Approximate Matching of Persistent LExicon using Search-Engines for Classifying Mobile App Traffic

Gyan Ranjan, Alok Tongaonkar and Ruben Torres
Center for Advanced Data Analytics,
Symantec Corporation, Mountain View, CA, USA.
E-mail: {gyan_ranjan, alok_tongaonkar, ruben_torresguerra}@Symantec.com

ABSTRACT

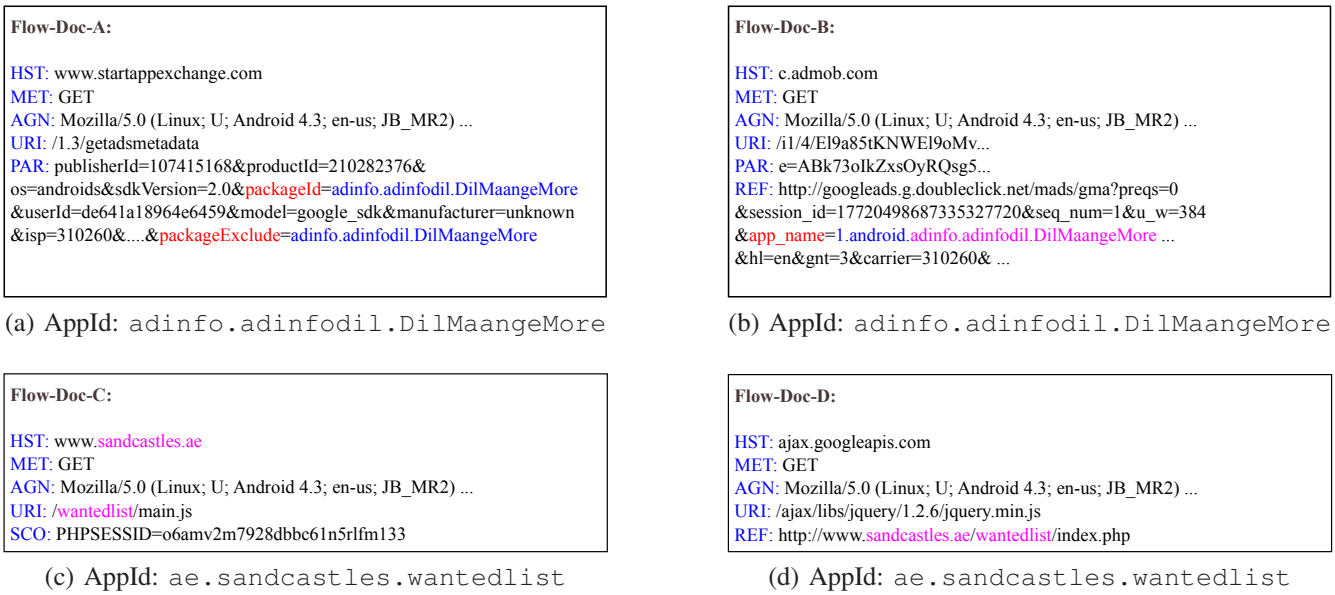
We present **AMPLES**, **A**pproximate **M**atching of **P**ersistent **LE**xicon using **S**earch-Engines, to address the Mobile-Application-Identification (**MApId**) problem in network traffic at a per-flow granularity. We transform MApId into an information-retrieval problem where lexical similarity of short-text-documents is used as a metric for classification tasks. Specifically, a network-flow, observed at an intercept-point, is treated as a semi-structured-text-document and modified into a flow-query. This query is then run against a corpus of documents pre-indexed in a search-engine. Each index-document represents an application, and consists of distinguishable identifiers from the metadata-file and URL-strings found in the application's executable-archive. The search-engine acts as a kernel function, generating a score distribution *vis-'a-vis* the index-documents, to determine a match. This extends the scope of MApId to fuzzy-classification mapping a flow to a family of apps when the score distribution is spread-out. Through experiments over an emulator-generated test-dataset (400 K applications and 13.5 million flows), we obtain over 80% flow coverage and about 85% application coverage with low false-positives (4%) and nearly no false-negatives. We also validate our methodology over a real network trace. Most importantly, our methodology is platform agnostic, and subsumes previous studies, most of which focus solely on the application coverage.

I. INTRODUCTION

Network operators require clear visibility into the applications running in their network for management and measurement tasks; such as billing, resource provisioning, application-level firewalls and access control. Such operations require high fidelity information at a flow level granularity. With the increasing adoption of hand-held devices, such as mobile phones and tablets, as preferred end-hosts for accessing the Internet, the need for a nuanced view is greater than ever before. The advent of wearables (e.g. Apple watch, Google-glass etc.), forebodes sustained continuation of this trend into the foreseeable future. These devices have on them applications, commonly called *mobile apps*, that provide a wide range of functions. Yet, a significant fraction of these applications use the HTTP protocol (and/or HTTPS) for communication [3], [22]. Traditional traffic classification systems, such as those based exclusively on protocol classification [9] or statistical/volumetric properties [14] alone, do not adapt well to this changing landscape. Moreover, with the so called Bring-Your-Own-Device (BYOD) to work phenomenon, the problems that enterprise network managers are faced with have only compounded. Although some recent studies have tried to address the issue [6], [15], [19]–[23], the solutions proposed are either platform specific (e.g. [15], [19]–[21]); or, require partial (e.g. [22], [23]), or exhaustive execution of applications [6] to produce different forms of signatures.

In this work we introduce **AMPLES**, **A**pproximate **M**atching of **P**ersistent **LE**xicon using **S**earch-Engines, to address **M**obile-**A**pplication-**I**dentification (MApId) in network traffic at a *per-flow* granularity. Unlike previous efforts, AMPLES provides a generic and unifying framework, that is platform agnostic, scalable and requires only a lightweight static analysis of application executable archives for training. By transforming MApId into an information retrieval problem, AMPLES uses lexical similarity of short-text-documents as a metric for classification tasks. Specifically, a network-flow, observed at an intercept-point, is treated as a semi-structured-text-document and modified into a flow-query. This query is then run against a corpus of documents indexed in a search-engine. Each index-document represents an application, and consists of distinguishable identifiers from the metadata-file and URL-strings found in the application executable-archive. The search-engine acts as a *kernel* function, generating a score distribution vis-a-vis the index-documents, which determines a match. This extends the scope of MApId to fuzzy-classification — mapping a flow to a family of apps — when the score distribution is spread-out. Through experiments over an emulator-generated test-dataset (400 K applications and 13.5 million flows), we obtain over 80% flow coverage and about 85% application coverage with low false-positives (4%) and nearly no false-negatives. We also validate our methodology over a real network trace with similar coverage and accuracy. To the best of our knowledge, we are the first to use an approximate matching alternative with such success.

The rest of this work is structured as follows: we formalize the MApId problem in §II. In §III, we discuss relevant related work in detail. We then reinterpret the MApId problem as an information retrieval problem in §IV. In §V, we provide the functional architecture and operational lifecycle of AMPLES followed by experimental evaluation in §VI. Finally, the paper is concluded with a discussion of future work in §VII.



(a) AppId: `adinfo.adinfodil.DilMaangeMore` (b) AppId: `adinfo.adinfodil.DilMaangeMore`

(c) AppId: `ae.sandcastles.wantedlist` (d) AppId: `ae.sandcastles.wantedlist`

Fig. 1. Mobile app flow documents produced by two different Android applications. Flow-Doc-A contains two key-value pairs in the PAR field with explicit app identifiers. Flow-Doc-B contains one key-value pair in the PAR field where the explicit app identifier is a sub-string of the value. Flow-Doc-C contains sub-strings of the app identifier split across the HST and URI fields while Flow-Doc-D contains the same sub-strings in the REF URL.

II. Problem Statement, Notation and Assumptions

In this section, we formalize the mobile app identification problem in §II-A. In §II-B, we detail the notation used and working assumptions made during this study.

A. Mobile App Identification in Network Traffic

We have already loosely posed the problem of mobile application identification in the introduction. For formal completeness, we now state the problem precisely followed by a discussion on the coverage objectives and hit-ratio.

Problem 1: MApId$(\mathcal{M}, \mathcal{F})$: Given a set of known mobile app identities, $\mathcal{M} = \{M_1, M_2, ..., M_x\}$ and a set of network flows observed at an intercept point, $\mathcal{F} = \{F_1, F_2, ..., F_y\}$, find a mapping $\forall i : \mathcal{X}(\mathcal{F}, \mathcal{M}) : F_i \rightarrow M_j$, s.t. $1 \leq j \leq x$ and $1 \leq i \leq y$.

In practice, however, the unit of observed network activity at an intercept point (e.g. a router or firewall), is always an individual flow. Hence, any real time mobile app classification must ideally work at a flow level granularity. This leads us to a dichotomy of coverage objectives as described below.

Coverage Objectives and Hit-Ratio: The $MApId(\mathcal{M}, \mathcal{F})$ problem can be addressed with two subtly different coverage objectives in mind, viz.:

1. **Application Coverage**: The aim is to identify all $M_j \in \mathcal{M}$ running in the network. The hit-ratio is defined by $\eta_{AC} = |\mathcal{M}_{Id}|/|\mathcal{M}_{All}|$; where $\mathcal{M}_{Id}$ is the set of identified apps, and $\mathcal{M}_{All}$ is the set of all apps running in the network.
2. **Flow Coverage**: The aim is to correctly identify the application responsible for a given flow $F_i \in \mathcal{F}$. The hit-ratio is defined by $\eta_{FC} = |\mathcal{F}_{Id}|/|\mathcal{F}|$.

It is easy to see that application coverage is a sub-objective of flow coverage. As long as we correctly identify at least one flow generated by each mobile app, the app coverage objective is realized. Hence, we choose flow coverage as the coverage objective and η_{FC} as the hit-ratio in this work.

B. Notation and Assumptions

Network Flow: Unless specified otherwise, a network flow $\mathcal{F}$ is defined as a single HTTP request-response pair. An HTTP flow header, when parsed by a deep packet inspector, has the form of a semi-structured text document comprising fields and textual values (cf. Fig. 1). This is also true for HTTPS headers [17], [18]. Furthermore, as per [3], HTTP still represents a significant fraction of mobile app traffic. Henceforth, we use the term *flow-doc* to refer to an HTTP flow and the field tags in Table I for its constituent fields.

Assumptions: Following are pertinent to our study:

1. **ASCII Content**: All textual content in a flow has been preprocessed and converted to an ASCII format. Non-ASCII characters are removed.
2. **Encrypted Traffic**: We do not deal with encrypted traffic (such as HTTPS) in the wild [11]. However, within an enterprise network, where a network administrator has control over end-points, we have successfully implemented and tested AMPLES to handle HTTPS traffic, using a *Man-In-The-Middle-Proxy* [17], [18], [23]. This makes AMPLES particularly useful for policy enforcement on network access points (such as wifi-APs, routers etc.).
3. **Mobile App Executable**: We assume that the executable archive for an application (paid or free) can be obtained from the marketplace [22]. This is essential for constructing the corpus of index documents (see §V). At present, this assumption holds for all the mobile application platforms.

TABLE I
COMMONLY OCCURRING FIELDS IN THE HTTP FLOW HEADER OF MOBILE APP NETWORK TRACES. FOR DETAILS SEE [2], [4].

Field Name	Tag	Description
`Domain host-name`	HST	The host-name in a URL [**csi.gstatic.com**`/csi?v=3&s=gmob`]
`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`	XRW, ALP, APL_CV	Non-standard fields.

TABLE II
EXAMPLE: SAMPLE `User-Agent` STRINGS.

App-Name	AppId	Platform	`User-Agent`	Identifier String
French App	522076075	iOS	French/1.1 CFNetwork/548.0.4 ...	French
Men in Black 3	**504522948**	**iOS**	**Mozilla/4.0 (compatible; MSIE ...)**	**None**
Dublin Parking Free	acet.dublinparkingfree	And.	Mozilla/5.0 (Linux; U; Android 4.3; ...)	None

III. BACKGROUND AND RELATED WORK

Several recent studies that characterize mobile application behavior [8], [12], [15], [20], have had a need to address the MApId problem in one form or the other. However, as this is done only as a means to a *greater* end, such studies either employ ad-hoc techniques, which work only for specific platforms and a subset of flows, or rely on manually generated and labelled traces (through a dedicated human beings interacting with the applications). An example of the first kind is [21]. While studying usage behavior of smart phone apps in network traffic, the `User-Agent` string is used as an application identifier. Alas, this approach is not universal, particularly if flow coverage is the objective. For instance, for a significant number of flows generated by mobile apps, on iOS or Android, the `User-Agent` strings are generic (cf. Table II), and do not contain any identifying attributes in them. In our emulator generated dataset (see §VI), we observe that less than 40% of iOS flows have identifying attributes in the `User-Agent` strings, while the number is less than 5% for Android.

For the Android platform, a more general approach is suggested in [19], which exploits the presence of application identifiers in advertisement related flows. Such applications have a unique identifier registered with the advertisement service that it partners with. These identifiers can be either explicit (e.g. the unique application id in the form of a package name) or implicit (e.g. identifier assigned by an advertisement service for the app). Implicit identifiers are often found as named key-value pairs in a metadata file called `AndroidManifest.xml` that comes bundled with the application executable. When the app runs, these identifiers feature in the flows that are communicated to and from the advertisement service, possibly for accounting purposes. Thus, identifying these key-value pairs can help identify the app in the network traffic. There are, however, some issues with this approach, particularly from the point of view of a flow coverage objective. First, this approach is restricted to advertisement flows only, and hence cannot help achieve the flow coverage objective on its own. Second, there are many advertisement services in the market-place, and many more may emerge in the future. Clearly, it is difficult to know *a priori* what the key-values for each service is, or where to look for them in a network flow as they can occur in different fields of the HTTP header (cf. Fig 1), depending upon developer idiosyncrasies or api definitions. Moreover, not all the keys that are observed in network traffic are present in the manifest files (e.g. `msid, app_name` etc. for the Double-Click service). The labeling task is thus infeasible at large scales.

In [6] a more holistic approach for application *fingerprinting* has been proposed for the Android platform. The authors suggest creating an individual state-machine per mobile application. In an emulator, with advanced automated clicking behavior, the app under question is *exhaustively* executed to generate a flow-set comprising all possible network traffic for the app. Each flow in the flow-set is then parsed and tokenized using the various fields of the HTTP header (e.g. HST, URI, PAR etc.); and recombined into a state machine. Thus the matching infrastructure has one state machine per application in the marketplace. When a network flow arrives in the network, it is run against this state-machine infrastructure. The flow is run against each state machine, one at a time, and a match ratio is computed in terms of matched path lengths. The flow is then assigned the label of the application for which the matched path length is the longest. In essence, this work does provide a means of realizing a flow coverage objective. Alas, the issue is, once again, of scale. First, the construction of a state-machine requires an exhaustive execution of the mobile application to produce the flow-set. This is not a trivial task given the rate at which the number of mobile applications grow. Secondly, state-machine based matching inherently depends on the ordering of the states.

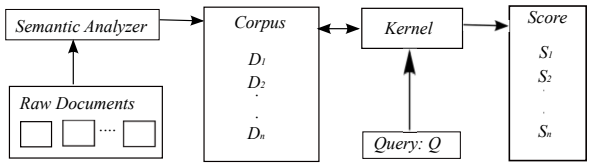


Fig. 2. Schematic: A generic information retrieval system. A query is matched against the corpus of indexed documents to perform a classification task.

An application generated network flow may potentially match multiple transition paths in the same state-machine; thereby necessitating an all-paths exploration per state machine at all times. This is computationally very expensive.

Last, but not least, two recently proposed approaches in [22], [23], attempt to combine the static information from application executable archives and the dynamic information from controlled partial executions to produce signatures and rules respectively. While interesting in their own right, these solutions rely on general-but-strict patterns for an exact matching criteria. **AMPLES**, with its *fuzzy matching* capabilities, can be used in conjunction with these solutions to attain higher flow coverage.

IV. MApId as an Information Retrieval Problem and the Search Engine Paradigm

As noted earlier in §II, a network flow header is simply a semi-structured short-text document. Though seemingly apparent and trivial, this analogy is the cornerstone of our methodology. It helps us transform the MApId problem into a well studied problem in the field of information retrieval (IR): that of indexing and retrieving documents based on a notion of lexical similarity [7], [10], [13].

The central theme in IR is to first process a set of documents using a semantic analyzer and produce a text *corpus*: a set of processed documents that are indexed. To each document in the corpus is attached a class label (based on the objective). Then, given an input document, called a *query*, the task is to infer the class to which the query document belongs. This is done by assessing the lexical similarity between the query and the documents in the corpus. The class of the query document is decided based on which document (or documents) is/are the most likely to represent the textual content of the query. The rationale behind the classification based on text similarity is somewhat simple and elegant. The maximum similarity score between two documents is attained if they are the same. The converse inference is also of interest: higher the similarity scores between two documents, more likely they are to be the same [1]. The *goodness* of such an inference is solely based on the mathematical function that determines the similarity score. This function is fashionably called the *kernel*.

But how does all this relate to the task at hand; the one defined as MApId in Problem 1? Take, for example, the four flow-documents presented in Fig. 1. By inspection, we know that flow-doc-A is more similar to flow-doc-B as compared to flow-doc-C and flow-doc-D. This, despite the fact that package name identifiers in flow-doc-A and flow-doc-B are assigned to different keys. Thus, if a corpus had flow-docs A, C and D in its index, and flow-doc-B were to be queried against it, we expect to get a higher match with flow-doc-A, which would infer flow-doc-B to be generated by `adinfo.adinfodil.DilMaangeMore` as opposed to `ae.sandcastles.wantedlist`.

Consider then that a hypothetical *oracle*-like corpus, denoted henceforth as $\mathcal{F}^*$, exists, that contains all possible flow documents that can be produced by any mobile application in the mobile universe. Moreover, for each individual flow document $F \in \mathcal{F}^*$, the name of the application generating the flow is known *a priori*. We load this *oracle*-like corpus in a commercial search engine (e.g. Apache-Lucene [5]). Then, any given flow F_i observed at a network intercept point, can be run as a query against this hypothetical search. The search engine computes a similarity score between F_i and every flow in F_k^*; thereby generating a score distribution. Therefore, the search engine is, indeed, a *kernel* [16].

Clearly, the fact that the *oracle*-like corpus contains all possible flow documents in it, there exists exactly one document in $\mathcal{F}^*$, with which the highest score is attained. This document must be F_i itself and hence the app id can be inferred. Thus, in the ideal, hypothetical case, with a labelled *oracle*-like corpus the MApId problem can be solved for the flow coverage objective perfectly. But of course, we do not have such an *oracle*-like corpus of network flow documents for every single mobile application in the universe. Even if we were to relax the universality constraint, and consider only a subset of mobile apps, the issue of exhaustive execution of each mobile application is daunting. And execution of apps, exhaustive or partial, for corpus creation is precisely what we want to avoid. As we shall show in the next section, §V, a good working approximation to the desired corpus can be obtained through analyzing metadata and source-code analysis alone. Also, the resulting corpus contains one index document per mobile application as opposed to one per possible flow.

Secondly, there is the question of a good similarity score. This is where a search engine paradigm makes its greatest contribution to our problem. Note first of all, that unlike exact matching based techniques (such as hash map look ups), a commercial search engine, facilitates approximate matching of n-grams [5], [16]. This helps overcome the lack of match when an identifier occurs across split fields in an HTTP flow document (cf. Fig 1 (c) and (d).)

Finally, even though it is reasonable to assume that two different mobile applications never produce exactly the same flow document, relative similarity is to be expected. Hence, in the search engine paradigm, short text matching is most likely to produce a fuzzy match, mapping a query to a subset of index documents. This inevitable consequence actually has some positive consequences for the MApId problem, as it helps us map flows to a family of apps; a desired side effect (cf. §VI-C).

[1]This is, of course, a simplification which ignores word ordering. But the point is made purely for illustrative purposes.

V. System Architecture and Operational Life-Cycle of AMPLES

We now present the system architecture and the operational life-cycle for AMPLES. In §V-A, we present the main constituent components of AMPLES. We then demonstrate the steps of the operational life-cycle — per mobile app index document creation and per HTTP flow query construction in §V-B and §V-C respectively.

A. System Architecture

Following is a functional description of the five principal components of AMPLES (for further details cf. [2]):

1. **Crawler and Downloader:** Periodically crawl the application market place to discover newly added mobile apps. Download application executable archives to maintain an up-to-date repository.
2. **Index Creator:** For each mobile app executable in the repository, extract-parse-tokenize-refine relevant textual information from the metadata and application source-code files. Create a per-app interim index document (cf. §V-B for details).
3. **Index Consolidator:** Periodically execute to identify, record and filter out *stop-words* from the per-app interim index documents to generate per-app index documents for the search engine index and a stop-word list for the query constructor. A term is deemed a stop-word if it occurs in the index documents of more than a fixed threshold of index documents. As the interim index created from the metadata and application source-code files, is bound to have a lot of noise in it (for e.g. multiple applications use the same advertisement/analytics service), the consolation step is critical to the success of our methodology. A search engine score is only meaningful if the index documents contain enough distinguishing terms in them.
4. **The Search Engine:** An Apache Lucene search engine that runs over a Solr server substrate. The search engine contains the index documents and a customizable scoring heuristic that is used to compute similarity scores between an incoming flow-query and the index documents.
5. **Query constructor:** Parse-tokenize-refine an observed HTTP flow at an intercept point; and converts it into a flow-query document. Remove all stop-words from the flow-query and run it against the search engine index. Essentially, the query constructor operates on the HTTP flow in the same way as the index creator operates on the meta-data and source code (cf. §V-C for details).

B. From a Mobile App Executable to an Index Document

A mobile application, downloaded from the marketplace, comes in the form of an archive file. These archive files are henceforth referred to as an *apk* for an Android app, and as an *ipa* for an iOS app. The *apk* archives, found in the Google-Play marketplace, have the same name as the unique package id for the Android app (e.g. `com.rovio.angrybirdsspace.premium`). On the other hand, the *ipa* archives, found on iTunes store, are named as a 9-digit unique app identifier (e.g. `501968250` for the same app). Each archive file, in turn, has an application executable and a text file that contains *metadata* related to the application. It is the content of these files that we use to construct the index document for an application. But first we need to define the structure of an index document.

Index Document: A per-app index document in our system is a structured XML file with the following fields [2]:

1. **APPID:** A unique identifier for the application (e.g. package name for Android apps and 9-digit identifier for iOS apps). This identifier field is used by the search engine as the equivalent of a primary key for databases. The value for this field is found in the archive names as well as in the metadata files.
2. **HST:** Domain host-names contained in all the URL strings extracted from the application executable. If more than one host-name is found, the field is multi-valued with comma used as a delimiter.
3. **URI:** The path/uniform resource identifier contained in all the URL strings extracted from the application executable. Multiple values are stored with comma as a delimiter.
4. **KEY-VALUE:** This field is populated using textual information extracted from both the metadata as well as the application executable. However, it is worth mentioning here that while metadata files contain well defined key-value pairs, those found in the query parameters (PAR) of a URL strings extracted from the application executable may not always do so. We provide more details of it in the subsequent paragraphs while describing the process of extraction and refinement of textual information from the executable archives.
5. **MISC:** A bag of words produced by combining *atomic* terms from all the other fields described above. Atomic terms are obtained from individual field values by splitting them along special characters (e.g. {/, ., ,, =, ...}). For instance, APPID:com.rovio.angrybirdsspace.premium yields four atomic terms {com, rovio, angrybirdsspace, premium} while HST: `csi.gstatic.com/csi` yields three atomic terms {`csi, gstatic, com`}.

Next, we show the details of how the fields of an index document are populated. As the specifics of the extraction process differ for the Android and iOS platforms, we deal with each separately.

From an Android apk: We decompile an *apk* using the publicly available *Android Asset Packaging Tool* (or *aapt*-tool) [1]. This yields a metadata file called *AndroidManifest.xml* and the application source code in a DEX/SMALI format.

Consider, for example, the popular Android app *Angry Birds Space* (Paid) by *Rovio Mobile Ltd*. The file `AndroidManifext.xml` for this app has a field called `<manifest>`, which contains within it the application version code (`versionCode="1510"`), version name

[2] We use the same tags as were defined in Table I.

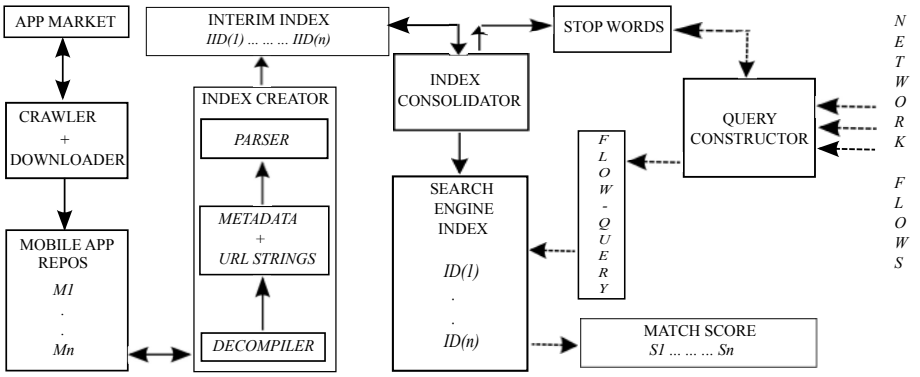


Fig. 3. AMPLES: System Architecture and operational life-cycle. The operational life cycle is shown from left to right for the search engine index creation (solid control flow lines); whereas the query construction process is shown from right to left (dashed control flow lines).

(`versionCode="1.5.1"`), and the unique identifier `package=com.rovio.angrybirdsspace.premium`. The free version of the same app has a unique app identifier `package=com.rovio.angrybirdsspace.ads`.
This unique identifier constitutes the *APPID* field in the index document. Additionally, the manifest file may also contain multiple `<meta-data>`) fields in it. These fields usually have an `android:name` and `android:value` pair in them (eg. `name="ADMOB_PUBLISHER_ID" value=x14d2b...a346"`). We treat all such fields as key-value pairs and place them in the KEY-VALUE field of the index document. It is known that some of these key-value pairs are used as identifiers when the application contacts an online host, or an advertisement/analytics service (eg. `Admob`, `Data-Flurry` etc.).

Next, from the decompiled source code, in the DEX/SMALI format, we extract all strings that resemble a URL. Essentially, we look for strings that contain within them the tags `http` or `www`. For example, `http://neptune.appads.com/ ... /Services/Client.svc/GetConfiguration?pub=`. Using a URL parser, we split each such URL string into its constituent fields (as discussed in Table I). We place the domain-host name (`neptune.appadds.com`)in the HST field of the index, the URI string (`/Services/Client.svc/GetConfiguration`) into the URI field and so on and so forth. Every substring of the URL that contains a key-value pair in it, identified by an = sign, (e.g. `pub=`) is tokenized and individual pairs are placed in the KEY-VALUE field as well.

Finally, the MISC field of the index is populated by gathering *atomic* terms from all the other fields and making a bag of words out of them. This concludes the index creation process from an Android *apk*.

From an iOS ipa: The application executables for iOS are named with the 9-digit unique app identifier. For example, the ipa for *Angry Birds* (Free) is named `409807569.ipa`. We use this 9-digit id to populate the *APPID* field in the index document. An *ipa* file, like an *apk*, also contains an application executable and a metadata file called an *infoPlist* (e.g. `AngryBirds.infoPlist.txt`). This metadata file is, in fact, an XML which contains a number of key-value pairs in it in the form of `<key>...</key>`, `<string>...</string>` combinations. A few of these are: {`CFBundleName=AngryBirdsClassicLight`, `CFBundleIdentifier=com.rovio.angrybirdsfree`, `CFBundleVersion=1.5.1 ...`}. All of these are placed in the KEY-VALUE field of the index document.

Alas, there is no analogue of the *aapt*-tool for the iOS platform. Hence, decompiling source code is not an option for ipa files. Instead, to extract the URL strings from the executable, we simply run the UNIX command `strings` and select all strings which have a URL like structure (i.e. contain `http` or `www` in them). We then parse, tokenize and arrange these strings into the index document in much the same way as was described for the Android platform.

Index field weight and Score function: To maximize the matching potential of our system, we use a supervised methodology for assigning weights to index fields. Taking a subset of applications in the training set, for which we have *ground truth* (cf. §VI-A), each index field is given a weight based on the fraction of apps for which it matches at least one flow. For details, please see the extended version of this paper [2].

C. From an HTTP Flow to a Flow-Query

As described in §II-B, an HTTP flow header is essentially a semi-structured text document. Using a DPI, the various constituents of an HTTP flow header can be parsed and tokenized to obtain all HTTP fields in the header. A flow-query document is then created in the same way as an index document is created with HST, URI and KEY-VALUES. However, there are two essential differences. First and foremost, a flow-query document does not have the APPID field in it. Secondly, the HTTP header might contain non-standard fields that are not known *a priori*. Hence, we take all the fields which have textual content, tokenize them into *atomic* terms and put them in the MISC field of the query document. Due to space constraints, we do not detail the process here.

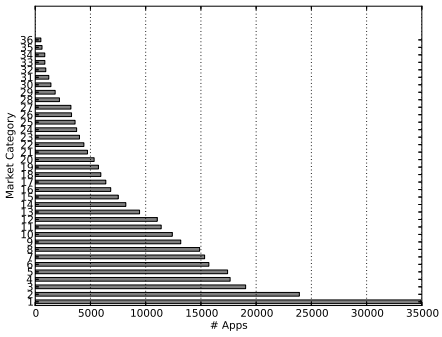


Fig. 4. Emulator generated test data: Mobile applications per market category [**Android:** 1-Entertainment, 2-Lifestyle, 3-Books and Reference, 4-Education, 5-Business, 6-Music and Audio, 7-Tools, 8-Personalization, 9-Travel and Local, 10-Games - Brain and Puzzle, 11-News and Magazines, 12-Sports, 13-Health and Fitness, 14-Games: Casual, 15-Social, 16-Games: Arcade and Action, 17-Media and Video, 18-Productivity, 19-Finance, 20-Shopping, 21-Brain and Puzzle, 22-Communication, 23-Photography, 24-Casual, 25-Arcade and Action, 26-Medical, 27-Transportation, 28-Games: Cards and Casino, 29-Comics, 30-Games: Sports Games, 31-Weather, 32-Games: Racing, 33-Libraries and Demo, 34-Cards and Casino, 35-Sports Games, 36-Racing].

VI. Experiments

In this section we present experimental results over a variety of datasets (for more detailed evaluation see [2]). In §VI-A, we describe our test dataset which contains over 400 K mobile applications and network traces generated by executing them in an emulator environment (roughly 13.5 million flows). We then present coverage results in §VI-B when this trace is run against a *full* index as well as several randomly selected *partial* indices. The analysis of these results leads us to a relaxation of the *MApID* problem: mapping of flows to a potential *family of apps* (§VI-C); for enhanced coverage. Next, we evaluate our system against a real network traffic trace §VI-D.

A. Emulator Generated Test Dataset

Test Data: Our test data comprises of application executables for 330 K *free* Android apps and 70 K *free* iOS apps. Though not exhaustive by any means, it spans all market categories (cf. Fig. 4) and respective operating system versions (**Android:** `Froyo, Gingerbread, Icecream-Sandwich and Jelly-beans` and **iOS:** `4.x, 5.x, 6.x`). Each of the 400 K applications in our dataset is executed in an emulator environment. For fairness, we execute an application thrice, each execution instance lasting a 5 minute period during which automated random clicks are arbitrarily performed on the UI. This helps trigger random events some of which are bound to produce network traffic. Additionally, given that our dataset is constituted of free applications, we expect background network traffic (e.g. ad-networks, analytics-services etc.) to be generated periodically, as well. In all, this emulation process generates 13 million HTTP flows for the 330 K Android applications and 477 K HTTP flows for the 70 K iOS applications. For each HTTP flow, we maintain a record of the unique application id which produced the flow. Therefore, for every flow in this test dataset, we have the *ground truth*.

Search Engine Indices: As described in §V-B, we create a per app index document for the 400 K applications. For convenience, our test system keeps two separate index sets: one for Android and another for iOS.

Test Queries: As described in §V-C, we create a flow-query for each of the 13.5 million flows in our test dataset. Once again, we maintain a record of the application id that generated the flow associated with the flow-query (the *ground truth*); and keep the Android and iOS queries separate from each other.

B. Evaluation: True Match vs. No-Match

Full Index Run and True Match: As a first cut, we load all the index documents in the test dataset into the search engine; and run all the flow-queries, one at a time, against it. We begin with the simplest and strictest possible dichotomy, that of a *true match* versus a *no-match*. We define a *true match* as a result in which the maximum match score is *uniquely* attained by the application that generated the flow-query. If a flow-query does not produce a true match in our system, we count it as a no-match.

Results: We obtain a true match for 5.3% million out of 13 million Android flows and 147 K out of 477 K iOS flows in our test dataset. Hence, the flow coverages (η_{FC}) for Android and iOS platforms, in this preliminary experiment, are at 40.76% and 30.81% respectively. Despite the relatively lower η_{FC}, we identify 216 K out of 330 K Android apps and 40.6 K out of 70 K iOS apps respectively, which translates to at 65.4% and 58% application coverages (η_{AC}) respectively. Upon relaxing the *uniqueness* constraint [3], we see a rise in η_{FC} to 66.8% and 64.6% for Android and iOS respectively, while η_{AC} rises to 82.7% and 76% respectively. This clearly shows that the exact matching criteria can potentially under represent the match potential. Hence, we propose fuzzy matching as an alternative to the exact matching criteria in §VI-C. But first, we present an analysis of the score distributions obtained during this experiment, for both the true match and no-match sets, to provide some interesting insights.

Analysis of Results: For each flow-query, we analyze the score obtained by the app that generated the flow-query, vis-a-vis other apps. Let S^* denote the normalized score obtained by the app that generated the flow query. Fig. 5 shows the ECDF plot for S^*. As expected, for the true match set the normalized scores are higher on an average as opposed to the no-match sets for both platforms. But more interestingly, for more than 45% of the flows in the Android no-match set (NM Android) and about 33% in the iOS no-match set, S^* has a value in the range of [0.2, 0.4].

[3]The app producing the flow is one of the apps that attain the highest match score, although not uniquely so.

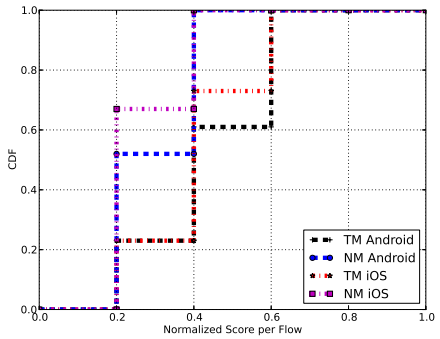


Fig. 5. Emulator generated test data: S^* per flow for Android and iOS.

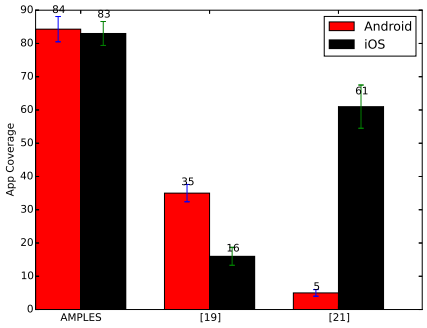


Fig. 6. App coverage compared with [19] and [21]. Error bars: False +ves.

This is not an insignificant score fraction by any means. In fact, in terms of absolute ranks, we observe that the app generating the flow attains a rank in the top-3 each time. Through manual inspection, we find that in most cases other apps — attaining a higher score than S^* — were either a different version of the same app (e.g. `aaos.xlite`, `aaos.xlite2012`, `aaos.xlite2013`), or different apps by the same developer (e.g. `br.com.pontomobi.ei`, `br.com.pontomobi.Rio20Vivo`). This observation automatically leads us to consider a relaxation of the true match objective to a *fuzzy match* as explained in the next subsection.

C. Fuzzy Match, Application Families, No-Result, False Positives and False Negatives

A *fuzzy match*, as opposed to a true match, returns a set of possible candidates as its final result. Recall, a search engine score is basically a probability distribution. If $S(ID_i, Q_j)$ be the score obtained by index document ID_i for flow-query Q_j, we can interpret the result as the following: Q_j was generated by the app M_i with the probability:

$$P = \frac{S(ID_i, Q_j)}{\sum_{k=1}^{n} S(ID_k, Q_j)} \quad (1)$$

Given this interpretation, we consider all applications, arranged in decreasing order by probability to be a fuzzy match for a query Q_j. But all contests must have a winner! Without a definitive result, we cannot possibly quantify the false positive and false negatives. Thus we finally come to the concept of *application families*. We organize all the applications in our test dataset (Android and iOS separately) into application families on the basis of two features:

1. **Developer ids:** Such as Zynga, Disney, Daniel K., John Doe etc., crawled from the application web pages in their respective market places.
2. **Package name:** For Android we use the application package names (e.g. `com.rovio.angrybirdsspace.premium`, `ae.sandcastles.wantedlist` etc.), while for iOS, we use the `CFBundleIdentifier` as the application ids are numerical; and have little lexical value.

We first separate the applications into clusters by the developer ids, and then within each such cluster, use substring similarity in package names to form sub-clusters. Each one of these sub-clusters is a family. Using this basic technique, we obtain roughly 113 K app families for the Android set (roughly one-third the total number of apps), and 46 K for iOS (roughly two-thirds) [4]. Next, we define the winner of the *fuzzy match* as follows: the app family that cumulatively obtains the highest fraction of the match score is considered the winner (or the result). Note that we do not need to alter the per-app index documents for this. We simply do a post-processing of the score vector to obtain the cumulative scores per app family.

Fuzzy Match Results: Using the fuzzy match criteria, as described above, we obtain a flow coverage of about 81% for the Android set and about 76% for the iOS set, with roughly 3.7% false positives. The respective application coverages rise to 278 K out of 330 K (84.3%) for Android and 58 K out of 70 K (83%) for iOS. Most importantly, we match over 3.5 million flows for Android and about 100 K flows for iOS, without any explicit application identifier (package name or 9-digit app id). Compared to other state-of-the-art techniques in literature ([19] for Android, and [21] for iOS), this is a significant improvement on both platforms (cf. Fig. 6). Note also the strikingly higher false positive rates ($\approx 12\%$) for the User-Agent based identification method [21], even in the case of the iOS platform.

False Positives, No Result and False Negatives: A result is considered a *false positive* if the family to which the app generating the flow belongs to, does not obtain the highest cumulative score. A result is considered no-result if the highest cumulative score attained for a query is less than a threshold (κ). This helps control the false positive rates. Through experimentation, we attain a false positive rate of 4% on an average for $\kappa = 0.10$. The definition of a false negative is slightly tricky in our set up. We call a result false-negative if a flow with an explicit identifier (e.g. the unique package name), is classified as a no-result. We observe less than 1% cases of this kind. Through inspection, we observe that this is

[4]The relatively lower reduction in the case of iOS is due to the fact that we have fewer apps which are not necessarily representative of the entire market.

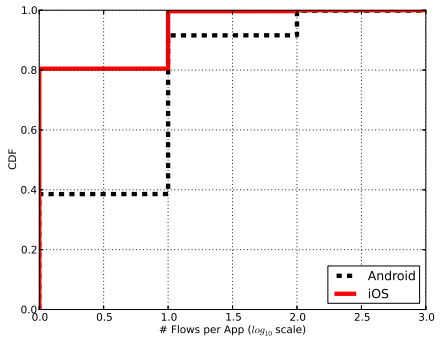


Fig. 7. Emulator generated test data: # Flows per app for Android and iOS.

only true when the app-identifier itself is a short string made up of generic sub-strings (e.g. `aa.a3`). We acknowledge that there is a wide ranging diversity in flow counts per mobile app in our dataset (cf. Fig. 7). About 40% Android applications produce 10 or fewer flows; while the same is true for about 80% iOS apps. This is a limitation of our dataset. Despite this, our methodology seems to perform well.

D. Real Network Trace

We now repeat the experiments from the previous section, with a 3 hour long real network traffic trace collected at an intercept point. Using tags from the set {iOS, iPhone, iPad, iPod, CFNetwork, Android, Andromo, Dalvik}, we isolate potential mobile application traffic; thus obtaining a million mobile flows from the 3 hour trace. We do not have *ground truth* for this dataset. Hence, we use an indirect method to create the ground truth. Using the list of unique application identifiers for over 400 K mobile apps (330 K Android apps + 70 K iOS apps), we select flows from this dataset where at least one of these identifiers is present in the flow header. In all, we obtain about 104 K flows from 1,055 applications. Through manual inspection of a few thousand flows, we were able to verify the correctness of our ground truth creation process. Next, we ran the 104 K flows through our search engine indices. We obtain a true match for over 98 K flows and identified every single application from amongst the 1055 apps. This in itself is not surprising as each flow has an explicit identifier in it. However, it gives us confidence that AMPLES would function well in the wild as well.

VII. Conclusion & Future Work

In this work, we presented **AMPLES**, a system that addresses the MApId problem with a flow coverage objective. Having cast the MApId problem in the information retrieval domain; we proposed a search-engine based solution that allows for approximate substring comparisons (n-grams) as well as fuzzy matches, thereby mapping a network flow to a family of applications. Our experimental results — 80% flow coverage for approx. 85% of applications in our test dataset with less than 4% false positives and nearly no false negatives — demonstrate the efficacy and feasibility of our methodology. This makes AMPLES particularly suitable to complement recently proposed signature / rule based solutions (e.g. [22], [23]). A preliminary study of some other mobile platforms (e.g. Nokia-Symbian, Windows Mobile and BlackBerry) gives us confidence that AMPLES can be extended to these platforms (cf. [2]). We propose these as future work.

VIII. Acknowledgement

Our sincerest thanks to Stanislav Miskovic, Yong Liao and Ramya Gadiayaram for their help with data collection and preliminary discussions; and to the anonymous reviewers whose suggestions helped improve the quality of this work.

References

[1] AAPT. http://developer.android.com/tools/building/index.html.
[2] AMPLES extended version. http://www-users.cs.umn.edu/~granjan/Reports/TECHREP_AMPLES_Extended.pdf.
[3] HTTP-vs-HTTPS. http://resources.infosecinstitute.com/data-traffic-network-security/.
[4] HTTPFields. http://en.wikipedia.org/wiki/List_of_HTTP_header_fields.
[5] The lucene search engine. http://lucene.apache.org.
[6] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. Networkprofiler: Towards automatic fingerprinting of android apps. In *Proc. of IEEE Infocom*, 2013.
[7] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *J. of the American Society for Information Science*, 41(6):391–407, 1990.
[8] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In *Proc. ACM IMC*, 2010.
[9] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. Acas: Automated construction of application signatures. In *Proc. ACM SIGCOMM workshop on Mining Network Data*, 2005.
[10] T. Hofmann. Probabilistic latent semantic indexing. In *Proc. ACM SIGIR*, 1999.
[11] M. Korczynski and A. Duda. Markov chain fingerprinting to classify encrypted traffic. In *Proc. of IEEE Infocom*, 2014.
[12] G. Maier, F. Schneider, and A. Feldmann. A first look at mobile handheld device traffic. In *Proc. of PAM*. Springer-Verlag, 2010.
[13] D. Metzler, S. Dumais, and C. Meek. Similarity measures for short segments of texts. In *Proc. of ECIR '07*, 2007.
[14] T. Nguyen and G. Armitage. A survey of techniques for Internet traffic classification using machine learning. *IEEE Communications Surveys and Tutorials*, 10(4):56–76, 2008.
[15] A. Patro, S. Rayanchu, M. Griepentrog, Y. Ma, and S. Banerjee. Capturing mobile experience in the wild: A tale of two apps. In *Proc. ACM CoNEXT*, 2013.
[16] M. Sahami and T. D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *Proc. of WWW*, 2006.
[17] 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 *Proc. of EWSDN*, 2014.
[18] 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 netwk. func. virtualization. In *Proc. of ACM MobiArch*, 2014.
[19] A. Tongaonkar, S. Dai, A. Nucci, and D. Song. Understanding mobile app usage patterns using in-app advertisements. In *Proc. of PAM*. Springer-Verlag, 2013.
[20] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: Multi-layer profiling of android apps. In *Proc. of ACM MobiCom*, 2012.
[21] Q. Xu, J. Erman, A. Gerber, Z. M. Mao, J. Pang, and S. Venkataraman. Identifying diverse usage behaviors of smartphone apps. In *Proc. of ACM IMC*, 2011.
[22] 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 *Proc. of IEEE Infocom*, 2015.
[23] H. Yao, G. Ranjan, A. Tongaonkar, Y. Liao, and Z. M. Mao. Samples: Self Adaptive Mining of Persistent LExical Snippets for classifying mobile application traffic. In *Proc. of ACM MobiCom*, 2015.