identifier "com.facebook.katana", which uniquely identifies the app in the Google Play market as can be checked via https://play.google.com/store/apps/details?id=com.facebook.katana. Fig. 1 shows an example of another seed, where a unique app identifier "zz.rings.rww2" is included in HTTP message sent from its corresponding app. We refer to any *individual* app identifier that is unique and appears in the network traffic as *app signature*. The signatures can be known a priori or discovered by a system like FLOWR.

```
GET /pagead/images/go_arrow.png HTTP/1.1
Host: pagead2.googlesyndication.com
Referer: http://googleads.g.doubleclick.net:80/&...
&msid=zz.rings.rww2&...
User-Agent: Mozilla/5.0 (Linux; U; Android 2.3.3; ...
```

Fig. 1. Unique app identifier is embeded into the "Referer" field of a HTTP message sent from an Android app "zz.rings.rww2".

An approach to app identification could be just to know few patterns of how seed identifiers occur in the traffic. For instance, one could know that a key "msid=X" always carries values that are app identifiers, as shown in Fig. 1. This pattern alone could identify numerous apps. However, applying this methodology to a *huge number of* mobile apps has several challenges. First, there are numerous ways for each individual app to embed its identifiers in traffic. The same app sending out a flow like the one shown in Fig. 1 can send a completely different flow to some other web service, which does not have the key-value pair "msid=X" at all. Moreover, it is simply not feasible to manually examine all existing apps for such behaviors. This makes both flow- and app-identification coverages questionable when one knows only a limited set of the app-identifier embedding patterns. Secondly, it is even not feasible to gather a priori knowledge of all possible unique app identifiers. We show such an example in Fig. 2, where the URL field of the GET message has a key-value pair "sdkapid=67526". It *could possibly be* a proprietary identifier used by service mydas.mobi to identify some app, although we cannot be sure about its uniqueness a priori.

```
GET /getAd.php5?sdkapid=67526&...&country=US
&age=45&zip=90210&income=50000&... HTTP/1.1
Connection: Keep-Alive
Host: androidsdk.ads.mp.mydas.mobi
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)
```

Fig. 2. Potential unique app identifier embedded into a key-value pair in the URL of a HTTP GET request.

To address these challenges, this paper studies the problem of *automatically learning the mobile app signatures included in network traffic, and assessing the quality of the learned signatures*.

III. Methodology

FLOWR does automated learning of app identities: it grows app signature knowledge from a very small set of a priori known signatures. Such initial signatures exist and can be easily identified, as we confirmed by a widely known doubleclick.net signature in HTTP referer fields that points to apps via "msid=X" parameter. We grow the set of known signatures by observing the co-occurrence of temporally close flows and features in them. This methodology is based on a simple intuition: if two types of events are intrinsically related, one should be able to observe them co-occurring repeatedly over time; on the other hand, if the events are not related, they may co-occur occasionally, but such co-occurrence should only be transient.

Let F_X be a set of flows that readily match a signature of app X (either an initial or a learned signature). Then, for each flow f_i in F_X, there will be some flows that come from the same source IP address and co-occur with f_i closely in time. We denote such flows as $\hat{f}_i$ and deem them likely to be from app X. Accordingly, let $\hat{F}_X$ be the union of $\hat{f}_i$ for each f_i in F_X. If some feature $\mathcal{F}$ repeatedly appears in flows belonging to set $\hat{F}_X$, and $\mathcal{F}$ rarely appears in any other $\hat{F}_Y$ ($Y \neq X$), $\mathcal{F}$ is likely to be a signature of app X as well. This is how we learn new app signatures.

The challenge to this approach is that different apps can have their flows mixed in the traffic coming from the same source IP addresses. This is due to multiple mobile devices

being tethered, or mobile OS multi-tasking its apps, or due to NAT, etc. We leverage the huge amount of diverse traffic available in mobile networks to cancel the noise cases. Over time, only the features coming from the same app should persistently co-occur in many instances.

A. App Features and Signatures

FLOWR leverages metadata information in HTTP headers to create app features and consequently signatures. We rely on HTTP because it is the predominant protocol adopted by mobile apps. Our features are individual key-value pairs exchanged in HTTP queries. We analyze such pairs per each distinct HTTP host service (i.e., HTTP hostnames or HTTP referer hostnames). This type of "feature plus host service" analysis is based on the fact that meanings of key-value pairs are tied to specific web services.

As shown in examples in Fig. 1 and 2, the "query" part of HTTP URIs usually contains a rich set of information in the form of key-value pairs[1].

For practical purposes, FLOWR keeps only the meaningful parts of service names as a sufficient indicator of the web services in most cases. Usually, this is two or three right-most labels in fully qualified domain names. We show in §VI-A that this type of features provides a sufficient amount of information to identify apps.

Definition III.1. *An app **feature** is a concatenation of the name of a web service employed by the app and a key-value pair in the query part of the service's HTTP URI, i.e.* $\mathcal{F} = \{name : K = V\}$.

Definition III.2. *An app feature $\mathcal{F}$ that identifies app X with good confidence is a **signature** of app X.*

Let's exemplify some features and their subset of designated app signatures: In Fig. 1, "doubleclick.net:msid=zz.rings.rww2" is an app feature. It is also a signature of the app whose id is "zz.rings.rww2" because it uniquely identifies the app. In Fig. 2, "mydas.mobi:sdkapid=67526" is an app feature which may be a signature due to its structure suggesting that it may be an app ID. On the other hand, "mydas.mobi:country=US" is also an app feature, but it is unlikely to be a signature of any app.

B. Counting Co-occurrence of App Features

The basic idea of flow regression, our key method for producing app signatures, is based on counting the "co-occurrences" of app features. Let's consider two app features, $\mathcal{F}_1$ and $\mathcal{F}_2$, from two different flows, $flow_1$ and $flow_2$, generated by the same source IP address. If start time of $flow_2$ is less than T seconds after the end time of $flow_1$ (assuming $flow_1$ starts before $flow_2$), we consider this event as a co-occurrence instance between the flows' features $\mathcal{F}_1$ and $\mathcal{F}_2$.

[1]A URI's syntax is standardized as follows: scheme://authority/path?query\#fragment [17], where the `scheme` is HTTP and the `authority` refers to the contacted host name. The `query` consists of a sequence of key-value (KV) pairs in the format of `k1=v1&k2=v2&...`.

Furthermore, let's assume that feature $\mathcal{F}_1$ is a signature of app X. Then if $\mathcal{F}_2$ persistently co-occurs with $\mathcal{F}_1$, we infer that $\mathcal{F}_2$ is likely to be another signature of app X. We introduce the concept of *co-occurrence likelihood*, denoted as $P[X|\mathcal{F}]$, to quantify whether a given feature $\mathcal{F}$ co-occurs persistently with known signatures of app X.

Definition III.3. *Feature $\mathcal{F}$'s **co-occurrence likelihood** with app X is defined as a ratio of the number of unique IP addresses for which feature $\mathcal{F}$ co-occurs with app X's signatures, and the total number of IP addresses in which $\mathcal{F}$ can be observed.*

The reason for FLOWR to rely on unique IP addresses for counting the co-occurrence events is that various types of bias could occur otherwise. For example, the features that appear very frequently would easily bias the computation of co-occurrence likelihood. Such features are often present in advertisement and analytics services (referred to as *a* services*) and "fired" frequently from apps by the embedded ads and analytics libraries. Similarly problematic would be the counting based on time windows, where all co-occurrences within a certain window would be counted only ones. This method remains biased towards the apps that keep running in the background, e.g., `com.accuweather.android`. These background apps would then appear in many time windows, resulting in their features co-occurring many times with signatures of potentially different apps.

C. Flow Regression

Flow regression is a process in which FLOWR identifies app features suitable for app signatures. For a given feature $\mathcal{F}$, FLOWR consistently updates the feature's co-occurrence likelihoods with all apps already having known signatures in our system. Together with this updating, FLOWR evaluates the quality of the feature $\mathcal{F}$ to become a signature. The promotion of a feature to a signature can be two fold: First, if $\mathcal{F}$'s co-occurrence likelihood with app X is significantly higher than the likelihoods with other apps, $\mathcal{F}$ should obviously be promoted to a new signature of app X. Secondly, if $\mathcal{F}$ has high co-occurrence likelihoods with a set of apps, it should become the signature for all apps in the set. This helps reducing ambiguity when the exact apps cannot be identified. For instance, feature $\mathcal{F}$ can be a developer identifier included in several apps published by the same author. Then, using $\mathcal{F}$ as a signature significantly focuses our identification – as opposed to being unable to attribute the traffic to any particular apps in the entire existing app universe.

We can address the potential over-fitting issue in computing co-occurrrence likelihoods by using higher threshold on the number of samples used in the computing. In practice, considering the large traffic volume in a real world mobile service provider's network, FLOWR can easily gather enough samples to compute the co-occurrrence likelihoods.

D. Seeding the Knowledge Base

FLOWR needs an initial set of seeding app signatures to bootstrap the learning of new ones. Although the seeding signatures are not the focus of this paper, we provide some explanation on how to get a good set of such signatures in order to maximize FLOWR's gains.

A good set of seeding signatures is the one that provides wide app coverage. That is, the signatures should identify at least some flows, but from a very large number of apps. We learned that many free mobile apps employ a* services. The traffic to these services often includes elements feasible for seeding signatures, i.e., the unique names of apps or names of app packages published in mobile markets (e.g., com. instagram.android). The situation is largely similar for paid apps: While these apps may not use advertisement services, most of them still rely on analytics to track app usage and provide troubleshooting data collection.

Examples of a* services that embed app identification parameters include doubleclick.net, admob.com, airpush.com, smaato.com, etc. Studying the corresponding app coverage of these individual services, we found that doubleclick.net is present in almost half of the free Android apps in our traffic traces. Hence, one can manually reverse engine a few popular a* services to study the patterns of how app identifiers are embedded in the flows sent from apps to these services. The patterns can serve as a good set of seeding signatures to FLOWR.

IV. Limitations

FLOWR has problems in identifying encrypted or hashed network traffic or traffic originated by apps that do not use a* services. Here, we discuss these limitations and work-arounds.

A. Apps Using Protocols Other Than HTTP

In its current form, our FLOWR system processes only HTTP traffic. This is a design decision based on: (1) HTTP is the preferred protocol for the majority of mobile apps; and (2) HTTP headers usually provide sufficient app identification information. Thus, FLOWR needs to examine only the initial packets of each HTTP flow and parse only their headers without digging deep into the privacy-sensitive HTTP payload.

We strongly believe that FLOWR's basic methodology of tracking co-occurrence for signature building is generic and can be universally applied to apps that use other protocols. Choosing the right features for non-HTTP protocols (instead of using $\{name : K = V\}$ features currently employed by FLOWR) needs to be carefully studied.

An especially challenging case are the apps that fully encrypt all their traffic. Due to the randomness of encryption, it would be difficult to find any meaningful signatures for these apps. At best, one could rely on the host names or SSL certificates employed by the apps as features that hint app identity. However, we are uncertain how effective such solutions would be. Thus, if needed, FLOWR can be deployed together with other techniques providing visibility into the encrypted traffic, such as the man-in-the-middle tools for HTTPS. Those deployment related issues are not the focus of our study, and are out of the scope of this paper.

B. Coverage Bounded by Initial Seeding Signatures

It is very important that the initial set of supplied app signatures covers a large number of apps. Via its self-learning logic, FLOWR can then discover many new signatures that cover more flows generated by the apps. However, the set of apps that can be identified is usually bounded to the apps covered by the seeds.

Due to the pervasive adoption of a* services by mobile apps, we believe that it is feasible to acquire a good set of seeding app signatures with minimal effort, such as manually reverse engineer a few popular advertisement services, or conduct static byte-code analysis of app binaries [18].

It is worthwhile to note that in some cases FLOWR can learn signatures for "new" apps beyond the ones covered by the initial signature set. For instance, FLOWR can leverage flows of a free app A sent to a* services and learn new A signatures. Then, the learned signatures may also be applicable to the paid counterpart of A, because paid apps usually have similar functionality and communications except the ones to a* services.

V. System Tuning

FLOWR system needs proper calibration to realize the full potential of the methodology presented in §III. To this end, we employ two datasets to tune key system parameters. Our first dataset (FlowSet) is a network trace from a nationwide cellular network provider. The second dataset (AppSet) is a lab trace generated by running more than 10K most popular Android apps in software emulators. More details about the datasets can be found in Appendix A.

A. The Size of Time Windows for Co-occurrence Analysis

One of the key parameters of FLOWR is the co-occurrence time window T. The window largely impacts the set features that would be promoted to app signatures. To understand the importance of tuning the window size, we note that underestimating T results in missing some valid co-occurrence events, and thus reducing the system's app identification capabilities. On the other hand, if T is over-estimated, FLOWR is more likely to mix flows from different apps, thus inducing noise and over utilizing system resources. Given that FLOWR is capable of removing the noise by repeated validation of co-occurrence events for different users and time periods, we are inclined to tolerate slight over-estimation in order to prevent serious implications of underestimating T.

To evaluate an appropriate window size, we first identify flows (from our FlowSet) that match the seeding app signatures. Particularly, we rely on the flows sent to doubleclick.net and containing the key-value pair (signature) "msid=X". We take a brute-force approach in measuring T and examine *all* features co-occurring with the doubleclick seeds. Specifically, for each feature $\mathcal{F}$, we find the temporally closest doubleclick.net flow coming from the same user (source IP

address). We keep only the features that consistently co-occur with doubleclick.net flows pointing to the same apps. Finally, T estimations are measured as the intervals between these flows and their doubleclick.net references.

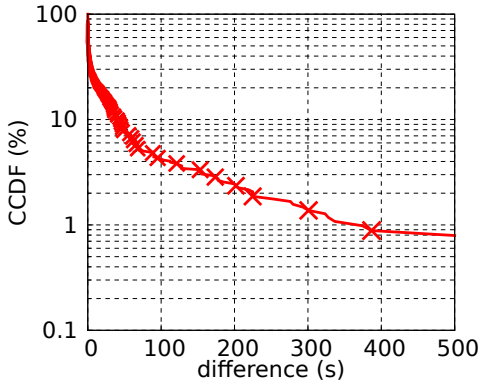

Fig. 3. The time difference between 2 co-occurred flows that highly likely from the same app.

In Fig. 3, we plot the complementary cumulative distribution function (CCDF) of the measured T window samples. The results show that 99% of samples are within 300 second intervals. Accordingly, FLOWR sets T to 300 seconds as the threshold that covers most relevant co-occurrence events. In our further evaluation, we discovered marginal benefits from further increasing T.

B. Threshold for Promoting Features to Signatures

FLOWR promotes a feature $\mathcal{F}$ to a new signature of app X if the feature's co-occurrence likelihood with the known signatures of app X ($P[X|\mathcal{F}]$) is sufficiently high. Obviously, if $P[X|\mathcal{F}] = 1$, we can safely say that the feature should be promoted to a signature. On the other hand, when $P[X|\mathcal{F}] \approx 1$, the promotion inevitably incurs some false positives in app identification.

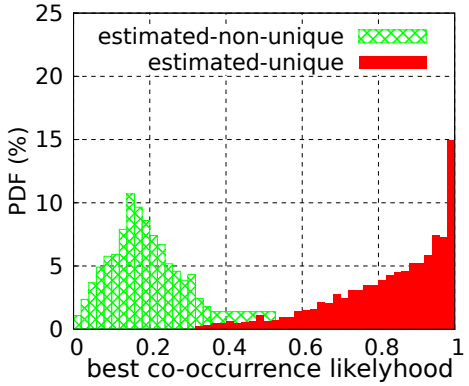

Fig. 4. The distribution of P_{max} values for unique and non-unique features.

Therefore, FLOWR sets a threshold p on the co-occurrence likelihood for feature promotion based on a tolerable rate of false positives. The threshold can be configured by a system administrator.

We first study the distribution of co-occurrence likelihoods for all features. Employing only $\{doubleclick.net : msid = ...\}$ signature seeds, we run FLOWR on the FlowSet. Our methodology is the following: Let $\mathcal{A}$ be a set of all apps related to doubleclick.net signatures in FlowSet. Then, for each feature $\mathcal{F}$, we compute $P_{max}[X|\mathcal{F}]$ as the highest co-occurrence likelihood of $\mathcal{F}$ being related to any potential app in $\mathcal{A}$, i.e.,

$$P_{max}[X|\mathcal{F}] = \max_{X \in \mathcal{A}} \left(P[X|\mathcal{F}]\right).$$

Having identified apps related to $P_{max}[X|\mathcal{F}]$, we then employ our ground truth knowledge from AppSet to verify that each $\mathcal{F}$ is indeed unique to its identified app X: For any feature $\mathcal{F}$ and its identified app X, we check whether $\mathcal{F}$ appears only in the traffic generated by app X in AppSet. If this is the case, we say $\mathcal{F}$ is *unique*; otherwise $\mathcal{F}$ is *non-unique*.

Fig. 4 shows the probability distribution function (PDF) of unique and non-unique features' best co-occurrence likelihoods. As shown in Fig. 4, the best co-occurrence likelihood values of unique features are much greater than those of the non-unique features. The majority of the unique features have best co-occurrence likelihood greater than 0.5, which sets a feasible range for a selection of the promotion threshold.

TABLE I
NOTATIONS USED IN DERIVING THE FALSE POSITIVE PROBABILITY.

U	feature $\mathcal{F}$ is unique
$\overline{U}$	feature $\mathcal{F}$ is not unique
M	$P_{max}[X|\mathcal{F}] > p$

Next, to identify a tradeoff between the chosen promotion threshold and the corresponding rates of false-positive app identifications, we establish a mapping between the two quantities. Using the notation listed in TABLE. I, we note that the probability of a false positive app identification corresponds to a feature $\mathcal{F}$ being non-unique and its best co-occurrence likelihood being larger than the promotion threshold p, i.e., $P_{max}[X|\mathcal{F}] > p$. Let $P(fp)$ be the false positive probability and we have:

$$P(fp) = P(\overline{U}|M).$$

To establish the mapping between the threshold p and $P(fp)$, we employ Bayes' theorem:

$$P(fp) = \frac{P(M|\overline{U}) \times P(\overline{U})}{P(M|\overline{U}) \times P(\overline{U}) + P(M|U) \times P(U)}.$$

Given the threshold value p, we can identify two types of related probabilities from Fig. 4: (1) $P(M|U)$, the probability that $\mathcal{F}$ is unique, but $P_{max}[X|\mathcal{F}] > p$; and (2) $P(M|\overline{U})$, the probability that $\mathcal{F}$ is non-unique, but $P_{max}[X|\mathcal{F}] > p$. The remaining probabilities in Bayes' formula can be measured: $P(U)$, the probability that $\mathcal{F}$ is unique, can be measured via the ground truth provided by AppSet. There, we "count" the percentage of features that appear only in traffic of single apps. Once $P(U)$ is measured, $P(\overline{U}) = 1 - P(U)$.

With the above derivation, we plot the mapping curve between the likelihood of false positive app identifications $P(fp)$ and the selected feature promotion threshold p in Fig. 5. This plot can be used for tuning the threshold p. According

to our results, guaranteeing false positives lower than 5%, means setting p higher than 0.8. To avoid any false positives, according to our extensive datasets, p should be set to 0.97.

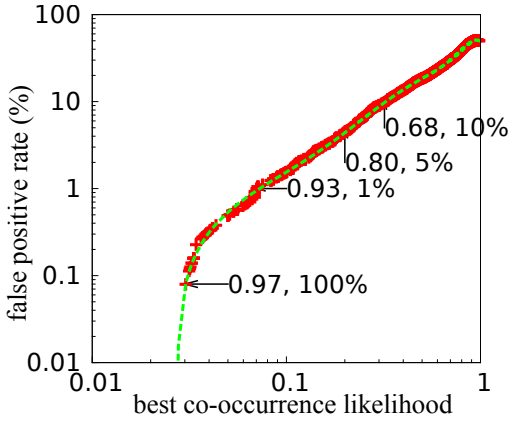

Fig. 5. The probability of false positives as a function of the threshold of promoting a feature to a signature.

VI. Evaluation

We evaluate FLOWR's identification performance via (i) FlowSet, as a source of real-world traffic, and (ii) AppSet which provides us with ground truth of app-identifying features. We focus on two crucial topics: First, we demonstrate that service names and key-value pairs are indeed suitable for app identification. Subsequently, we evaluate FLOWR's app- and flow coverage, provided by a significant amount of new signatures discovered by flow regression.

Before we start, it is important to understand the overarching idea of our experiments. For practical purposes, we characterize only the identification of apps that have some seeding signatures as sources of ground truth. Specifically, our evaluation focus on the apps related to one type of seeds, the doubleclick plus "msid=X" signatures. We rightfully assume that FLOWR would behave similarly in growing its app identification knowledge around any other seeds. Therefore, it is fair to extrapolate the results presented in this section to the general app identification.

A. Feature Quality

We start feature evaluation by testing the persistence of feature values. The persistence means that the values would not change over time or different app runs, which is the crucial for app identification. To this end, we exploit AppSet. Our experiment employs 8 independent app runs in which we ask the KVs to stay persistent (as indicated in TABLE. III); four runs are on the Gingerbread platform and the rest on Ice Cream Sandwich.

In Fig. 6, we show the distributions of persistent features, conditioned on observations after each app run. On average, in a single run there will be about 50 persistent features per app (see g^1 curve). Aggregating two Gingerbread runs (see $g^{(1-2)}$ curve), the number of features that are persistent in both runs reduces significantly to about 15 per app. With further runs, the number of persistent features remains relatively stable (from 8 to 15 features per app). This result provides us with confidence that mobile apps usually produce sufficient persistent features for app identification.

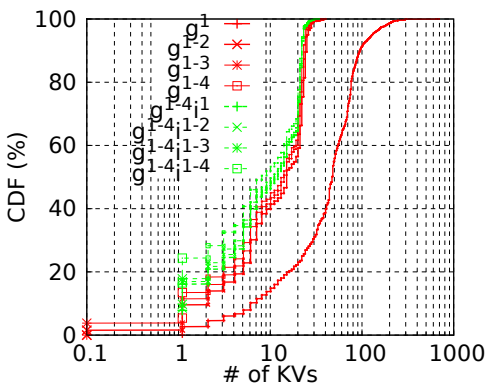

Fig. 6. Persistence of KVs. All tests but Gingerbread-1 (g^1) are consistent with one another.

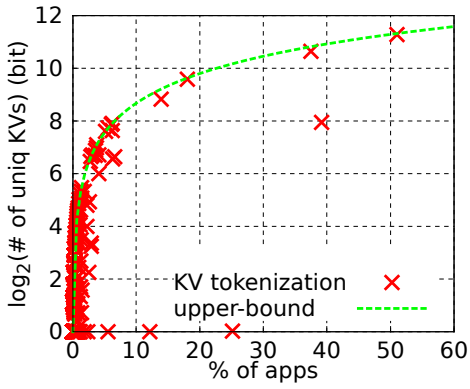

Fig. 7. Entropy of app features. A k-bit difference from the upper bound means classifying a flow within candidate groups of 2^k apps.

Next, we measure the entropy of persistent features in order to evaluate whether the features contain enough information to identify all apps. Given a service S that appears in AppSet, we first find all apps in AppSet that visit S as well as all persistent features related to this service. In principle, each app would correspond to a set of such features (denoted as $App_i\{KV\}$). Let's suppose that the number of distinct feature sets identified in AppSet is K, and the sets are $\{KV\}_1, \{KV\}_2, \ldots, \{KV\}_K$. The entropy can be evaluated as:

$$E = -\sum_{i=0}^{K} P[\{KV\}_i] \log_2 P[\{KV\}_i].$$

The corresponding probabilities can be measured as ratios of the number of apps related to the distinct feature set $\{KV\}_i$ over the total number of apps in AppSet, i.e., $P[\{KV\}_i] = count(App_i\{KV\} \supset \{KV\}_i)/count(apps)$.

We also identify the upper bound of entropy that would grant identification of all apps. Suppose that S is visited by N apps, the upper-bound of its feature entropy is then $\log_2(N)$. The bound is achieved only if each of the N apps has its own unique set of persistent features. A k-bit difference from the

upper-bound indicates that N apps have only $\frac{N}{2^k}$ unique and persistent feature sets. This in turn implies that such features can only identify $2^k < N$ "groups" of apps.

Fig. 7 shows the entropy of features as a function of the percentage of apps related to them. The results show that most features are related to very few apps. Moreover, the entropy is predominantly close to its upper-bound. Given the diversity of apps in our AppSet, we are confident that the observed result also applies to the real settings. We thus conclude that features composed of service names and HTTP URI parameters have very good app-identification properties.

B. Coverage Benefit Provided by FLOWR

The coverage of flows and apps that can be identified by FLOWR depends on the system's knowledge initial of app signatures. The key role of flow regression is to grow that knowledge base through a continual identification of new features suitable for app signatures.

In order to determine the growth of flow coverage during FLOWR's learning process, we iterate FlowSet in rounds through the flow regression. In each round, FLOWR updates the knowledge base by capturing the co-occurrences between yet uncharacterised features and the existing signatures, asking for a near-zero false positive identification rate (according to the settings presented in § V-B).

Flow coverage. We evaluate the flow coverage based on the real-world FlowSet traffic and verify the findings via the lab-generated AppSet. Given that features differ in their app-identification quality, FLOWR classifies the features in 5 categories listed in TABLE. II. Four categories correspond to the features that offer unique app identification: 100% true positive ("t100"), 99% true positive ("t99"), 95% true positive ("t95"), and 90% true positive ("t90"). To determine a category of a feature, we refer to Fig. 5 for a setting of the categorization threshold based on the feature's co-occurrence likelihood. We also introduce another feature category ("n5") that shrinks the identification indications to 2–5 apps with less than 1% false-positives.

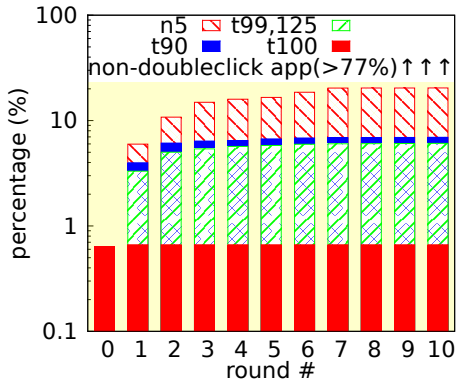

Fig. 8. The flow coverage seeded with doubleclick flows.

Fig. 8 shows the percentage of flows identified by each feature category in each flow regression round. Without the flow regression, FLOWR can only identify apps (and flows) suggested by our seeding doubleclick signatures. This accounts only for 0.7% of the flows in FlowSet. This is also the identification percentage for "t100" feature category which remains consistent across the flow regression iterations. On the other hand, flow regression easily increases the identification for about 6–7% of flows in FlowSet with no more than 5% false positives. Relaxing further the false positive requirement does not significantly improve FLOWR's flow coverage since "t90" only contributes 1% to the coverage as compared to "t95". On the other hand, if the unique app identification is not required, "n5" covers additional 14% of flows, reaching the total of 21% of identified flows in FlowSet.

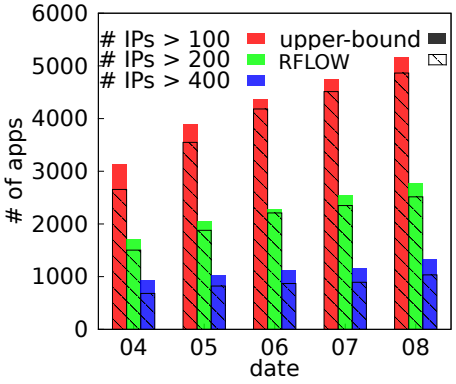

Fig. 9. The number of identified "doubleclick" apps. The R/G/B solid bars represent the upper-bound, and the patterned ones are for FLOWR's.

Next, we need to put these results in perspective and extrapolate them according to things that can be ideally discovered via the initial set of seeding signatures. In the presented evaluation, we employed only one type of seeds, the doubleclick plus "msid=X" signatures. Doing that, we significantly grew the flow coverage within the certain false-positive requirements. Let's next consider whether the achieved coverage is good. To this end, if the 21% of characterized flows is close to the total number of flows produced by the apps that employ doubleclick, it is safe to assume that flow regression came close to identifying everything it could. Consequently, if a larger set of seeding signatures were employed, the system would perform at least equally well.

To estimate the percentage of flows generated by "doubleclick" apps, we refer to the finding that most apps are used continually up to 30 minutes by the mobile users [14], [19]. In this interval, doubleclick would be contacted at least once (as we learned by analyzing AppSet). Therefore, we counted the number of flows in FlowSet whose closest doubleclick sessions were not more than 30 minutes away. We found that only 23% of flows matched, i.e., only this percentage of flows could be related to "doubleclick" apps. Consequently, scaling the flow regression's results to the "universe" of 23%, we can say that FLOWR uniquely identified 26–30% (i.e., 6–7% in 23%) of feasible flows and narrowed down 60–65% (i.e., 14–15% in 23%) flows to 2–5 candidate apps. Overall, the percentage of identified feasible flows is around 90%. Without FLOWR, only 3% (i.e., 0.7% in 23%) can be characterized.

TABLE II
THE CATEGORIES OF FLOW SIGNATURES. THE "T100", "T99", "T95", AND "N5" KVS ARE CONSIDERED AS IDENTIFIED.

Tag	Condition	Accuracy	Description
t100	$\exists KV, id(KV) \geq 0.97$	$FP = 0$	The best co-occurrence likelihood is at least 0.97. This leads to FLOWR's true positive in labeling the flow with 100%.
t99, t95,t90	$\exists KV, id(KV) \geq 0.93, 0.80, 0.68$	$FP \leq$ 1,5,10%	Similar to "t100", the best co-occurrence likelihood is at least 0.93, 0.80, or 0.68. This leads to true false positive of 99%, 95%, or 90%.
n5	$\exists KV_i, i = 1 \ldots 5, \sum_{i=1}^{5} id(KV_i) \geq 0.99$	$FP \leq 1\%$	FLOWR cannot classify the flow to a single app"t100", but can narrow down it to 2–5 apps with true positive >99%.

App coverage. Next, we try to evaluate the percentage of "doubleclick" apps that our system is able to discover. We first limit the scope to apps that are used by more than 100 users (i.e., more than 100 IP addresses) in one day. As shown in Fig. 9, on July 4th the total number of such apps was around 3000, while FLOWR identified around 2700 apps, i.e., 90%, without relying on seed signatures. Increasing the limit to apps used by at least 200 or 400 users, FLOWR identified 1500 from 1700 apps, and 700 from 900 apps respectively. Our results also show that the flow regression is capable of growing its app identification coverage over time.

VII. RELATED WORK

There have been numerous previous effort in investigating mobile apps from different aspects. Among the previous work, mobile or smartphone app profiling has yielded insights in the mobile community benefiting users, developers, OS vendors, network operators, and content partners. To our best knowledge, none of the prior studies have proposed any online approach to trace network traffic back to individual apps, which is essential in many practical scenarios. We broadly classify previous studies and position our study as follows.

App profiling: There have been studies focusing on profiling apps at the granularity of app types (e.g., email, social networking, music, and gaming) [4]–[6], [14], [20], [21]. or groups of apps (e.g., P2P vs. non-P2P) [9]. Compared with identifying individual apps, identifying app types or groups is easily accomplished with the port number, the hostname, and the application-level protocol. Identifying individual apps has different usage scenarios such as app-based policy enforcement, anomaly detection. Thus, FLOWR is a further step along the path of profiling apps. In app-based profiling, NetworkProfiler [15], ProfileDroid [10], PowerTutor [22], TaintDroid [23], and App Profiles [24] perform individual app profiling via instrumented devices or emulators, which limit themselves to small scales. Unlike these studies, FLOWR is an attempt to identify individual apps at a large scale targeting the majority of apps on mainstream app markets. Without on-device information, FLOWR aims at identifying the network traffic originated by apps. Thus, FLOWR is complementary to previous on-device app profiling approaches.

Traffic profiling: Diverse information inside network traffic has been explored by previous studies from many aspects, e.g., user browsing pattern [4], [20], content diversity [14], [25], privacy leakage [26]–[29]. Standing on the shoulder of previous studies, FLOWR leverages the rich information inside traffic that can reveal app identities to construct flow signatures. Rather than investigating the user-centric information, it emphasizes on app-centric information. FLOWR attempts to determine which part of the network traffic originated by apps can best identify apps.

Endhost profiling: Similar to the problem that FLOWR attempts to address, identifying users and devices is challenging due to the hardly manageable number of users, devices, or apps. In a small scale, the mobile users and smartphone devices have been characterized in a range of domains [13], [19], [30]. In a large but limited scale, MobiPerf [31] is developed to test network performance for users with their apps installed. FLOWR's approach can potentially shed light on studies on endhost profiling.

VIII. CONCLUSION

In this study, we developed FLOWR, a system automatically learns new signatures of mobile apps from a small set of initial seeding app identification signatures. FLOWR exploits the information in HTTP header fields as app features and infers new app signatures by observing the co-occurrence of app features.

Using a set of seeding signatures related to advertisement service doubleclick.net, we show that FLOWR can accurately associate 86–95% of flows to their generating apps. More specifically, with a false positive rate lower than 1%, FLOWR uniquely identifies the generating apps of 26–30% of the flows; for another 60–65% of the flows FLOWR narrows down the generating app of each flow to 5 or fewer candidates.

REFERENCES

[1] "Mobile Apps Put the Web in Their Rear-view Mirror," http://blog.flurry.com/bid/63907/Mobile-Apps-Put-the-Web-in-Their-Rear-view-Mirror.

[2] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis," in *Proc. ACM CCS*, 2007.

[3] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin, "A First Look at Traffic on Smartphones," in *Proc. ACM SIGCOMM IMC*, 2010.

[4] R. Keralapura, A. Nucci, Z. Zhang, and L. Gao, "Profiling Users in a 3G Network Using Hourglass Co-Clustering," in *Proc. ACM MOBICOM*, 2010.

[5] W. Cui, J. Kannan, and H. Wang, "Discoverer: Automatic Protocol Reverse Engineering from Network Traces," in *Proc. USENIX Security*, 2007.

[6] G. Wondracek, P. Comparetti, C. Kruegel, E. Kirda, and S. Anna, "Automatic Network Protocol Analysis," in *Proc. ISOC NDSS*, 2008.

[7] J. Kannan, J. Jung, V. Paxson, and C. Koksal, "Semi-Automated Discovery of Application Session Structure," in *Proc. ACM SIGCOMM IMC*, 2006.

[8] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song, "Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation," in *Proc. USENIX Security*, 2007.

[9] S. Sen, O. Spatscheck, and D. Wang, "Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures," in *Proc. ACM WWW*, 2004.

[10] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "ProfileDroid: Multi-Layer Profiling of Android Applications," in *Proc. ACM MOBICOM*, 2012.
[11] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Profiling Resource Usage for Mobile Applications: A Cross-Layer Approach," in *Proc. ACM MobiSys*, 2011.
[12] E. Chin, A. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-Application Communication in Android," in *Proc. ACM MobiSys*, 2011.
[13] "Netsense," http://netsense.nd.edu.
[14] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman, "Identifying Diverse Usage Behaviors of Smartphone Apps," in *Proc. ACM SIGCOMM IMC*, 2011.
[15] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, "NetworkProfiler: Towards Automatic Fingerprinting of Android Apps," in *Proc. IEEE INFOCOM*, 2013.
[16] "App Engagement: The Matrix Reloaded," http://blog.flurry.com/bid/90743/App-Engagement-The-Matrix-Reloaded.
[17] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," RFC 3986, Internet Engineering Task Force, 2005. [Online]. Available: http://www.ietf.org/rfc/rfc3986.txt
[18] A. Tongaonkar, S. Dai, A. Nucci, and D. Song, "Understanding mobile app usage patterns using in-app advertisements," in *Passive and Active Measurement*. Springer, 2013.
[19] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in Smartphone Usage," in *Proc. ACM MobiSys*, 2010.
[20] I. Trestian, S. Ranjan, A. Kuzmanovic, and A. Nucci, "Measuring Serendipity: Connecting People, Locations and Interests in a Mobile 3G Network," in *Proc. ACM SIGCOMM IMC*, 2009.
[21] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, "BLINC: Multilevel Traffic Classification in the Dark," in *Proc. ACM SIGCOMM*, 2005.
[22] "PowerTutor," http://powertutor.org.
[23] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proc. USENIX OSDI*, 2010.
[24] "APP Profiles," http://appprofiles.eecs.umich.edu.
[25] G. Maier, F. Schneider, and A. Feldmann, "A First Look at Mobile Hand-Held Device Traffic," in *Proc. International Conference on Passive and Active Network Measurement (PAM)*, 2010.
[26] B. Krishnamurthy, K. Naryshkin, and C. Wills, "Privacy Leakage vs. Protection Measures: the Growing Disconnect," in *Proc. IEEE Workshop on Web 2.0 Security and Privacy*, 2011.
[27] C. Mulliner, "Privacy Leaks in Mobile Phone Internet Access," in *Proc. International Conference on Intelligence in Next Generation Networks (ICIN)*, 2010.
[28] B. Krishnamurthy and C. E. Wills, "On the Leakage of Personally Identifiable Information via Online Social Networks," in *Proc. ACM Workshop on Online Social Networks (WOSN)*, 2009.
[29] C. Riederer, V. Erramilli, A. Chaintreau, B. Krishnamurthy, and P. Rodriguez, "For Sale: Your Data. By: You," in *Proc. Workshop on Hot Topics in Networks (HotNets)*, 2011.
[30] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "LiveLab: Measuring Wireless Networks and Smartphone Users in the Field," in *Proc. Workshop on Hot Topics in Measurement & Modeling of Computer Systems (HotMetrics)*, 2011.
[31] "MobiPerf," http://mobiperf.com.
[32] "APK Downloader," http://codekiem.com/2012/02/24/apk-downloader.

APPENDIX A
DATASETS

We utilize network traffic from two data sources in our study to evaluate FLOWR, referred to as FlowSet and AppSet, respectively. TABLE. III provides an overview of the datasets.

TABLE III
DATASETS OVERVIEW.

Dataset	Source	Scalability	Availability	Duration
FlowSet	network	>22K apps	real-time	07/04/20xx∼ 07/09/20xx
AppSet	emulator	5K apps	4+4 runs	N/A
		10K apps	4+4 runs	N/A

FlowSet is an anonymized packet trace provided by a nationwide cellular network provider. It contains six days' of traffic traversing a gateway of the network provider in July 20xx[2], which has more than 10 billion traffic flows. This packet trace is dominated by traffic from Android devices and has a fraction of traffic from Windows Phone, Blackberry, and iOS devices as well. We observe 22K distinct apps in FlowSet just by counting the KVs of `msid=` in the HTTP requests sent to doubleclick.net. However, this dataset does not have the ground truth of the app identity for every network flow.

AppSet is produced by executing Android apps in emulators provided by Android SDK. Before installing an app into the emulator, we make sure that the emulator instance is in a clean state, in which no other apps are installed or running. We capture traffic from the emulator when running the app. In this way, we can have the ground truth of the originating app for every flow in this dataset. We use the Android monkey tool to automatically drive the execution of apps. Although the Android monkey cannot comprehensively emulate user interaction with apps, in our study as long as the emulated traffic covers some traffic that can be potentially observed in FlowSet, it is sufficient for our evaluation purpose.

Determining target apps: Given that it is prohibitive to crawl all apps on Google Play, we first download the most popular apps. At the time when we conducted our experiments, only the top 5K apps popular were directly accessible through a simple web query on Google Play. Those top 5K apps on Google Play can reasonably well represent the dominant apps in FlowSet. Previous study [14] shows that the total contribution of the top 5K apps consistently cover 98% of total traffic volume, regardless of how apps are ordered. To improve the app coverage, we also discover and crawl another 10K apps by automating random searches on Google Play website, in addition to the top 5K apps.

Crawling installation packages: In order to download the installation packages from Google Play website, we modify a Chrome browser extension named "APK Downloader" [32] to download the target installation packages in parallel.

Emulating apps: We run the crawled apps on two different Android OS emulators, i.e., Gingerbread and Ice Cream Sandwich. We use Android monkey too to interact with apps by injecting random clicking and typing events. On average, we can emulate 10K apps once every 24 hours via 10 Linux virtual machines on an off-the-shelf server, where one run of an app uses 600 random events.

[2]We obfuscate the exact year for privacy reasons.