SAMPLES: Self Adaptive Mining of Persistent LExical Snippets for Classifying Mobile Application Traffic

Hongyi Yao[†], Gyan Ranjan[‡], Alok Tongaonkar[‡], Yong Liao[‡] and Z. Morley Mao[†]

[†]Department of EECS,
University of Michigan, Ann Arbor, MI, USA.
[†]{hyyao, zmao}@umich.edu

[‡]Center for Advanced Data Analytics,
Symantec Corp., Mountain View, CA, USA.
[‡]{firstname_lastname}@symantec.com

ABSTRACT

We present **SAMPLES**: **S**elf **A**daptive **M**ining of **P**ersistent **LE**xical **S**nippets; a systematic framework for classifying network traffic generated by mobile applications. SAMPLES constructs *conjunctive* rules, in an automated fashion, through a supervised methodology over a set of labeled flows (the *training set*). Each conjunctive rule corresponds to the lexical context, associated with an application identifier found in a *snippet* of the HTTP header, and is defined by: (a) the identifier type, (b) the HTTP header-field it occurs in, and (c) the prefix/suffix surrounding its occurrence. Subsequently, these conjunctive rules undergo an *aggregate-and-validate* step for improving accuracy and determining a priority order. The refined rule-set is then loaded into an application-identification engine where it operates at a per flow granularity, in an *extract-and-lookup* paradigm, to identify the application responsible for a given flow. Thus, SAMPLES can facilitate important network measurement and management tasks — e.g. behavioral profiling [29], application-level firewalls [21, 22] etc. — which require a more detailed view of the underlying traffic than that afforded by traditional protocol/port based methods.

We evaluate SAMPLES on a *test set* comprising 15 million flows (approx.) generated by over 700 K applications from the Android, iOS and Nokia market-places. SAMPLES successfully identifies over 90% of these applications with 99% accuracy on an average. This, in spite of the fact that fewer than 2% of the applications are required during the training phase, for each of the three market places. This is a testament to the universality and the scalability of our approach. We, therefore, expect SAMPLES to work with reasonable coverage and accuracy for other mobile platforms — e.g. BlackBerry and Windows Mobile — as well.

Categories and Subject Descriptors

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

Keywords

Mobile App Identification; Automated Rule Generation.

MobiCom'15, September 07 - 11, 2015, Paris, France
Copyright 2015 ACM 978-1-4503-3619-2/15/09 ...$15.00
DOI: http://dx.doi.org/10.1145/2789168.2790097.

1. INTRODUCTION

Network operators need clear visibility into the applications running within — and responsible for the traffic in — their networks to perform key network management tasks. These may include network planning and bandwidth provisioning, billing, traffic engineering, deployment of application level firewalls and access control systems. For instance, a network operator might want to prioritize traffic from online meeting applications, block traffic from a given gaming app, or catalog application behavior profiles [29] and usage statistics for users within a given enterprise [21,22]. Such nuanced policy formulation and enforcement requires a granular view of the network traffic that is clearly beyond the scope of traditional protocol/port/volumetric based classification systems [8, 10, 18]; and their primitive DPI (deep-packet-inspection) counterparts [14].

The problem has been further compounded in recent years with the adoption of hand-held devices, such as mobile phones and tablets, as preferred end-hosts for accessing the Internet. Users of such devices often download and install applications, commonly called *mobile apps*, that provide a wide range of functions. A significant number of these applications use the HTTP/HTTPS protocols for communicating with not only their respective content hosting servers, but also a plethora of third party services (e.g. advertisement and analytics). This, coupled with the so called *Bring-Your-Own-Device* (BYOD) to work phenomenon, poses unprecedented challenges for network managers; particularly in an enterprise setting. With the advent of wearable devices (e.g. Apple watch, Google-glass etc.), this trend is expected to continue for the foreseeable future. Needless to say, all this necessitates the design and development of new approaches to tackle a rapidly changing network landscape. Consequently, several recent studies have tried to address the problem of identifying mobile applications in network traffic [11, 19, 23, 27, 29], at a per flow granularity. While interesting in their own right, these solutions are either platform specific [19, 23, 27, 29], and thus limited, or too computationally expensive to attain desired levels of scalability [11]. We discuss each of these in detail later.

In this work, we present **SAMPLES**: **S**elf **A**daptive **M**ining of **P**ersistent **LE**xical **S**nippets; a systematic framework for classifying network traffic generated by mobile applications at a per-flow granularity. In order to do so, SAMPLES relies on the content of the HTTP headers generated by mobile applications. An HTTP header is, in effect, an ensemble of structured fields (cf. Table 1); one or more of which may potentially contain an identifier string — such as the market-place application id or the name of the application — that identifies the app responsible for the flow. Indeed, it is known that mobile applications use such identifiers while communicating with their content host servers as well as third party services (e.g. advertisement and analytics); for a variety of practi-

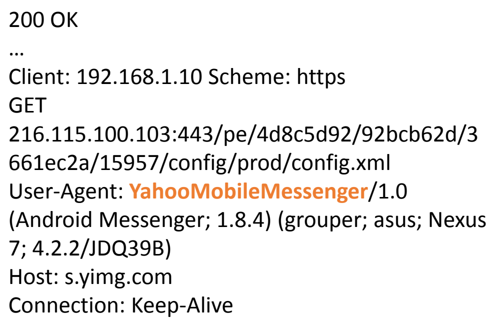
Field Name	Tag used in this study	Description
`Domain host-name`	HST	The host-name in a URL [**csi.gstatic.com**`/csi?v=3&s=gmob`]
`Method`	MET	HTTP method [GET, PUT, POST, REST].
`User-Agent`	AGN	See [4].
`Path`	URI	Part of URL b/w first / and ? [`csi.gstatic.com/`**csi**`?v=3&s=gmob`).
`Query Parameters`	PAR	Part of URL after ? [`csi.gstatic.com/csi?`**v=3&s=gmob**).
`Referrer`	REF	A referrer URL, see [4].
`Cookies`	COO, SCO	Cookie and Set-Cookie, see [4].
`Others`	ALP, APL_WO, APL_CX, XRW	Non-standard fields.

Table 1: Commonly seen fields in the HTTP flow header of mobile app network traces. For details see [4].

cal reasons [23, 24]. SAMPLES exploits the occurrence of such identifiers, and the lexical context in which they occur, to construct generalized *conjunctive rules* through a supervised methodology as follows: SAMPLES samples a subset of randomly selected applications for each mobile platform (Android, iOS and Nokia). Using a lightweight *crawl-download-execute* infrastructure, it builds a repository of all possible identifier strings — such as *application name*, *application id* etc — by parsing market webpages as well as lexically meaningful contents of the application's executable archive. Next, in an automated execution environment, SAMPLES executes the applications, one at a time, to produce HTTP flows. The flows that contain in their header one or more identifiers of the parent app, are then grouped them into a flowset labeled with the unique application identifier. These flowsets constitute both the *training* as well as the *test* sets for the supervised methodology. Next, for each flow in the training set, SAMPLES characterizes the *lexical context* associated with an identifier string, found in a snippet of the header, in terms of three lexical conjuncts: (a) the identifier type, (b) the HTTP header-field (or payload) it occurs in, and (c) the prefix/suffix that surrounds such occurrences. Clearly, the same *lexical context* may persist across flows produced by multiple applications. This is due, in part, to functional specifications of host services as well as structural idiosyncrasies of third party libraries [23, 30]. It is precisely such persistent lexical contexts that constitute a conjunctive rule in the SAMPLES framework. Once constructed, the rule-sets are loaded into an app identification engine where they operate at a per-flow granularity.

We evaluate SAMPLES using 15 million flows generated by 700 K applications from the Android, iOS and Nokia market-places. SAMPLES successfully identifies over 90% of these applications with 99% accuracy on an average. This, in spite of the fact that the training phase requires fewer than 2% of the applications in the market-place, per platform; a testament to the scalability and generality of our design. Moreover, SAMPLES can achieve processing *goodputs* of 217 Mbps, 40 Mbps and 150 Mbps respectively on the three platforms. Although measured in a controlled laboratory environment, the results bode well from a real-world deployment standpoint. Given that SAMPLES relies on seemingly universal network behavior of mobile apps, we believe that it should extend to other platforms such as Windows Mobile and BlackBerry; as is.

The rest of this paper is organized into the following sections: in §2, we introduce preliminary notations and assumptions, along with the datasets used in the course of this study. Next, in §3, we describe the various types of application identifiers and the lexical context in which they are found in the *snippets* of HTTP headers, followed by a formal definition of what constitutes a *conjunctive rule*. We then outline the design goals for SAMPLES in §4 and the system architecture in §5. Experimental evaluation of SAMPLES from coverage, accuracy and performance standpoints is presented in §6. Finally, in §7, we review relevant studies in literature and conclude the paper with a discussion of potential future work in §8.



```
200 OK
…
Client: 192.168.1.10 Scheme: https
GET
216.115.100.103:443/pe/4d8c5d92/92bcb62d/3
661ec2a/15957/config/prod/config.xml
User-Agent: YahooMobileMessenger/1.0
(Android Messenger; 1.8.4) (grouper; asus; Nexus
7; 4.2.2/JDQ39B)
Host: s.yimg.com
Connection: Keep-Alive
```

Figure 1: Decrypted HTTPS flow header from Yahoo Messenger app on Android platform.

2. PRELIMINARIES AND DATASETS

We now introduce the notations and assumptions of our framework (§2.1), followed by a brief discussion of the datasets used in the course of this study (§2.2).

2.1 Notations and Assumptions

Network flow: A flow $\mathcal{F}$ is defined as a single application layer request-response pair. In particular, all flows considered in this work are HTTP flows. When referring to an HTTP flow, we will use the notational schema for its constituent fields as defined in Table 1.

Identifiable flow/app: We define an HTTP flow as **identifiable** if it contains at least one **identifier** for its parent app in the header (cf. §3.1). Similarly, a mobile application is deemed **identifiable** if it generates at least one identifiable flow during its execution. In our dataset, there are roughly 76.8%, 67.4% and 63.2% identifiable apps for the Android, iOS and Nokia platforms respectively. The rest of the apps either do not exhibit any meaningful network behavior; or simply do not execute in our automated execution environment. However, as shown in §6, this does not hamper the coverage or accuracy of our system in any significant way.

HTTPS and encrypted traffic: HTTPS, in the very general case, is a limitation of our work. We do not deal with the issue of encrypted traffic *in the wild* which, although an interesting and complimentary area of work, is orthogonal to our system [15]. However, in an enterprise deployment scenario, the SAMPLES framework can utilize a *man-in-the-middle* proxy, as described in [21, 22], to gain visibility in the underlying HTTPS traffic. Figure 1 shows a snippet from the header of an HTTPS session between the Yahoo Messenger app installed on an Android device (Nexus 7 phone). The lexical construct of this header is no different from those seen in the case of HTTP (cf. Figure 2). Therefore, in an enterprise set up, the SAMPLES framework extends seamlessly — *as is* — to the HTTPS traffic generated by mobile apps.

```
AGN: AIM/5.1.3.11 CFNetwork/548.1.4 Darwin/11.0.0
HST: o.aolcdn.com
URI: /os/mservice/…/com_aol_aim/v3/manifest.json
MET: GET
```

```
AGN: AIM/5.1.3.11 CFNetwork/548.1.4 Darwin/11.0.0
HST: b.aol.com
URI: /ping
MET: GET
PAR: …&ap_v=1&dL_ch=com.aol.aim&h=com.aol.aim.ios.ap
plication&…
```

(a) AOL Messenger (iOS, app id = 281704574)

```
AGN: FOX 2 News 2.60.10776 rv:10776 (iPod touch; iPhone
OS 5.1.1; en_US)
HST: m.nowcache.com
URI: wjbk/imgdir/logo/wjbk_52x38.jpg
MET: GET
APL_CX: com.cbcnm.iapp.now.wjbk
```

```
AGN: Mozilla/5.0 (iPod touch; U; CPU OS 5.1.1 like Mac OS X;
en) AppleWebKit/531.21.10 (KHTML, like Gecko)
HST: wjbk.m0bl.net
URI: /json/iapp/main
MET: GET
APL_CX: com.cbcnm.iapp.now.wjbk
```

(b) FOX 2 News (iOS, app id = 364726223)

```
AGN: Nokia502/14.0.1/java_runtime_version=Nokia_
Asha_1_2 Profile/MIDP-2.1 Configuration/CLDC-1.1
HST: a.vserve.mobi
URI: /delivery/adapi.php
MET: POST
TPL: …&app=1&mn=1001%20Essential%20Words&showat=
start&…
```

(c) 1001 Essential Words (Nokia, app id = 389585)

```
AGN: Mozilla/5.0 (Linux; Android 4.3; en-US; sdk
Build/JB_MR2) AppleWebKit/534.30 (KHTML, like Gecko)
Version/4.0 Mobile Safari/534.30
HST: images1.statsheet.com
URI: /javascripts/flotr2/flotr2.min.js
MET: GET
XRW: statsheet.statblogs.FAUReview
```

(d) FAU Football & Basketball
(Android, app id = statsheet.statblogs.FAUReview)

Figure 2: Flow headers: AOL messenger (iOS), Fox 2 News (iOS), 1001 Essential Words (Nokia) and FAU Football and Basketball (Android).

Mobile app executable: We assume that the executable archive for an application (paid or free) can be obtained from the marketplace. This is essential for constructing the identifier sets as well as generating labeled flowsets during the training phase (cf. §5).

2.2 Datasets

App executables, identifiers and labeled flowsets: A set of app executables downloaded from the Google Play (Android), Apple iTunes (iOS) and OVI store (Nokia Symbian) markets. In all, the dataset comprises of over 700 K apps. For each mobile app, we collect the various classes of identifiers (static analysis) as well as network traffic produced by the app (dynamic analysis). The traffic is organized into flowsets at a per app granularity (cf. §5 for details). Each such flowset is labelled and thus constitutes the *ground truth* for our experiments (cf. §6).

Manually generated rulesets: As a baseline for the comprehensiveness of SAMPLES, we leverage three sets of conjunctive rules — respectively for Android, iOS and Nokia platforms — constructed by a team of human experts. These rules were created by manual inspection of HTTP traces. The conjunctive ruleset produced by SAMPLES should be, at the very least, an encompassing superset of these manually generated rule sets.

3. IDENTIFIERS, LEXICAL CONTEXT & CONJUNCTIVE RULES

We introduce the various *identifier types* associated with mobile applications (§3.1) and the *lexical context* in which they are found in the network traffic (§3.2). We then define an *application identification rule* as a conjunction of lexical contexts (§3.3); which forms the intuitive basis for this work.

3.1 Application Identifiers

Simply put, an application identifier is a string label that is uniquely associated with a mobile application. For example, consider the popular game *Angry Birds*. The mobile version of this game has identifiers *com.rovio.angrybirds*, *409807569* and *23158* respectively in the Google Play, Apple iTunes and Nokia OVI store marketplaces[1]. When downloaded from Google Play and Apple iTunes markets, the *apk* and *ipa* files are named *com.rovio.angrybirds.apk* and *409807569.ipa* respectively. In the Nokia OVI Store, the executable archive, a *jad* file, bears the application name instead i.e. *Angry Birds.jad*. Note that the application name — *Angry Birds* — is itself an identifier string, albeit not necessarily unique. Henceforth, we use the terms **app id** and **app name** to respectively mean the market place application identifier and application name for a given app. Both **app id** and **app name** may occur in the network traffic generated by mobile applications (cf. Figure 2).

The app id and app name, however, are not the only identifiers associated with a mobile app. The app executable archive (*apk*, *ipa* and *jad*) contains, amongst other things, a metadata file — *AndroidManifest.xml*, *info.plist* and *Manifest.xml* respectively for the Android, iOS and Nokia platforms — that provides behavioral specifications and access permissions required by the app for proper functioning. This metadata file may also contain other potential identifiers. For example, the *info.plist* file for an iOS app, contains a set of XML-fields (keys); such as **cfbundleidentifier**, **cfbundledisplayname**, **cfbundleexecutable** and **cfbundlename**. For the *Angry Birds* app, they respectively take on the values: **com.rovio.angrybirdsfree**, **Angry Birds**, **AngryBirdsClassicLight** and **Angry Birds Free**. The CFBundle identifiers are specified by application developers for packaging during the app development process. More importantly, some CFBundle identifiers (values) occur in the **User-Agent** (AGN) of the HTTP header. These have been exploited previously for classification tasks [29].

There are several functional reasons why identifiers exist in the HTTP headers. First and foremost, this is a simple, yet effective, mechanism for authentication and access control. Most mobile applications rely on content hosting servers that form the backend for an app enabled service. The hosting servers need to differentiate between different versions of the same app — paid vs. free — while catering content. Similarly, the same application developer may have a suite of applications accessing a common cloud-based content service. Identifiers help keep track of resource usage, userbase and demographics, for each application. This is essential for business objectives. Last but not least, mobile applications often use third party services such as analytics (e.g. Google-Analytics), APIs (e.g. Adobe-Air) and ad-vendors (e.g. Double-Click, iAds, Migital). Such third party services in turn assign additional service specific identifiers to each app for their own record keeping and accounting. Indeed, it is the presence of such service specific identifiers, in the form of key-value pairs in the *manifest* files of Android applications, that is used in [23] for application identification. Our system assumes that most apps use their true identifiers while communicating with their host servers or third party services. This is a reasonable assumption given the aforementioned usefulness of such behavior. If, however, for some reason an app is deliberately adversarial, i.e. obfuscates its identity or puts misleading identifiers in its headers, our system will surely not be able to identify it correctly.

To summarize, a mobile application can have a variety of identifiers, unique or otherwise; and some of these occur in the network traffic generated by the application a subset of which has been alluded to in previous studies [23, 29].

3.2 Identifiers in a Lexical Context

We define the lexical context associated with an identifier string occurring in a given network flow in terms of three contextual clauses — *which*, *where* and *what* — as described below:

1. *Which* identifier type is it?
2. *Where* in the header/payload does it occur?
3. *What* prefix/suffix surrounds its occurrence?

[1] Market-place application identifiers for Android apps usually have a Java package-name format. For iOS, these identifiers are in the form of a 9-digit string, and Nokia OVI store uses a numerical string id.

App Id	Identifier Type	Field Name: Snippet
281704574	cfbundlename/displayname	**AGN:** AIM/5.1.3.11 CFNetwork/548.1.4 Darwin/11.0.0
281935788	cfbundlename/executable	**AGN:** Essentials/1309241730 CFNetwork/609.1.4 Darwin/13.0.0
282935706	cfbundlename/displayname/executable	**AGN:** Bible/4.0.1 CFNetwork/548.0.4 Darwin/11.0.0
		AGN: ˆ(.+)/[0-9].+ CFNetwork/[0-9].+ Darwin/[0-9].+
364726223	cfbundledisplayname	**AGN:** FOX 2 News 2.60.10776 rv:10776 (iPod touch; iPhone OS 5.1.1...)
372198459	cfbundlename/displayname/executable	**AGN:** Rugby FM 4.3 rv:403 (iPad; iPhone OS 6.0.1; en_US)
371932548	cfbundlename/displayname/executable	**AGN:** Bakodo 3.4 rv:403 (iPad; iPhone OS 6.0.1; en_US)
		AGN: ˆ(.+)/[0-9].+ rv:*.+ (iP.+; iPhone OS.+; en_US)
364726223	cfbundleidentifier	**APL_CX:** com.cbcnm.iapp.now.wjbk
377749075	cfbundleidentifier	**APL_CX:** com.cbcnm.iapp.now.wfld
386978415	cfbundleidentifier	**APL_CX:** com.cbcnm.iapp.now.kjct
		APL_CX: ˆ(.*?)$
281704574	cfbundleidentifier	**PAR:** ... **&dL_ch=**com.aol.aim& ...
307840047	cfbundleidentifier	**PAR:** ... **&dL_ch=**com.aol.moviefone& ...
646100661	cfbundleidentifier	**PAR:** ... **&dL_ch=**com.aol.mobile.aolclient& ...
		PAR: **\b dL_ch=**(.*?) \b
380335515	app id	**APL_WO:** ... **&appAdamId=**380335515& ...
382287127	app id	**APL_WO:** ... **&appAdamId=**382287127& ...
382303945	app id	**APL_WO:** ... **&appAdamId=**382303945& ...
		APL_WO: **\b appAdamId=**(\d{9}) \b
286851614	app id	**PAR:** ... **&msid=**286851614& ...
288553125	app id	**PAR:** ... **&msid=**288553125& ...
291826753	app id	**PAR:** ... **&msid=**291826753& ...
		PAR: **\b msid=**(\d{9}) \b

Table 2: Lexical context for different identifier types found in network traffic generated by iOS applications.

For example, consider a pair of flow headers, generated by the *AOL messenger* app (*app id = 281704574*) on the iOS platform (cf. Figure 2(a)). The identifier **AIM**, occurs in both flow headers in the following lexical context: {*Which*: **cfbundlename/ cfbundledisplayname/ cfbundleexecutable**, *Where*: **AGN**, *What*: prefix = ˆ (i.e. beginning of the field), suffix = **/5.1.3.11 CFNetwork/548.1.4 Darwin/11.0.0**}. On the other hand, the identifier **com.aol.aim** occurs in two different lexical contexts within the same flow header. First, {*Which*: **cfbundleidentifier**, *Where*: **PAR**, *What*: prefix = **\b dL_ch=**, suffix = **\b**}. And then, {*Which*: **cfbundleidentifier**, *Where*: **PAR**, *What*: prefix = **\b h=**, suffix = **.ios.application\b**} [2]. Other variations can be seen in the remaining flow headers in Figure 2, where identifiers of different types are present in the AGN, HST and URI fields as both *leading* and *non-leading sub-strings* (2(b)), in the TPL field as a value to a key (2(c)) and in the APL_CX and XRW fields as *spanning field values* (2(b), (d)) respectively. In spite of such variations, each one of these occurrences can be represented in the form of the three contextual clauses: *which*, *where* and *what*. Evidently, multiple applications may share one or more generalized lexical contexts for the same identifier type (cf. Table 2). Next, we show how such lexical contexts can be generalized to formulate *conjunctive rules* that help identify mobile applications in network traffic.

3.3 From Context to Rules

In its simplest form, an app identification rule is a *conjunction* of a generalized lexical context associated with an identifier type. For instance,

App-Ident-Rule 1. ***Extract FROM AGN, Pattern:^(.+)/[0-9].+ CFNetwork/[0-9].+ Darwin/[0-9].+, AND Lookup IN {cfbundleexecutable / cfbundlename / cfbundledisplayname}.***

can be interpreted as the following: if the **User-Agent**(AGN) of a flow matches the pattern in Rule 1, extract the leading sub-string until the first '/'; and then lookup if the sub-string corresponds to a valid {**cfbundleexecutable / cfbundlename / cfbundledisplayname**} for any app (or apps). If found, report the corresponding app (or apps) as the output. In practice, the *lookup* step involves a dictionary (hash) reference. Each identifier type has its own dictionary — built *a priori* through market-place crawling and static analysis of app executable archives (cf. §5 for details) — with identifier strings as keys and the **app id**'s as values. For example, Rule 1 will extract the string **AIM** from flow headers 1 and 2; which then maps to {**cfbundledisplayname**: **AIM** → *281704574*} as the output. Observe from Table 2, that similar *extract-and-lookup* conjunctions can be derived for various identifier types occurring in multiple lexical contexts. Moreover, the same lexical context is found across the headers corresponding to multiple applications.

[2] The regular expression **'\b'** denotes a *word* boundary.

However, before we describe how SAMPLES constructs such conjunctions using a semi-supervised learning approach, we need to add a final contextual dimension to our conjunctive rules, that of a *condition*.

APP-IDENT-RULE 2. ***Extract** FROM **PAR\REF**,*
***Pattern:\b msid=(\d{9})\b**, AND **Lookup** IN {app id},*
*IF **HST: googleads.g.doubleclick.net**.*

The *IF* clause in Rule 2 specifies a condition under which the extraction should be performed. Intuitively, flows directed to and from third-party-services — ad networks, API's, analytics — are likely to have their own specific lexical structure. Therefore, the domain host name provides a good preliminary filter for rules specific to such services. In general, the *IF* clause can be combination of multiple conditions involving more than one field/pattern in the HTTP flow header. Such conditions not only help reduce potential false positives, but also contribute to improved system performance. SAMPLES framework generates such conditions by default, without any additional methodological overheads (cf. §5).

4. IDENTIFICATION & DESIGN GOALS

The set of app identification rules are consumed by an **app identification engine**. For each incoming flow, the app identification engine applies a subset of rules to the flow based on a number of pre-filtering conditions (e.g. host-name / field based pre-filtering) . If a flow matches one of the rules in the rule set, we classify it as **identified**, and report the **app id**; else it is reported as **unidentifed**.

Identification results: The identification result for each flow in the test set can be one of the six possible cases illustrated in Figure 3. We explain each of these below:

1. True positive (unique or fuzzy match): If the extracted app identifier can map either uniquely to its originating app or fuzzily to a few apps including the correct app, we say it is a **unique match** (Figure 3(a) & (b)).

2. False-positive: if the extracted identifier maps to an app id different from the flow's originating app, we say it is an **false positive** (Figure 3(c)).

3. True-negative: if the flow does not contain any known identifier, it is definitely not possible for SAMPLES to identify this flow (Figure 3(d)).

4. False negative (lacking identifier info): if the flows actually contains its originating app's identifier, but we either do not have the correct rule to extract it or cannot find it in the id hash table, it becomes a false-negative (Figure 3(e)).

The reason why we treat both unique and fuzzy match as true-positive is that fuzzy match also helps to reveal the "possible" mobile apps. In fact, these potential apps usually have some common properties. As shown in Figure 3(b), in iOS the cfbundlename *AIM* is used by app *444081514*, *494258199* and *281704574*. All these three apps are AIM Messenger apps developed by different companies. To avoid having fuzzy matching via some generic strings shared by many apps, we can set a threshold to the number of related apps on each identifier and prune the identifier out when the threshold is exceeded. In this case, we can ensure that fuzzy match can also give us useful information about apps in the network.

Design goals: Our goal is to build an efficient rule-based identification system. Specially, we would like to have

1. High coverage: we should be able to find most apps in the network whose flows actually contain app identifier. In other words, the false-negative rate should be small.

2. High accuracy: the identification results should be reliable, i.e. containing as few false-positives as possible.

3. High processing throughput: the system should be able to process flows at high speed.

It is challenging to create a rule set that can achieve all three aforementioned goals. Like any engineering activity, there are tradeoffs involved. For example, if we add more "app-specific-rules" (which identify flows from a small set of very specific apps) into the rule set, the coverage can be increased. However, a larger rule set results in longer processing times leading to lower system throughput. In particular, there are bound to be unidentifiable flows that must go through the entire rule set before being regarded as "not identified'. Besides that, if we generate rules with less matching conditions and more general regular expressions, the identification throughput and app coverage may increase. However, the identification results will contain more false-positive; thus being less reliable. Therefore, we should have an effective rule generator that automatically builds rules from training over labeled network flows, via a supervised learning methodology; thereby ensuring good quality and performance.

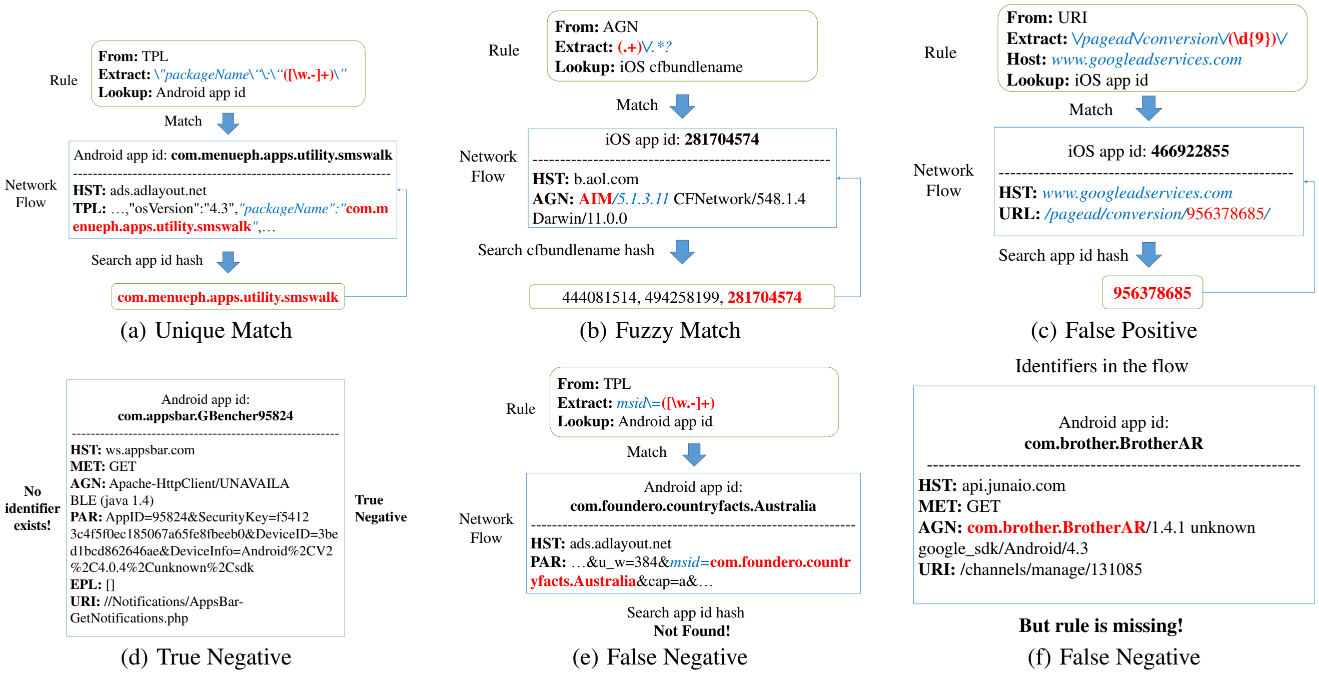
5. SAMPLES: SYSTEM ARCHITECTURE & OPERATIONAL LIFE CYCLE

In this section, we present the system architecture and the operational life cycle of SAMPLES (cf. Figure 4). SAMPLES comprises of two parts, an offline training system for rule generation and an online app identification engine for traffic classification. Both components can be implemented on commodity hardware. Each subsection describes an individual component along with the underlying intuitions, operational algorithms and the various challenges associated with the sub-tasks that they are designed to handle. Where applicable, we also compare and contrast the three mobile application platforms against each other to bring out the subtle differences between them; particularly those pertinent to the SAMPLES framework.

5.1 The Crawl-Download Infrastructure

SAMPLES comes equipped with a crawl-download infrastructure that periodically samples the three market places for newly released mobile apps: Google Play (Android), Apple iTunes (iOS) and Nokia OVI Store (Symbian)[3]. For the Android and iOS market places, our implementation is essentially the same as that presented in [26]. We deploy a globally distributed infrastructure and crowd-source authentication credentials, to avoid being blocked by the host markets. The task for Nokia platform is more easily managed. Unlike the other two platforms, the Nokia OVI store assigns a numerical **app id** string to the apps; e.g. *app id = 23158* for the *app name = Angry Birds*. The URL is simply `http://store.ovi.com/content/23158`. Given a relatively smaller number of applications in the Nokia market place (roughly 130 K), compared to the Android and iOS market places, it can be spanned quite easily. More importantly, the application executable is accessible by simply appending the tag */download* to the content URL. The download, of course, requires a valid login session.

[3] Needless to say, even though we collect app description web pages for all crawled apps, we only download the executable archives of the *free* ones.


Rule
From: TPL
Extract: \"packageName\"\:\"([\w.-]+)\"
Lookup: Android app id
Match
Network Flow
Android app id: com.menueph.apps.utility.smswalk
HST: ads.adlayout.net
TPL: …,"osVersion":"4.3","packageName":"com.menueph.apps.utility.smswalk",…
Search app id hash
com.menueph.apps.utility.smswalk
Rule
From: AGN
Extract: (.+)\/.*?
Lookup: iOS cfbundlename
Match
Network Flow
iOS app id: 281704574
HST: b.aol.com
AGN: AIM/5.1.3.11 CFNetwork/548.1.4 Darwin/11.0.0
Search cfbundlename hash
444081514, 494258199, 281704574
Rule
From: URI
Extract: \/pagead\/conversion\/(\d{9})\/
Host: www.googleadservices.com
Lookup: iOS app id
Match
Network Flow
iOS app id: 466922855
HST: www.googleadservices.com
URL: /pagead/conversion/956378685/
Search app id hash
956378685
Android app id: com.appsbar.GBencher95824
HST: ws.appsbar.com
MET: GET
AGN: Apache-HttpClient/UNAVAILABLE (java 1.4)
PAR: AppID=95824&SecurityKey=f54123c4f5f0ec185067a65fe8fbeeb0&DeviceID=3bed1bcd862646ae&DeviceInfo=Android%2CV2%2C4.0.4%2Cunknown%2Csdk
EPL: []
URI: //Notifications/AppsBar-GetNotifications.php
No identifier exists!
True Negative
Rule
From: TPL
Extract: msid\=([\w.-]+)
Lookup: Android app id
Match
Network Flow
Android app id: com.foundero.countryfacts.Australia
HST: ads.adlayout.net
PAR: …&u_w=384&msid=com.foundero.countryfacts.Australia&cap=a&…
Search app id hash
Not Found!
Identifiers in the flow
Android app id: com.brother.BrotherAR
HST: api.junaio.com
MET: GET
AGN: com.brother.BrotherAR/1.4.1 unknown google_sdk/Android/4.3
URI: /channels/manage/131085
But rule is missing!

(a) Unique Match (b) Fuzzy Match (c) False Positive

(d) True Negative (e) False Negative (f) False Negative

Figure 3: Six kinds of identification results.

The Crawl-Download infrastructure creates two repositories per application platform: (a) an app web page repository (all apps); and (b) an app executable archive repository (free apps). For Android and iOS, these archives are respectively called *apk*'s and *ipa*'s, after their respective extension types. The case for Nokia is a little different where several distinct application executable types (and in fact media files) are found in the market place. Predominant amongst these are Java based executables, called the *jad*'s. Henceforth, all Nokia apps referred to in this work are *jad* apps.

5.2 Identifier Extractor

5.2.1 From application web pages

From the app web pages, we extract the **app id** and the **app name** identifiers. This can be accomplished easily by parsing for static tags in the HTML page. SAMPLES creates two hashes for these two identifier types, on a per platform basis. The hash for identifier type **app id** contains within it the **app id** as key, and a constant 1 as the corresponding value. On the other hand, the hash for **app name** contains the **app name** as the key, and the corresponding **app id** as its value. In the case of key-collision, i.e. more than one app with the same **app name** string, the value field is an array of the corresponding app ids. Clearly, in the matching and look up phase, such a match will qualify as **fuzzy**.

5.2.2 From application executable archive

The app executable archive for a given mobile application contains, amongst other things, the application binary and a metadata file. The metadata file — *AndroidManifest.xml*, *info.plist* and *Manifest.xml* respectively for the Android, iOS and Nokia platforms — is an XML file that provides behavioral specifications and access permissions required by the app for proper functioning. It may also contain a variety of other identifiers in the form of XML fields and their corresponding values (such as the CFBundle identifiers for iOS and those assigned by third party services [23]).

SAMPLES treats the metadata file for each app individually to extract potential application identifiers through a statistical heuristic, thereby obviating the need for manual inspection (as employed by [23]). Using a DOM parser, SAMPLES extracts all XML fields and their corresponding values from the metadata file. Each field name (XML-tag) is considered to be a potential identifier type, and the corresponding value is considered to be a potential identifier string for the app in question. As with the identifier type **app name**, SAMPLES builds a hash for each potential identifier type, with the potential identifier string as key, and the app id as its value. SAMPLES then selects out of these potential identifier types by two criteria.

1. **Universality**: The fraction of apps in the app repository in whose metadata file this potential identifier type (XML field) is found. In other words, the length of the hash divided by the cardinality of the app executable archive repository. Higher the value, more likely the XML field is an identifier type. For instance, the identifier type **app name** is likely to have this fraction is exactly 1 (as all apps have an **app name**).

2. **Uniqueness of values**: The average number of values (i.e. **app ids**) per key in the hash. The lower this number, the greater the likelihood that the XML field is an identifier type. Once again, for the identifier type **app name**, this value is close to 1 (very few apps, if any, share the exact same name for the same platform).

A good example of an identifier type discovered through this statistical sampling is the *pid* XML field found in the manifest files for Nokia apps. As we shall see, even if the identifier extractor errs by selecting an XML field that is not really an identifier type, the impact is mitigated in the dynamic analysis phase where each prospective rule goes through a validation step. This built in checking mechanism helps us prune out extraneous patterns at an early stage.

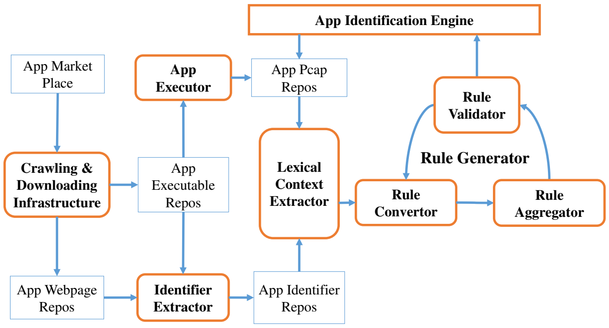


Figure 4: SAMPLES: system architecture and operational life cycle.

5.3 Application Executor

SAMPLES also comes equipped with an app executor which can automatically execute downloaded apps in the emulator or real devices and capture generated HTTP flows. App executor will run each app and generate multiple UI events in order to generate network traffic with more diversity. Then the captured traffic is processed by a deep-packet-inspector (DPI) to extract HTTP flows. For each HTTP flow, We only store the HTTP header and the first 1024 bytes of payload to save storage space.

We can directly control the execution of Android apps by Android automation tools, like monkey-runner [5]. However, there are no similar automation tools for iOS and Nokia platform. So we have to rely on UI automation tools on PC, like AutoIt [2], to control app execution. For Nokia, we could run apps in a emulator and trigger UI events by AutoIt scripts, while for iOS, we have to project the devices' screen to a VNC viewer to export UI and use automation tools due to a lack of emulator support for iOS platform, which is essentially the same infrastructure as presented in [30].

Due to scalability concerns, we do not run a complete and exhaustive app execution to invoke every possible app execution path like [11] requires, which is time-consuming and not scalable. Instead, we allow incomplete app execution to ensure our app executor can scale with large number of apps. In fact, this approach is enough to capture enough useful network flows because we do not rely on a single app to generate all kinds of flows, as rules are common patterns in flows of multiple apps.

5.4 Lexical Context Extractor

First, we perform lexical context extraction on the training flow set. Given a platform, lexical context extraction is performed for each identifier type found in the identifier extractor phase for the apps in the training set for that platform individually. For illustration purposes, we will use the **app id** for the Android platform exclusively for the remainder of this section. The described methodology applies to all other identifier types as well as the other two platforms.

As shown in Figure 2, for a given Android app in the training set, with **app id = statsheet.statblogs.FAUReview**, the lexical extractor searches for the existence of the app id at a per flow basis from the flow set of that app. Once found, it extracts the lexical context with {**Where**: **XRW**, **What**: prefix=^, suffix=$, **Which**: **Android app id**}. Note that each **what** clause, which comprises of a prefix and a suffix part, is terminated at either a predefined delimiter set of the given field (e.g. PAR) or the beginning or the end of of that field (e.g. APL_CX or XRW). For standard HTTP fields, the delimiter set is defined in HTTP grammar. For example **&** is the delimiter in URI-query (PAR field). For other non-standard fields, the delimiter set can be automatically inferred from the field value [25].

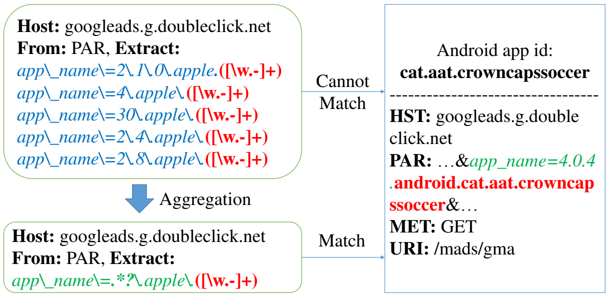

Host: googleads.g.doubleclick.net
From: PAR, Extract:
app_name\=2\.1\.0\.apple.([\w.-]+)
app_name\=4\.apple\.([\w.-]+)
app_name\=30\.apple\.([\w.-]+)
app_name\=2\.4\.apple\.([\w.-]+)
app_name\=2\.8\.apple\.([\w.-]+)
Cannot Match
Aggregation
Host: googleads.g.doubleclick.net
From: PAR, Extract:
app_name\=.*?\.apple\.([\w.-]+)
Match
Android app id:
cat.aat.crowncapssoccer
HST: googleads.g.double click.net
PAR: …&app_name=4.0.4 .android.cat.aat.crowncap ssoccer&…
MET: GET
URI: /mads/gma

Figure 5: Rule aggregation: EXTRACT clause.

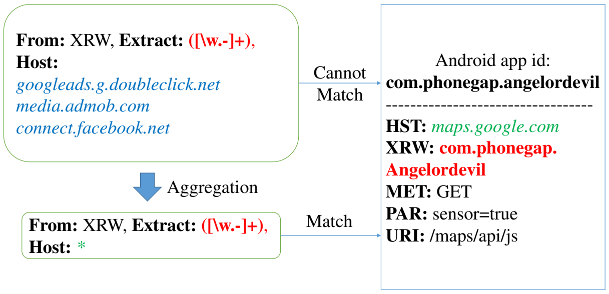


Figure 6: Rule aggregation: IF clause.

5.5 Rule Generator

5.5.1 Create rules from lexical contexts

As an initial step, we directly convert lexical contexts extracted from training flows into rules by generating regular expression in **Extract** clause and attaching host names, identifier class and other information to the rule. For example, the lexical context {**Where**: **PAR**, **What**: prefix=\b msid=, suffix=\b, **Which**: Android app id} will be converted into

APP-IDENT-RULE 3. ***Extract** FROM **PAR**,*
***msid=([\w.]+)** , AND **Lookup** IN {**Android app id**},*
*IF **HST**: **googleads.g.doubleclick.net**.*

Note that instead of simply using ".*" to catch identifier, the characteristic of the identifier class is encoded in the matching pattern. For example, Android app id can contain any number of word, number, underscore and the dot, so the pattern will be **[\w.]+**. And iOS app id is a 9 digit number, so the pattern will be **\d{9}**. This will increase the identification accuracy of the generated rules.

5.5.2 Aggregate existing rules

Although rules converted from lexical contexts can directly identify network flows, their matching conditions are usually too strict, thus leading to the following two problems.

Low coverage: as shown in Figure 5, none of those rules in the first rule set can match that network flow. In fact, it is because those rules require to exactly match the version number between "=" and "apple" in PAR field, which is not necessary. Similarly, in Figure 6 we also cannot identify the given flow only because this flow is under a different host. In fact, with a limited training set, we cannot find all possible version numbers and host names. Therefore, it is always possible to miss such flows in the network traffic, which contributes to coverage loss.

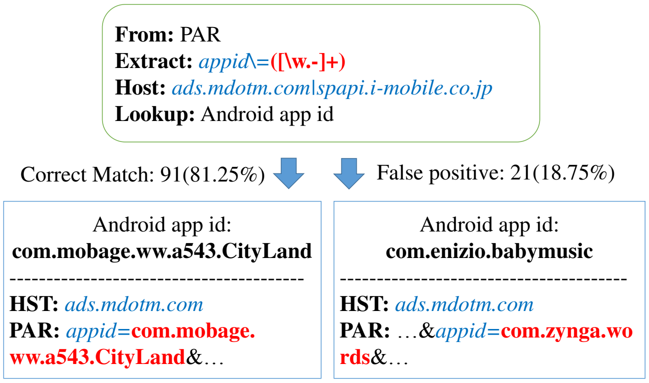

From: PAR
Extract: appid\=([\w.-]+)
Host: ads.mdotm.com|spapi.i-mobile.co.jp
Lookup: Android app id
Correct Match: 91(81.25%)
False positive: 21(18.75%)
Android app id:
com.mobage.ww.a543.CityLand
HST: ads.mdotm.com
PAR: appid=com.mobage.ww.a543.CityLand&...
Android app id:
com.enizio.babymusic
HST: ads.mdotm.com
PAR: ...&appid=com.zynga.words&...

Figure 7: Example of low accuracy rules that should be pruned.

Low throughput: again as shown in Figure 5, we have to go through all 5 "app_name" rules to identify the HTTP flow with such pattern. Therefore, we must spend a longer processing time to find a match, which is not necessary and thus lower the identification throughput.

Obviously, it is the redundant matching conditions that hinder both the coverage and throughput of the rule set. In order to solve this problem, SAMPLES will aggregate and merge similar rules into more general rules to remove redundant matching conditions. Specifically, we can aggregate rules on two clauses:

1. **EXTRACT**(cf Figure 5): To aggregate the EXTRACT clause, we use sequence alignment, a technique which is widely used in bioinformatics area [7], to calculate the similarity between regular expressions. Then we group rules into different clusters based on the alignment sequence. The alignment sequence of two rules contains three parts, MATCH (two tokens are the same), MISMATCH (two tokens are different) and INDEL (one token aligns to a gap in the other rule). In our implementation, we generate new rules by strictly keeping The MATCH and replacing MISMATCH and INDEL with a wildcard **.*?** , as shown in the second rule set in Figure 5.
2. **IF**(cf Figure 6): Here we only take host condition as an example. We first cluster rules with the same FIELD, LOOKUP and EXTRACT conditions together. Then we build a prefix-tree on host names based on their DNS name and generate new rules by keeping the longest common substring between hosts from root to leaf and replacing all the different prefix by a wildcard.

5.5.3 Validate rules

To ensure the quality of generated rule set, SAMPLES will validate each rule by identifying a special flow set, called validation set, whose flows contains source app id as the ground-truth. Then we record the coverage and accuracy of each rule in its identification results. To better explain this process, we generate rule set using 5000 training apps from those three platforms, and plot the validation result of each rule as a point on Figure 8. There are three kinds of rules.

High coverage and high accuracy rules (top right part of Figure 8): these are the rules, out of the constructed rule set, that we want in our rule-based identification engine. Besides that, to improve identification throughput, we assign higher priority to higher coverage rules as they are more likely to be matched first. And if an aggregated rule passes the validation, all the simple rules which generate this rule are also pruned out because their matching range are smaller than the aggregated rule.

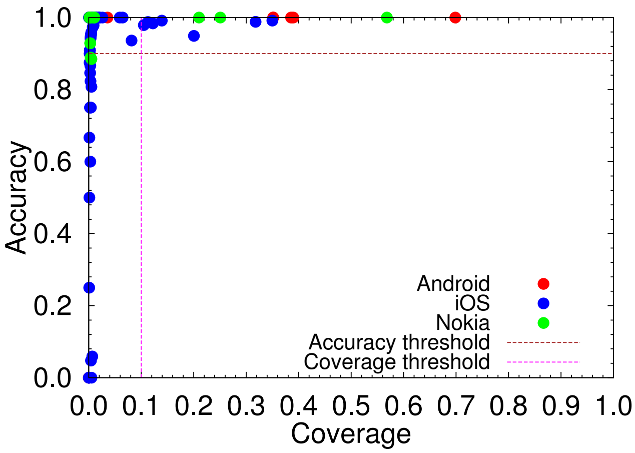


Figure 8: Coverage and accuracy of each individual rule from three platforms (5000 training apps).

Low accuracy rules (bottom part of Figure 8): some rules will generate large ratio of false-positives during identification process, which will severely undermine the reliability of identification results. Figure 7 shows such an example, which misclassified 21 apps out of 112 identified apps. Such low accuracy rules could result from aggressive aggregation. For example, if we merge all initial rules into **Extract: .***, then it will definitely cause lots of false positives. However, it is also likely that those rules are affected by app developers, who might fill corresponding HTTP flow with inaccurate app information. For example, the false positive shown in Figure 7 could be caused by a careless developer forgetting to update the corresponding code in his app or an intentional developer who forks several apps with the same ad service identifier. Whatever the reason is, such low accuracy rules will interfere with our identification results and should be pruned out using an **accuracy threshold**.

App-specific/low coverage rules (top-left part of Figure 8): after rule aggregation, there are still some rules which can only match to a small fraction of apps. Some of them can even match to only 1 app, which is therefore named as **app-specific rules**. Such app-specific rules exist mainly because SAMPLES tries to find every occurrence of app identifier in HTTP flows but sometimes those identifiers coincide within an random context that never indicates app identifier for other apps. For example, here is an app-specific Android rule.

APP-IDENT-RULE 4. ***Extract** FROM **HST**,*
***Pattern:www\.(\w.)**, AND **Lookup** IN {app id},*
*IF **HST**: **www.i9suaradio.com.br**.*

It is obvious that this rule does not work for most apps because few apps put its identifier directly in the host name. But it is still generated when SAMPLES finds app *i9suaradio.com.br* generates an HTTP flow with host name *www.i9suaradio.com.br*.

Although most of those rules will eventually either be pruned out after rule validation or does not affect identification results, we still remove them from rule set prior to rule validation for two performance reasons. First, the average app identification rate decreases with the size of rule set, and there is no need to include those rules because they do not contribute any significant app coverage gain. Second, validating all app-specific rules will greatly increase the time overhead on rule validation since the majority of rules are app-specific, as shown in §6.5 Therefore, we purpose two heuristics to prune out those rules. First, any **app-specific rules** will be prune out. Second, if their coverage is lower than a **coverage threshold**, it will be recognized as low coverage rules and be pruned out.

5.6 Application Identification Engine

SAMPLES comes equipped with an app identification engine that performs rule-based identification on network flows. Therefore, SAMPLES can be seamlessly integrated into any middleboxes, like DPI components, firewalls or gateways, that perform traffic classification, without impeding the overall throughput. SAMPLES takes unidentified network flows as input and for each network flow, the app identification engine tries to find a matching rule. Once a match is found, the flow is labeled as **identified**, else it is deemed **not identified**.

Note that the app identification is mainly determined by the rule set. In order to achieve higher coverage, we may add many different rules into the rule set to catch various lexical patterns. However, in the basic implementation we have to run a linear search in the rule set in order to find a match. According to §2.1, since about 35% of flows in network traffic are not identifiable, the average identification time per flow is proportional to the length of rule set. So as the number of rules increases, the processing throughput will gradually drop down and lower the identification engine's performance.

However, as our rules contain many additional matching conditions besides the main **EXTRACT** clause, we could choose a smaller rule set for each incoming flow, thus improving the processing throughput. For example, we can split the whole rule set in different subsets, each of which contains rules matching a particular host name. During identification, we can first calculate the hash value of incoming flow's host name and only use the corresponding subset for rule-based identification. With this optimization, we can make the processing throughput scalable with the number of rule set.

6. EXPERIMENTS

We evaluate SAMPLES on two aspects: rule set quality (app coverage, app accuracy and app identification goodput) and system performance (overhead of rule generation). To generate dataset for training and testing, we run SAMPLES's crawling and downloading infrastructure on Google Play store, iTunes store, and Nokia OVI store and collected $651K$ Android apps, $68K$ iOS apps, and $10K$ Nokia apps, respectively. By executing each downloaded app in software emulators, we captured $13,003K$ Android flows, $478K$ iOS flows and $39K$ Nokia flows. We use flows from $22K$ Android apps, $32K$ iOS apps, and $5K$ Nokia apps as the training set and the rest as the testing set.

6.1 Comprehensiveness of generated rules

We first evaluate the rule set generated from the complete training set for each platform. Table 3 summarizes the number of rules yielded by the engine, and shows a few representative rules for each platform. As one can see, SAMPLES generates many important rules because it finds the general traffic pattern around app identifiers. For example, the #1 Android rule is found by SAMPLES because after Android 2.2 (Froyo), Android webview includes app's package name into `X-Requested-With` field instead of a general "XMLHttpRequest". The #1 iOS rule is discovered since iOS has a guideline that the `User-Agent` field should contain the app identifier [6]. There are also many rules related to various advertisement/analytic services. Those rules are discovered simply because the corresponding services are widely used by mobile apps. If a new service becomes popular in the future, our rule generation engine can also find it by the same analysis.

We also compare the generated rule set with a rule set manually crafted by human analysts to evaluate the comprehensiveness of rules generated by SAMPLES. The manually crafted rule set includes 16 Android rules, 68 iOS rules, and 2 Nokia rules. The rule set generated by SAMPLES has covered 12 of those manually crafted Android rules, 43 of those iOS rules, and all 2 Nokia rules.

6.2 Effectiveness of the generated rules

Here, we evaluate the quality of the conjunctive rule set generated by SAMPLES by verifying whether the rules can correctly identify the apps from in testing set at a per flow granularity. The results are presented in Figure 9 (first column in each plot). One can see that the rules generated by SAMPLES produce highly reliable identification. Over 99% of the flows' originating apps can be correctly identified. Furthermore, the rule set also covers most of identifiable apps in the network. We can identify 74%, 62% and 63% apps on Android, iOS and Nokia test set, respectively. As a comparison, there are only 76.8%, 67.4% and 63.2% apps among those three datasets that is identifiable, respectively. In another word, SAMPLES can identify over 90% of identifiable apps among three platforms

Note that there are still around 2.8% Android apps and 5.4% iOS apps that have app identifier but cannot be identified by our rule set. The reasons are threefold. First, the training data is not comprehensive enough. When executed in emulators in limited amount of time, some apps in the training data do not generate enough flows to exhibit persistent lexical patterns. We believe that this problem can be mitigated if we periodically update the rule set with new training data. Secondly, pruning out low accuracy rules may lead to coverage loss, as shown in Figure 7. However, in most cases it is worthwhile to ensure high accuracy at the cost of minor coverage degradation. Similarly, the removal of app-specific/low coverage rules can lead to coverage loss as well. We will provide more discussion on this issue later in §6.5.

6.3 Identification goodput

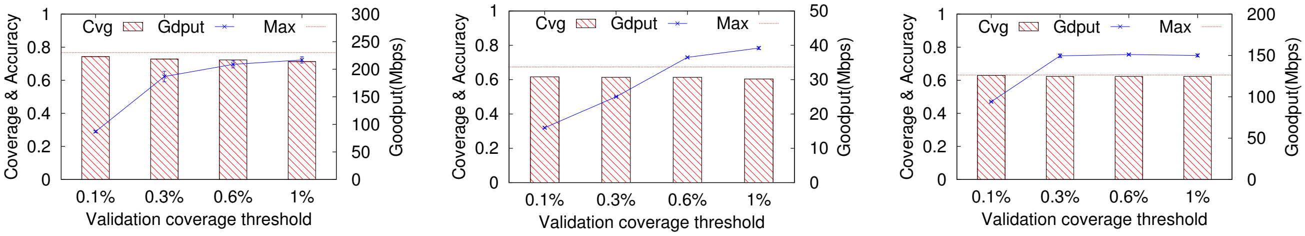
Next, we evaluate the system *goodput* in exploiting the generated rule set to identify the originating apps of network traffic. Note that the measured *goodput* should be much lower than the throughput on real network traffic since our dataset has only the HTTP headers and the first 1024 bytes of HTTP payload.

As shown in the first column of Figure 9, SAMPLES's app identification engine can process pure HTTP flows at 87 Mbps. To achieve even higher processing throughput, we can shrink the rule set by adjusting the coverage threshold in validation phase. As shown in Figure 9, when the coverage threshold increases from 0.1% to 1%, the identification goodput of Android rule set increases from 87 Mbps to 217 Mbps, while the coverage drops from 75% to 70%. Similarly, we can achieve at most 40 Mbps and 150 Mbps goodput for iOS and Nokia platform with coverage dropping from 62% to 61% and from 63% to 62.4%, respectively. Obviously, there is a tradeoff between coverage and identification goodput. Having a smaller rule set increases the system identification goodput, at the expense of coverage.

We also notice that the goodput of running iOS rule set is smaller than the goodput of running Android and Nokia rule sets. For example, with 1% coverage threshold, running the iOS rules is 5 times and 3 times slower than running the Android and Nokia rules, respectively. This is mainly due to the characteristic of iOS rules. There are more iOS rules without host matching conditions, and more iOS rules are referring to popular HTTP fields like `User-Agent` that can be found in almost any flow in the test set. Therefore, iOS flows will be checked by more rules. Our experiments show that each iOS flow needs to be checked by at most $3-4$ rules on an average before it can be identified. Similarly, for Android and Nokia platforms the number of rule checks required per flow is $1-2$, on an average.

Pri.	Lookup	Host	From	Extract	Not-Matching
Platform: Android, Number of rules: 51					
1	app id	*	XRW	([\w.]+)	XRWHttpRequest
2	app id	*	PAR,REF	msid\=([\w.]+)	the.nebula
3	app id	googleads.g.doubleclick.net	PAR,REF	app_name\=.*?\.android\.([\w.]+)	
Platform: iOS, Number of rules: 64					
1	cfn,cfe,cfd[4]	*	AGN	(.+)\/.*?	Safari, ..., Mail
5	app id	mob.adwhirl.com	EPL	\"key\"\:\"(\d{9})\"	441502580
Platform: Nokia. Number of rules: 12					
1	app name	*.vserv.mobi	EPL,PAR,TPL	mn\=(.+)	
8	app name	www.google-analytics.com	AGN	(.+)\/1\.0\..*?	

Table 3: Rules generated by SAMPLES for Android, iOS, and Nokia with 10% accuracy threshold, removing app specific rules and 0.1% coverage threshold. Here cfn, cfe, cfd stand for cfbundlename, cfbundleexecutable, and cfbundledisplayname, respectively.



(a) Android

(b) iOS

(c) Nokia

Figure 9: Quality and performance of rule set built with entire dataset, 10% accuracy threshold and 0.1% coverage threshold. Cov and max stand for the ratio of apps identified by SAMPLES rule set and all identifiable apps in the test set. Gdput stands for identification goodput. Bars and lines are related with coverage and goodput, respectively. Note that the false positive ratios are all less than 1%.

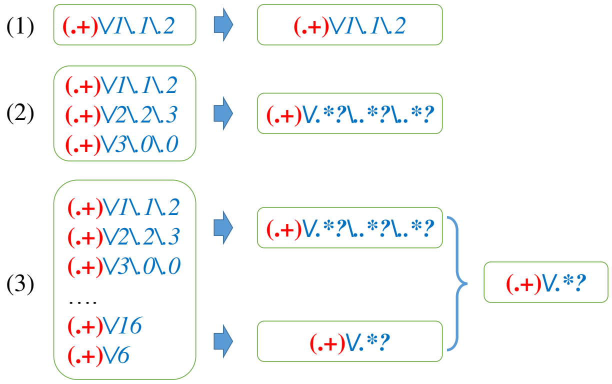


Figure 11: How diversity helps building more essential rules.

6.4 Impact of sampling training sets

We now study the impact of training set size on the coverage and accuracy of SAMPLES. Randomly sampling 200, $1K$, $5K$ and $10K$ apps from each platform's training set [4], we repeat the aforementioned experiments three times. We calculate the app coverage, accuracy, and identification goodput for each experiment. The results are plotted in Figure 10 and it provides us a few interesting insights. We can first confirm that a small sample of apps from market place are enough to build a comprehensive rule set for all three platforms. In fact, even the $10k$ apps represent a mere 1% of the apps in Google Play or iTunes store, both of which have more than 1.3 million apps each [1, 3].

More significantly, for the Android and Nokia platforms, 200 identifiable training apps are enough for building a rule set with similar coverage and accuracy as the one in § 6.1. However, the app coverage of iOS rule set gradually increases with the number of training apps until there are 5000 training apps. That is mainly because the highest priority rules for identifying Android and Nokia apps are all *simple* rules, like `XRW` rule and `msid` rule for Android and `mn` rule for Nokia. Network flows matching those rules are prevalent in the traffic generated by Android and Nokia apps. Hence, even small number of training apps provide sufficient chance to generate the network flows for SAMPLES to learn those rule. However, the highest priority rules for iOS are usually *complex* rules with pattern ".*?" in their regular expressions. Hence those rules can only be generated by rule aggregation. As shown in Figure 11, we are more likely to build rules with essential pattern as more diverse rules are extracted by more training apps.

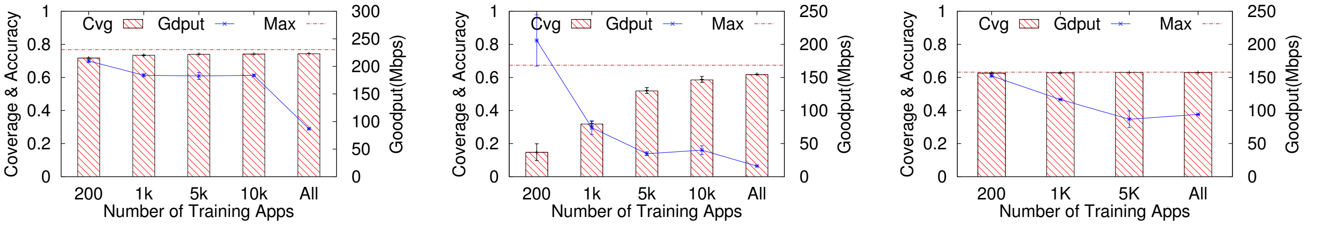
	# covered apps				
	1	2-10	11-100	101-1000	> 1000
Android	7748(79,3%)	1564	379	67	7
iOS	86187(88.7%)	10490	1173	67	0
Nokia	107(72.3%)	31	7	1	2

Table 4: Number of lexical contexts that can cover given number of apps.

6.5 Time overhead of rule set generation

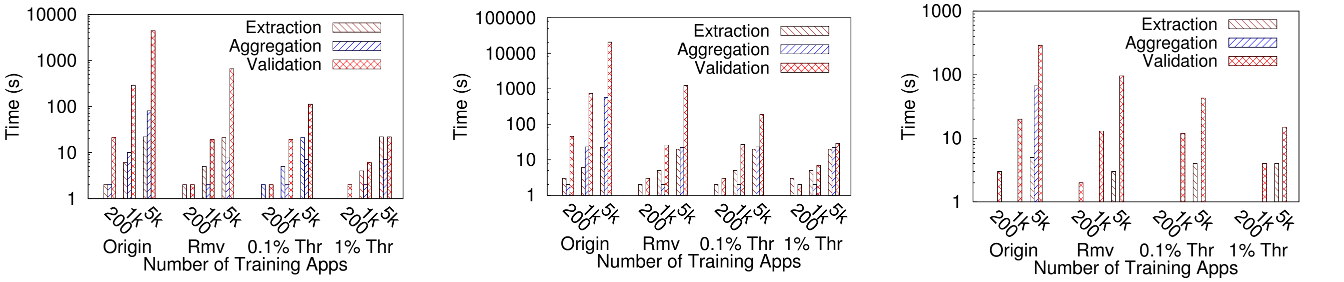
We next evaluate the overhead of generating the rule sets. To that end, we run SAMPLES with 200, $1K$, and $5K$ training apps from all three platform and measure the time consumption of the rule generation phase. We also evaluate how the two optimizations, i.e., *app-specific rule removal* and *coverage threshold*, affect the rule generation time. As shown in Figure 12, without any optimization, the rule generation time increases rapidly with the number of training apps. Besides, the processing time differs largely for apps from different platforms. For example, it takes more than an hour to process 5000 Android apps; processing 5000 iOS apps takes about six hours. The major part of the processing time is due to rule val-

[4] For Nokia, we sample at most $5K$ apps because there are only $5,123$ Nokia apps in the training set.



(a) Android (b) iOS (c) Nokia

Figure 10: Impact of sampling training set with 10% accuracy threshold and removing app-specific rules. Cvg and max stand for the ratio of apps identified by SAMPLES rule set and all identifiable apps in the test set. Gdput stands for identification goodput. Bars and lines are related with coverage and goodput, respectively. Note that the false positive ratios are all less than 1%.



(a) Android (b) iOS (c) Nokia

Figure 12: Time overhead for each rule generation step. Origin means not removing any rules. Rmv means only remove app-specific rules. 0.1% Thr and 1% Thr means removing both app-specific and low coverage rules.

idation, which requires each rule to be validated on the entire validation set. Hence, the computational complexity is $O(M \times N)$, where M is the length of rule set and N is the size of validation set. In general, more rules can be learned from a larger training set. So the validation time increases quadratically with the number of training apps.

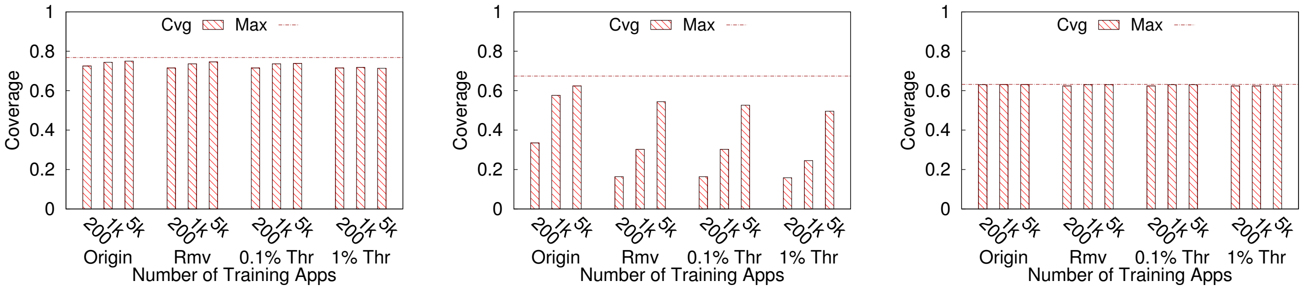
Our experiments also demonstrate that each of the two heuristic optimizations can shorten the validation time by one order of magnitude, as plotted in Figure 12. The reduction in rule generation time is mainly due to the removing of those app-specific/low coverage rules in rule generation. As shown in Table 4, more than 70% lexical contexts are app-specific. About 75% of the rest of the lexical contexts can cover only 2-10 apps. These can be considered as low-coverage rules and removed in order to speed up the rule generation. We also evaluate how discarding low coverage rules affects the coverage on the test set. As shown in Figure 13, the coverage of identifying iOS apps is more sensitive to removing of the low coverage rules, especially when there are 1000 training apps. This is because the top coverage rules in Android and Nokia platform are simple rules that can be easily learned, while iOS's top coverage rules are complex rules requiring more training samples to be learned. When there are not enough training samples to generate complex rules, e.g. with only 1000 training apps, the low coverage rules actually contribute to the most of app coverage. With more training apps, e.g. 10000 apps as shown in Figure 10, the impact of those two heuristics to the coverage of identifying iOS apps becomes marginal, and the benefit in saving the rule generation time becomes more significant.

6.6 Comparison with previous works

We now compare SAMPLES with two previous approaches: one based on identifiers associated with advertisement services as proposed in [23] and another on identifiers found in the user-agent field (cf. [29]). The advertisement services based approach obtains specific keys in HTTP headers generated by ad-libraries (e.g. double-click, Migital etc.) as app identifiers. Similarly, the user-agent based approach directly extracts app identifiers from the user-agent field to identify iOS apps. Both approaches are state-of-the-art solutions for mobile app traffic classification. We implement both approaches and use the same test set to evaluate their identification performance. To match the implementation details in [23], we focus on two popular ad services (Google Ads and Smaato) in the ad service approach. For user-agent approach, we manually prune out a list of generic strings as app identifiers, like Safari, iPhone, iOS, etc.

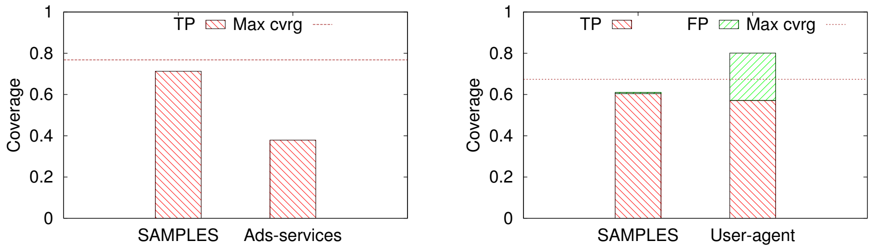
As shown in Figure 14, SAMPLES outperforms both existing approaches on either app coverage (Android) or identification accuracy (iOS). There are mainly two reasons for this. First, both approaches essentially yield a subset of rules that are generated by SAMPLES. Besides the simple key-value pair or user-agent field, SAMPLES generates more sophisticated rules based on other HTTP headers and payload. So the rule set generated by SAMPLES can identify apps whose identifiers appear in several lexical contexts, which explains the coverage gain. Second, the rules generated by SAMPLES are refined by rule validation so they are less prone to generating false positives. Take the User-Agent field as an example. Some strings extracted from this field, like iPhone, Game and Books, are not app identifiers and should be marked as exceptions. In previous work, one has to look for such exceptions by some heuristics, so it is hard to be comprehensive. While in SAMPLES, we can easily get a more complete exceptions set for this rule by providing more training apps. This explains why SAMPLES can achieve lower false-positive ratio on iOS.

To summarize, SAMPLES provides a general, light weight, cross-platform framework, for identifying mobile apps in network traffic with markedly better performance than other competing solutions.



(a) Android (b) iOS (c) Nokia

Figure 13: Two optimization's influence on app coverage. Note that the false positive ratios are always smaller than 1%.



(a) Android (b) iOS

Figure 14: SAMPLES's evaluation results compared with ads services approach for Android [23] and user-agent approach for iOS [29] (Note: the false positive ratios are less than 1% for both methods in (a)).

7. RELATED WORK

Several network management tasks require fine grained traffic classification. Consequently, a number of techniques have been proposed over the years to address this problem; a survey of these may be found in [9, 18]. With the advent of the *mobility-era*, however, the variety of devices and applications has exploded at an unprecedented rate [26], most of which use HTTP (and more recently HTTPS) as a communication protocol. Thus, protocol/port based methods are of little use. The need to identify mobile applications in network traffic, preferably at a per-flow granularity, has therefore been tackled, at least in part, in several recent studies [12, 13, 16, 17, 19, 20, 24, 27]. Alas, most of these techniques are either based on *ad-hoc* heuristics, are too platform specific or computationally too expensive to be of practical use. For instance, in [29], classification and identification of mobile applications at a per flow granularity is limited only to those flows which contain an identifier in the `User-Agent` field of the HTTP header. This happens to be the case for a very small fraction of mobile applications (cf. comparative analysis in §6.6) and mostly for those from the Apple iOS market-place. Similarly, in [23] manual inspection of the *AndroidManifest.xml* is suggested to extract *third-party* assigned identifiers (such as ad-networks). These are then used to build a knowledge base of key-value pairs that can potentially be found in network traffic. Clearly, both solutions lack comprehensiveness and are dependent on manual expertise. To tackle this, some studies suggest active network trace generation [11, 27, 28, 30]. However, the classification methodologies suggested in [27, 28, 30] still require human involvement. A notable exception is NetworkProfiler [11]: a system that generates state machine based signatures, per app, through *exhaustive* execution (i.e. emulating all potential network activity for the app *a priori*). Although automated and comprehensive, this approach is computationally expensive; and hence not scalable. Also, creating one state machine per app implies that each flow needs to be matched exhaustively across the set; rendering live deployment infeasible.

8. CONCLUSION & FUTURE WORK

In this work, we developed **SAMPLES**, a supervised framework for generating conjunctive rules for identifying mobile applications in network traffic at a per-flow granularity. SAMPLES uses a relatively small sample of mobile applications from three prominent platforms (Anrdoid, iOS and Nokia), to extract both the identifiers as well as the lexical context in which they are found in the traffic; which together yield the rule-set. Through experiments over a vast corpus of mobile applications (over 700 K), we have demonstrated the comprehensiveness and applicability of our approach across all three platforms. In particular, the coverage is significantly higher than the other state-of-the-art methodologies from literature. From a system performance standpoint, SAMPLES achieves desirable *goodputs* of 217 Mbps, 40 Mbps, and 150 Mbps respectively for the Android, iOS, and Nokia datasets. Although conducted in a controlled laboratory environment, these results are healthy indicators for a live deployment scenario.

In the immediate future, we intend to expand SAMPLES to include other mobile platforms (such as BlackBerry and Windows Mobile). We expect reasonably similar results — in terms of coverage, accuracy and performance — to those obtained in this work. Also, there is scope for extending the general methodology proposed in this work, that of exploiting lexical contexts around identities, to identify network events (e.g. download, upload, chat etc.) responsible for a given flow. All these we propose for future work.

9. ACKNOWLEDGEMENT

We express our sincerest gratitude towards the anonymous reviewers whose comments helped improve the quality of this work, and our shepherd for guiding us through the revisions. Also, our thanks to Stansilav Miskovic and Ramya Gadiayaram for their help with the data collection activity and preliminary discussions. This research was supported by the National Science Foundation grants CNS-1059372, CNS-1345226 and CNS-0964545, as well as the Narus Fellow Research Program.

10. REFERENCES

[1] App store (ios) - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/App_Store_iOS.

[2] Autoit. https://www.autoitscript.com/site/autoit/.

[3] Google play - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Google_Play.

[4] HTTP header fields. http://en.wikipedia.org/wiki/List_of_HTTP_header_fields.

[5] Monkey runner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[6] Optimizing web content. https://developer.apple.com/library/mac/documentation/AppleApplications/Reference/SafariWebContent/OptimizingforSafarioniPhone/OptimizingforSafarioniPhone.html.

[7] Sequence alignment. http://en.wikipedia.org/wiki/Sequence_alignment.

[8] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic library analysis. In *Proceedings of 14th ACM Conf. on Computer and Communications Security*, 2007.

[9] A. C. Callado, C. A. Kamienski, G. Szabo, B. P. Gero, J. Kelner, S. F. L. Fernandes, and D. F. H. Sadok. A survey on Internet traffic identification. In *IEEE Communications Surveys and Tutorials*, volume 11, pages 37–52, 2009.

[10] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of 16th USENIX Security Symposium*, 2007.

[11] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. Networkprofiler: Towards automatic fingerprinting of android apps. In *Proceedings of IEEE Infocom*, 2013.

[12] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In *Proceedings of 10th ACM Internet Measurement Conference*, 2010.

[13] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[14] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. Acas: Automated construction of application signatures. In *Proceedings of 1st ACM SIGCOMM Workshop on Mining Network Data*, 2005.

[15] M. Korczynski and A. Duda. Markov chain fingerprinting to classify encrypted traffic. In *Proceedings of the 33rd IEEE Infocom*, 2014.

[16] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don'T kill my ads!: Balancing privacy in an ad-supported mobile application market. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, 2012.

[17] G. Maier, F. Schneider, and A. Feldmann. A first look at mobile hand-held device traffic. In *Proceedings of the International Conf. on Passive and Active Measurement*. Springer-Verlag, 2010.

[18] T. Nguyen and G. Armitage. A survey of techniques for Internet traffic classification using machine learning. In *IEEE Communications Surveys and Tutorials*, volume 10, pages 56–76, 2008.

[19] A. Patro, S. Rayanchu, M. Griepentrog, Y. Ma, and S. Banerjee. Capturing mobile experience in the wild: A tale of two apps. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, 2013.

[20] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, 2012.

[21] A. Sapio, Y. Liao, M. Baldi, G. Ranjan, F. Risso, A. Tongaonkar, R. Torres, and A. Nucci. MAPPER: a mobile application personal policy enforcement router for enterprise networks. In *Proceedings of the European Workshop on Software Defined Networking*, EWSDN '14, 2014.

[22] A. Sapio, Y. Liao, M. Baldi, G. Ranjan, F. Risso, A. Tongaonkar, R. Torres, and A. Nucci. Per-user policy enforcement on mobile apps through network functions virtualization. In *Proceedings of the 9th ACM workshop on Mobility in the Evolving Internet Architecture*, 2014.

[23] A. Tongaonkar, S. Dai, A. Nucci, and D. Song. Understanding mobile app usage patterns using in-app advertisements. In *Proceedings of the International Conf. on Passive and Active Measurement*, 2013.

[24] N. Vallina-Rodriguez, J. Shah, A. Finamore, Y. Grunenberger, K. Papagiannaki, H. Haddadi, and J. Crowcroft. Breaking for commercials: Characterizing mobile advertising. In *Proceedings of 12th ACM Internet Measurement Conference*, 2012.

[25] R. Vargiya and P. K. Chan. Boundary detection in tokenizing network application payload for anomaly detection. In *https://cs.fit.edu/media/TechnicalReports/cs-2003-21.pdf*, 2003.

[26] N. Viennot, E. Garcia, and J. Nieh. A measurement study of Google Play. In *Proceedings of the ACM SIGMETRICS*, 2014.

[27] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: Multi-layer profiling of android applications. In *Proceedings of 18th Annual International Conference on Mobile Computing and Networking*, 2012.

[28] Q. Xu, T. Andrews, Y. Liao, S. Miskovic, Z. M. Mao, M. Baldi, and A. Nucci. FLOWR: A Self-Learning System for Classifying Mobile Application Traffic. In *Proceedings of the ACM SIGMETRICS (Poster)*, Austin, Texas, 2014.

[29] Q. Xu, J. Erman, A. Gerber, Z. M. Mao, J. Pang, and S. Venkataraman. Identifying diverse usage behaviors of smartphone apps. In *Proceedings of the ACM Internet Measurement Conference*, 2011.

[30] Q. Xu, Y. Liao, S. Miskovic, Z. M. Mao, M. Baldi, A. Nucci, and T. Andrews. Automatic generation of mobile app signatures from traffic observations. In *Proceedings of the 35th IEEE Infocom*, Hong Kong, China, 2015.