CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **28**:3927–3941
Published online 8 December 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3703

SPECIAL ISSUE PAPER

Automatically identifying apps in mobile traffic

Jianhua Sun*,†, Lingjun She, Hao Chen, Wenyong Zhong, Cheng Chang, Zhiwen Chen, Wentao Li and Shuna Yao

College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China

SUMMARY

With the rapid development of smartphones in recent years, we have witnessed an exponential growth of the number of mobile apps. Considering the security and management issues, network operators need to have a clear visibility into the apps running in the network. To this end, this paper presents a novel approach to generating the fingerprints for mobile apps from network traffic. The fingerprints that characterize the unique behaviors of specific mobile apps can be used to identify mobile apps from the real network traffic. In order to handle the large volume of traffic efficiently, we use non-negative matrix factorization (NMF) to perform traffic analysis to cluster similar network traffic into groups. Then, access patterns of individual apps that are extracted from each group can be used as fingerprints distinguishing apps from others uniquely. The experimental evaluations show that the proposed approach can identify the mobile apps from random and mixed network traffic with high precision. Copyright © 2015 John Wiley & Sons, Ltd.

Received 4 April 2015; Revised 21 July 2015; Accepted 5 September 2015

KEY WORDS: mobile traffic; network behavior; traffic classification; non-negative matrix factorization; clustering

1. INTRODUCTION

With the increasing dependence on smartphones to fulfill a wide range of tasks, more and more mobile apps have been developed to facilitate our daily life. As a result, the Internet traffic is increasingly composed of the traffic incurred by mobile apps. It was reported that the mobile apps made up 47% of Internet traffic. A survey carried out by one large cellular operator [1] shows a growth of 8000% of cellular data traffic over the past 4 years, and it is expected to rise to 10.8 exabytes per month by 2016. This shift of Internet traffic poses a great challenge on network operators to recognize mobile apps running in the network, which is important for traffic engineering and billing, network planning, provisioning, and network security.

In order to have a clear understanding of mobile apps in cellular networks, it is necessary to classify the apps' traffic and perform protocol identification. At present, many methods of traffic classification have been proposed, and most of them are based on the following three aspects [2]:

- packet-level traffic classification, which mainly focuses on the features such as the distribution of packet size, the distribution of time interval of packet arrival, and so on.
- flow-level traffic classification, which analyzes the features of flows and the arrival process for a TCP connection or a UDP flow. Flow usually refers to a five-tuple, consisting of source Internet protocol (IP), destination IP, source port, destination port, and application protocol.

*Correspondence to: Jianhua Sun, College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China.
†E-mail: jhsun@aimlab.org

- stream-level traffic classification, which considers the host pair and the traffic between them, referring to a three-tuple composed of source IP, destination IP, and application protocol.

To some extent, these methods can classify the network traffic, but they are not effective enough to classify the traffic of mobile apps and identify the specific apps from real network traffic. Several approaches have been proposed to address the issue of understanding the behaviors of mobile apps by analyzing network traffic. For example, the research conducted by Xu *et al.* [3] uses the user-agent field in the Hypertext Transfer Protocol (HTTP) header to identify apps, which only works for the certain mobile platform that has an app identifier in the user-agent field such as iOS, but not for Android apps whose user-agent field does not contain the corresponding identifier. Other work uses the host field in the HTTP header. However, it is not sufficient because the same host often provides services for multiple apps. For instance, the same host *www.sina.com* is typically used for different apps, such as *weibo* and *sina news*.

In this paper, we mainly focus on Android apps. Results from existing studies show that a vast majority of Android apps use the HTTP/HTTPS protocol, so we can perform traffic classification by only considering HTTP/HTTPS traffic. The method proposed in [4] classifies the traffic into third-party and non-third-party traffic. In addition, it needs to collect apps' identifications in the source code or in the manifest files that often is not always accessible, so it is not simple enough to achieve the goal of app identification. Therefore, in this work, we aim to develop simple and effective techniques such as non-negative matrix factorization and multiple similar strings clustering to extract the fingerprints from app's network traces obtained by manipulating the apps on a real smartphone. In our system, we do not need to rely on the availability of the source code.

In summary, the main contributions of our work are the following:

- We present the design and implementation of a system that can automatically generate fingerprints for mobile apps. Compared with previous schemes, our system is more effective.
- We present an algorithm to extract the fingerprint for each cluster of traffic, and through a large number of experiments, we can obtain the best parameters in our algorithm to extract the fingerprints.
- We perform extensive evaluation to demonstrate that our system can identify the mobile apps from network traffic with high precision.

The rest of the paper is organized as follows. Section 2 describes the primary principle of the method in our system. Section 3 gives a detailed explanation of system design. The evaluation is presented in Section 4. The related work is discussed in Section 5 and Section 6 concludes.

2. MOTIVATION

The key observation in this work is that every mobile app can exhibit specific network behavior that can be used to generate its unique feature to differentiate the diverse mobile apps. We call the unique features of mobile apps as fingerprints. It is analogically similar to human DNAs. Because we only consider the apps that use HTTP as the communication channel, the network behavior can be characterized in terms of different HTTP requests. The fingerprints are the abstract of these HTTP flows that may be different in terms of the HTTP method, hosts connected, uniform resource locator (URL) paths or query strings, and so on. Concretely speaking, only the specific tokens in network messages and the hosts are essential for identifying the apps.

On the other hand, with the technology of machine learning becoming more and more mature, many traffic identifying methods based on machine learning have been proposed. The course of machine learning often include the building of classification model to be used to classify traffic. In our paper, we can utilize it to extract the fingerprints in our system.

In order to more clearly explain the challenges of automatically extracting application-level fingerprints of mobile apps, we first introduce the mobile app's structure using Android as the target platform. Android apps are usually written in Java with some additional native code. A typical Android app consists of separate screens that are called activities, each of which typically contains a set of GUI elements such as pop-ups, text boxes, text view objects, check boxes, and so on. In order to interact with an app, users navigate different activities using the said GUI elements, which incurs transitions between activities. Therefore, activity transitions and coverage are fundamental, and activities are the main interfaces presented to an end user. Activities can serve for different purposes. For example, in a typical news app, an activity home screen shows the list of current news; selecting the headline of a news will trigger the transition to another activity that displays the full news item. Activities are usually invoked from within the app, although some activities can be invoked from outside the app if the host app allows [5].

For example, *weibo* is one of the most popular app in China. It is a social app for mobile user to share, spread, and obtain new things. The typical user interactions with the app are as follows. A user first open the app; then the user can click on an option to choose which among the desired function; finally, a table of all kinds of activities is fetched from the Internet and presented according to the choices, and the user can proceed to play with other functions they like. Figure 1 shows how the activity transition graph is created as a result of a user interacting with the *weibo* app.

In the aforementioned process, both the fetching of activities and the playing of functions involves network access. In this paper, we regard these as different network behaviors. As we know, every app may have a large number of network behaviors, and it is important to collect network traces for fingerprint extraction to reveal the network behaviors as many as possible. Consider an extreme case when the *weibo* app is executed only once, and many activities are not reached in a single execution. The fingerprints generated from the network trace will be too specific to be useful for app identification. So it is important to run the app many times to ensure that all of the network behaviors can be covered, in order to collect a significant part of the app's network behavior to produce the fingerprints that can be used to successfully identify the apps.

The key idea behind our fingerprint extraction algorithm is to identify the invariant parts of the message flows inherent to an app. If the invariants is unique to the app, they can be used to distinguish different apps in network traffic. Existing researches show that invariant strings in the URL and the host field can be used to uniquely identify mobile apps. Figure 2 shows exemplar HTTP flows we extract from the network trace of *weibo* and omit the fields that are not useful for identification.

We hope that we can extract the fingerprints from these HTTP flows as shown in Table I.

Of course, this simple example is only used to illustrate the general idea of fingerprints. Extracting fingerprints from real mobile apps is relatively more complex, and the detailed discussion is presented in the following sections.

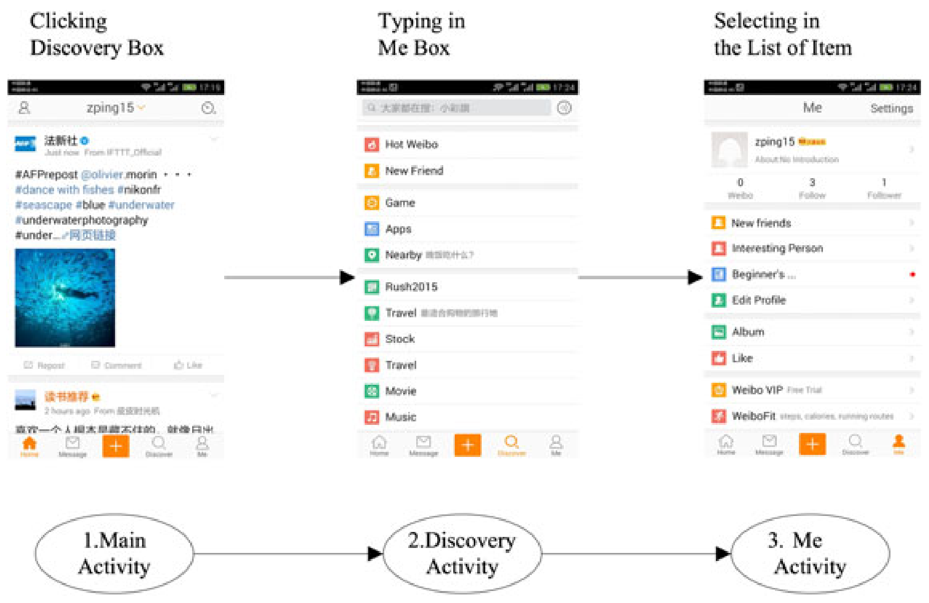

Figure 1. An example activity transition scenario from weibo app.

```
GET /webp360/                                  GET /woriginal/
63b4ac8djw1ek0n7ahe1zj21400u0wkx.jpg           bfc243a3gw1ek0nbfoeojg20640647jd.gif
HTTP/1.1 Host:ww4.sinaimg.cn                   HTTP/1.1 Host:ww4.sinaimg.cn

GET /webp360/                                  GET /woriginal/
5033b6dbjw1ek0ndwcjuyj20l00vkgr7.jpg           bfc243a3gw1ek0nblk7dhj20vv16\n54.jpg
HTTP/1.1 Host:ww2.sinaimg.cn                   HTTP/1.1 Host:ww2.sinaimg.cn

GET /webp360/                                  GET /woriginal/
61e7f4aagw1ek0nnlez5uj208m0df757.jpg           61e7f4aagw1ek0nnsd5m5j20bh0dwwfk.jpg
HTTP/1.1 Host:ww4.sinaimg.cn                   HTTP/1.1 Host:ww4.sinaimg.cn
```

Figure 2. The HTTP flows.

Table I. Fingerprints example.

GET	webp360	Host:ww2.sinaimg.cn
GET	webp360	Host:ww4.sinaimg.cn
GET	woriginal	Host:ww2.sinaimg.cn
GET	woriginal	Host:ww4.sinaimg.cn

3. SYSTEM DESIGN AND IMPLEMENTATION

Our fingerprints generating system consists of four main components: traffic collector, preprocessor, clustering module, and fingerprint generator. Figure 3 shows the main building blocks of our system and their main functionalities are listed as follows.

(1) Collecting the traffic of a specific mobile app. We can use wireshark to monitor the wireless local area network to collect the traffic by running the target app on the smartphone.
(2) Preprocessing the collected traffic, which include parsing the HTTP flows, filtering the unnecessary parts, and embedding the messages into a vector space.
(3) Clustering the HTTP flows (i.e., the messages) based on the similarity measurement between messages. Through clustering, all of the messages will be labeled so that they can be used to extract the fingerprints in later steps.
(4) Extracting the fingerprints using the methods detailed in later sections. The fingerprints represent the invariance of network communication patterns that is exploitable to identify apps from the network trace.

3.1. Traffic collector

Traffic Collector mainly include two parts:

Running mobile apps. As introduced in Section 2, in order to obtain the fingerprints of the target app, we need to run the app for multiple times to explore the network activities as complete as possible. There are several existing methods to run mobile apps automatically, such as Automatic Android App Explorer (A3E), a tool proposed in [5], to systematically explore real-world, popular Android apps running on actual phones to collect network traffic. But these methods are designed to explore some stand-alone activities rather than the activities related to network behaviors. But our goal is to explore the activities related to network behaviors, so we choose to run the mobile apps manually to collect network traffic.

Collecting traffic. For collecting raw network traffic, we use wireshark [6], which is one of the most popular network protocol analyzer and the de facto tool for packet-level traffic collection. We use it to monitor the wireless local area network in our experiment laptop. The target app runs on the smartphone that connects to the laptop using WIFI.

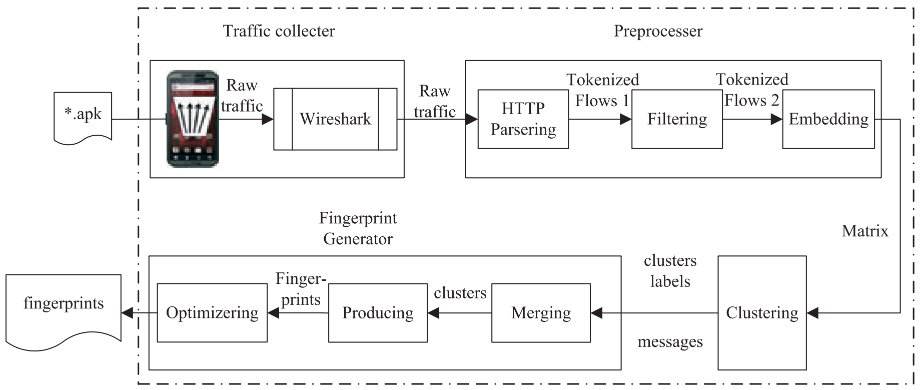

Figure 3. Overview of fingerprints extractor.

3.2. *Preprocessor*

In this part, we need to transform the raw packet traffic obtained to HTTP flows before performing other operations, because our approach is based on HTTP flows as introduced in Section 2. The preprocessor consists of three steps. First, we parse the raw packets into HTTP session messages. Then, we filter out the unnecessary information in the HTTP messages that is not essential to produce the fingerprints. Third, the HTTP messages are transformed and embedded into a vector space to facilitate the clustering algorithm.

3.2.1. HTTP parsing. Due to the fact that the majority of mobile apps use HTTP as the main communication protocol, we first transform the raw traffic collected to HTTP messages. There already exist many tools that can achieve this goal. In this work, we choose justiniffer [7], which is a network protocol analyzer that provides a wide set of functions in traffic analysis, such as capturing network traffic, emulating Apache web server logs in customizable ways, tracking response times, and extracting all 'intercepted' files from the HTTP traffic. With justiniffer, we can transform the raw packet traffic into HTTP messages, which contain the necessary information in the HTTP header to produce fingerprints. An HTTP message is an atomic exchange of a byte sequence between the client and server. A message is a string of bytes contained in network communication and composes of five main components including request-URI, HTTP-version, accept, host, and user-agent. For example, an HTTP message typically looks like this:

```
GET /webp360/7deb0325jw1emwe2f9d47j20c807m0td.jpg HTTP/1.1 Host:
ww2.sinaimg.cn Accept:- User-Agent:HM NOTE 1W_4.2.2_weibo_4.6.2_
android
```

3.2.2. Filtering. After tokenizing each message, the HTTP request can be broken into various parts including method (m), page (p), and query (q). Page can be further split into a number of page-components (pcs) and filename (fn). Query can be divided into key-value pairs. Through extensive experiments, we observed that the filename in certain fields of the message such as the request-URL, HTTP-version, accept, and user-agent are not useful to identify apps. So we can delete these parts in the message to lower the complexity in the later step of clustering. After filtering, the HTTP messages only include the method, request-URI, and host field. On the other hand, abnormal messages such as the HTTP responses indicating errors can also be filtered out because these messages do not convey any useful information. The following is the example of a filtered HTTP message:

```
GET webp360 Host:ww2.sinaimg.cn
```

3.2.3. Embedding. To characterize the content of a message x, we can model it as a sequence of bytes, that is, $x \in B^*$, with $B = \{0, \ldots, 255\}$. At the same time, we need to extract an alphabet W of relevant strings from a set of messages in order to embed the message into a vector space, which is a necessary step and provides a feasible model for the clustering algorithm.

Thus, what we should do next is to transform the HTTP messages into vector representation that is used as the input for the traffic clustering. To this end, we use Sally [8], a tool for embedding strings in vector space that allows for applying a wide range of learning methods to string data, to perform this task. After the message transformation, to infer common structures from a pool of messages, we need a similarity measure that is capable of analyzing discriminative features. We tokenize the byte sequence via pre-defined separators S and define the tokens as

$$W = \{\{0, \ldots, 255\} \setminus S\}^*.$$

Then, we define an embedding function $\phi : B^\star \longmapsto R^{|W|}$ that maps the byte sequence to a feature vector, which records the occurrences of all possible words W according to the separators, that is, $\phi(x) := (\phi_\omega(x))_{\omega \epsilon W}$ [9]. For example, if we only consider the set of separators $S = \{\sqcup\}$, we will get the following embeddings:

$$\phi(\text{"Hello World, Thanks"}) = (0, ..., 1^{Hello}, 1^{\,World,}, 1^{Thanks} ..., 0)^T \epsilon R^{|W|}.$$

For clustering the messages, the analysis has to focus on discriminative features in order to reduce the potential negative effect of high-dimension vector space on computational complexity. Volatile features, including randomly generated nonces or cookies and constant tokens that frequently occur in a message, always lead to an unnecessary bloated vector space. So constant and highly volatile components that do not augment semantic of features should be filtered out from the feature space. More precisely, we employ a statistical *t*-test for identifying non-constant and non-volatile features by testing whether their frequency is significantly different from 0 and 1 (see [10] for more details). Given these embeddings and the reduced feature space, we are now able to map these messages to the vector space. Figure 4 illustrates the process of embedding.

3.3. Clustering

After mapping messages into the vector space reflecting characteristics captured by the alphabet W, we can perform clustering on the vectorized messages. Because the messages share several substrings that are close to each other in similarity and network messages with different content exhibit larger geometric distances, we can use the concept of matrix factorization to cluster these messages. Given a set of messages $P = \{p_1, \ldots, p_N\}$, we first define a data matrix X containing the vectors of P as columns:

$$X := [\phi(m_1), \ldots, \phi(m_N)] \in R^{f \times N}.$$

To seek a solution for X that retains most information but with fewer base directions, we break the matrix X into two matrices $U \in R^{f \times L}$ and $V \in R^{L \times N}$ such that $L << f$ and

$$X \approx UV = \overbrace{[u_1, \ldots u_L]}^{basis} \underbrace{[v_1, \ldots v_N]}_{coordinates}$$

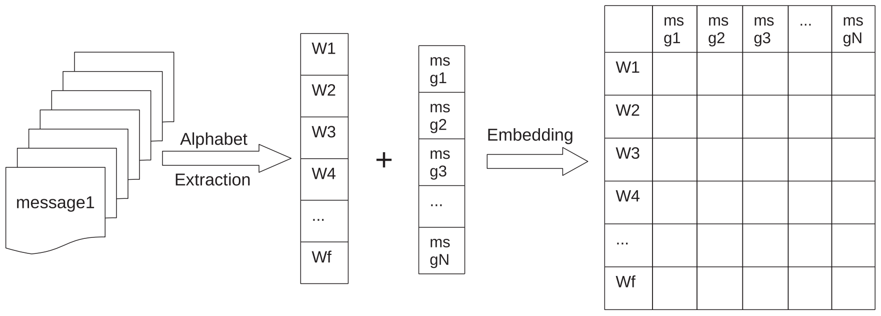

Figure 4. Embedding process.

The columns $u_1, \ldots u_L \in R^f$ of U form a new basis (i.e., denoting all kinds of message), where the dimensions of each base direction u_i are associated with the alphabet A. In this paper, the U and V are strictly positive matrices, so the matrix factorization method we use is called non-negative matrix factorization (NMF).

In NMF, the matrix U and V only contains non-negative entries. Non-negative entries in the basic vectors are a more natural representation for sequential data, as each string contributes positively to the basis representation. For a chosen dimensionality L, the factorization is defined in terms of the minimization criterion:

$$(U, V) = \operatorname{argmin} \| X - UV \| \quad s.t. \quad U_{ij} \geqslant 0,\ V_{jn} \geqslant 0$$

Because of the positivity constraint, the matrix U can be interpreted as a new basis (the parts of a message), while the matrix V contains the coordinates in this newly spanned space (the weights of the different parts), so these coordinates can be used to ultimately assign a message to a cluster by finding the position with the maximal weight.

To calculate U and V, suppose that $X' = UV$, we use the gradient descent method to solve the formula. With this method, we first initialize U and V with certain values, and then calculate the sum of the squares of the difference between UV' arithmetic product with true value in X in the corresponding position. At last, we can iteratively reduce the error until convergence. Here, we use squared error to calculate, and the formula is as follows:

$$e_{ij}^2 = (x_{ij} - x'_{ij})^2 = \left(x_{ij} - \sum_{L=1}^{L} u_{iL} v_{Lj} \right)^2 \tag{1}$$

In order to minimize the error, we alter the values of U_{ij} and V_{ij} as the direction of gradient descent. So we can compute the partial derivatives for Equation (1), and the outcomes are

$$\frac{\partial}{\partial u_{iL}} e_{ij}^2 = -2 \left(x_{ij} - x'_{ij} \right) (v_{Lj}) = -2 e_{ij} v_{Lj}$$

$$\frac{\partial}{\partial v_{Lj}} e_{ij}^2 = -2 \left(x_{ij} - x'_{ij} \right) (u_{iL}) = -2 e_{ij} u_{iL}$$

After calculating the gradient, U_{iL} and V_{Lj} can be further updated with

$$u'_{iL} = u_{iL} + \alpha \frac{\partial}{\partial u_{iL}} e_{ij}^2 = u_{iL} + 2\alpha e_{ij} v_{Lj} \tag{2}$$

$$v'_{Lj} = v_{Lj} + \alpha \frac{\partial}{\partial u_{Lj}} e_{ij}^2 = v_{Lj} + 2\alpha e_{ij} u_{iL} \tag{3}$$

The parameter α in the aforementioned formula is the gradient descent constant, and it is usually set to a small value such as 0.0002. With the updating rules, we can reduce the error step by step until convergence.

The algorithm described previously is only a simple implementation, and it will become very complex in reality. So we normalize it to prevent over-fitting by joining the parameter β to change the error formula as follows:

$$e_{ij}^2 = \left(x_{ij} - \sum_{L=1}^{L} u_{iL} v_{Lj} \right)^2 + \frac{\beta}{2} \sum_{L=1}^{L} (\| U \|^2 + \| V \|^2)$$

The parameter β is used to control the proportion of u and v to avoid oversize in matrix. In reality, we often set β to the value between 0 0.02. So Equations (2) and (3) are changed to

$$u'_{iL} = u_{iL} + \alpha \frac{\partial}{\partial u_{iL}} e_{ij}^2 = u_{iL} + \alpha(2e_{ij}v_{Lj} - \beta u_{iL})$$

$$v'_{Lj} = v_{Lj} + \alpha \frac{\partial}{\partial u_{Lj}} e_{ij}^2 = v_{Lj} + \alpha(2e_{ij}u_{iL} - \beta v_{Lj})$$

After X is decomposed into U and V, in the matrix V, we can assign labels for N messages based on the weight. For example, the second message's maximum weight is the third line. Then we can assign the label 'three' to the second message.

The following example of *NMF* shows the clustering process. To keep things simple, we choose four HTTP messages and map them to a matrix X as shown in the succeeding discussions using the method introduced in Section 3.2.3. Then we decompose the matrix X into two matrices U and V with the NMF algorithm.

In matrix V, we search the position of the maximum weight of every column and cluster them based on the position. For instance, in this example, we can observe that $n1$'s maximum weight is $l2$, $n2$'s maximum weight is $l1$, $n3$'s maximum weight is $l2$, and $n4$'s maximum weight is $l1$. So at last, we can assign the label $l2$ to $n1$, assign the label $l1$ to $n2$, assign the label $l2$ to $n1$, and assign the label $l1$ to $n2$, that is, we cluster the messages.

3.4. Fingerprints generator

After clustering the messages, we can obtain the cluster label for each message. Then we can assign the messages in each cluster with the cluster label and extract the fingerprints for each cluster. To this end, we perform three operations including merging, producing and optimizing as detailed below.

3.4.1. Merging. Unlike message clustering that only obtains the cluster label of each message, here, we focus on assigning messages with the same label to a class. The merging operation is shown in Figure 5.

For instance, in the example of *NMF* as shown in Figure 6, we can merge $n1$ and $n3$ with the same label $l2$ into a cluster, and merge $n2$ and $n4$ with the same label $l1$ into another cluster. Thus, all similar messages can be merged into a cluster that can be used generate the fingerprints.

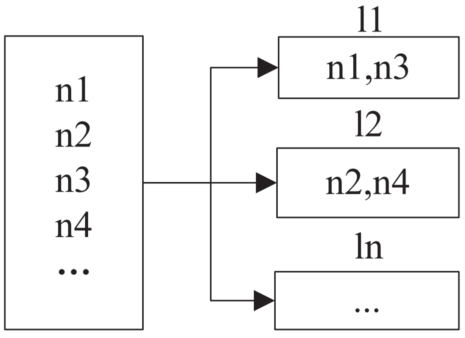

Figure 5. Merging process.

$$X = \begin{array}{c|cccc} & \mathbf{n1} & \mathbf{n2} & \mathbf{n3} & \mathbf{n4} \\ \hline \mathbf{f1} & 5 & 3 & 0 & 1 \\ \mathbf{f2} & 4 & 0 & 0 & 1 \\ \mathbf{f3} & 1 & 1 & 0 & 5 \\ \mathbf{f4} & 1 & 0 & 0 & 4 \\ \mathbf{f5} & 0 & 1 & 5 & 4 \end{array}, UV = \begin{array}{c|cc} & \mathbf{l1} & \mathbf{l2} \\ \hline \mathbf{f1} & 0.95 & 0.59 \\ \mathbf{f2} & 0.38 & 0.67 \\ \mathbf{f3} & 0.57 & 0.43 \\ \mathbf{f4} & 0.02 & 0.42 \\ \mathbf{f5} & 0.96 & 0.41 \end{array} \begin{array}{c|cccc} & \mathbf{n1} & \mathbf{n2} & \mathbf{n3} & \mathbf{n4} \\ \hline \mathbf{l1} & 0.07 & 0.63 & 0.45 & 0.67 \\ \mathbf{l2} & 0.66 & 0.45 & 0.86 & 0.29 \end{array}$$

Figure 6. The example of *NMF*.

3.4.2. Producing. After obtaining each cluster of messages, we can now produce the fingerprints. For each cluster of messages, we first calculate the length and discard those messages whose length is less than two. For the remaining messages, we classify them based on the length of every message and cluster messages with the same length to a class.

For messages with the same length, we count the number of the same message in a cluster; If the number is greater than a constant number that is chosen based on how many times the app was tested, we only keep one copy of the message. Otherwise, we discard the messages that may sneak in because of impure traffic during collection such as advertisement. The producing operation is shown in Figure 7.

For example, the *weibo* app is run three times in the experiment and there are several pieces of messages with the same length in a cluster extracted from the collected raw traffic, as shown in Table II.

As introduced previously, we first calculate the length of each message, and all the messages are retained because all are longer than two. Then, we observe that the number of the message `GET wap720 Host:ww3.sinaimg.cn` is four, which is bigger than three, and the number of the message `GET web360 Host:www.baidu.cn` is two, which is less than three. So the former is reserved and the latter is discarded. Finally, we have the fingerprint from these messages in a cluster:

```
GET wap720 Host:ww3.sinaimg.cn
```

3.4.3. Optimizing. The amount of fingerprints produced is typically huge, and many of them are similar and could be merged further. Therefore, we propose an optimization to address this issue. The main idea of the optimization is a piecewise comparison of these fingerprints and combining similar fingerprints that are measured with the longest common strings (LCSs). We call the method of combining fingerprints as multiple similar strings clustering (MSSC), which is shown in Figure 8.

Longest common strings is used to measure the similarity between two fingerprints. It is different from the LCS (traditional longest common substring). The traditional longest common substring is composed of a series of characters, but the longest common substring in this paper consists of a set of substrings, which is called the longest common substrings. For example, given two strings: `Hello world I am coming` and `Hello china I am going`, the traditional longest common

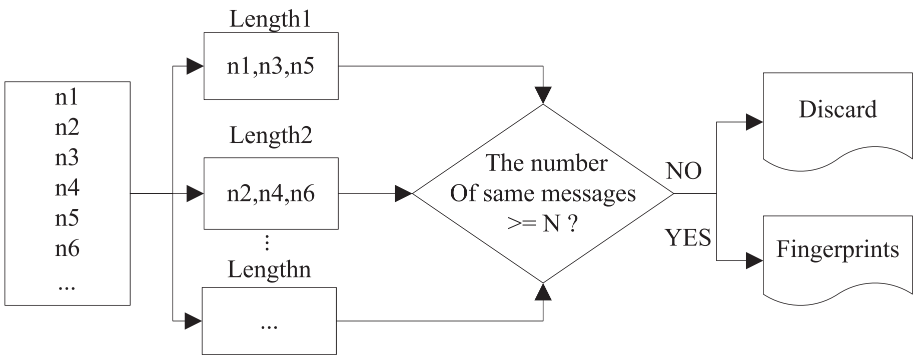

Figure 7. Producing fingerprints.

Table II. Fingerprints example in a cluster.

GET	wap720	Host: ww3.sinaimg.cn
GET	wap720	Host: ww3.sinaimg.cn
GET	wap720	Host: ww3.sinaimg.cn
GET	wap720	Host: ww3.sinaimg.cn
GET	web360	Host: www.baidu.cn
GET	web360	Host: www.baidu.cn

substring between them is `Hello I am oing`, but the longest common substrings between them in this paper is `Hello I am`.

The typical algorithm of solving traditional longest common substring is based on the dynamic programming to find the length of the LCS for all substring of the candidates. It mainly uses the following optimal substructure property to implement the algorithm.

- if$X[m] = Y[n]$, then $LCS(X_m, Y_n) = LCS(X_{m-1}, Y_{n-1}) + X[m]$
- if$X[m]! = Y[n]$, then $LCS(X_m, Y_n) = \text{Max}(LCS(X_{m-1}, Y_n), LCS(X_m, Y_{n-1}))$

The X and Y are two strings need to be compared, m and n are the length of the two strings, and the LCS indicate the longest common strings.

Our LCSs algorithm is similar to the traditional common string algorithm. As introduced earlier, the traditional longest common substring is composed of a series of characters, and the longest common substrings in this paper is consists of a set of substrings. So in our longest common substring algorithm, we only add one step that is to split our target string like `Hello world I am coming` by some predefined symbols such as spaces into multiple substrings, which are then regarded as different characters in the traditional longest common substring. The remaining operations are the same with the traditional longest common substring algorithm. Next, we describe the MSSC algorithm that is used to combine similar messages obtained previously. The MSSC algorithm is as follows:

The main purpose of MSSC is to cluster similar strings into a single string. We assume there are n strings in the following discussion of the MSSC algorithm. In order to evaluate the similarity between two strings, we introduce a parameter α to indicate the similarity. In our MSSC algorithm, we calculate the length of LCSs for each pair of strings iteratively. If the LCSs is greater than α multiplied by the minimum length between the two strings, we discard this pair of strings and keep the LCSs. Otherwise, we keep the two strings and discard the LCSs. It is described in detail in Algorithm 1.

The following example is used to help understand the algorithm. Suppose there are three fingerprints as shown in Table III.

We first calculate the LCSs of the first and the second, and find that the LCSs satisfy the similarity measurement. So we delete the two messages and continue comparing the LCSs with the third one. Then, we compare the LCSs with the third one, and find that the LCSs of them also satisfy the

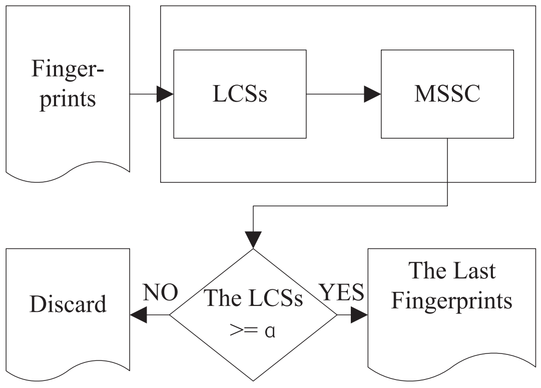

Figure 8. Optimizing fingerprints.

Table III. Fingerprints.

GET	wap360	Host: ww3.sinaimg.cn
GET	thumb180	Host: ww3.sinaimg.cn
GET	thumb150	Host: ww3.sinaimg.cn

Algorithm 1 MSSC

```
Require:
    The array of the strings s[n];
Ensure:
    The array of the string s[] that have been processed;
1:  for i ← 0 $to\ n$ do
2:    for j ← $i + 1$ $to\ n$ do
3:      $lcs$ ←LCSs(s[i], s[j]) //the longest common strings
4:      if len(lcs)> $\alpha$ * Min (len(s[i]), len(s[j])) then
5:        s[i] ← $lcs$
6:        delete s[j]
7:      else
8:        continue
9:      end if
10:   end for
11: end for
```

similarity measurement. Thus, we delete the first LCSs and the third message. At last, we will obtain the fingerprint from this example like this:

```
GET Host:ww3.sinaimg.cn
```

4. EXPERIMENTAL EVALUATION

4.1. Experimental setup

The smartphone used for the experiments was HM NOTE 1W running Android version 4.2.2 and Linux kernel version 3.4.39. The phone have Magny-Cours CPUs running at 1.7 GHz. We controlled the experiments from a desktop PC running Linux Mint 16. The smartphone is connected to the WIFI of this desktop that runs the fingerprints extracting system. The mobile apps we chose are *weibo*, *weixin*, and *zedge*, which are very popular apps at present. Among them, the *weibo* and *weixin* are very popular social apps. The *zedge* is one of the most popular apps in the Android market (more than 10 million downloads). *Zedge* is a simple previewer and downloader for mobile phone wallpapers, ringtones, and notification sounds.

The experiment evaluation is divided into two parts: the first is the extraction of fingerprints for each app and the second is the evaluation of the accuracy of the fingerprints.

4.1.1. Extraction of fingerprints. To ensure the precision of the fingerprints we extracted, we should collect enough raw traffic to guarantee covering all of the network behaviors of each app. So we run these apps for multiple times in our experiment to extract their fingerprints using the method discussed previously. For example, we run the *weibo* app three times on the smartphone without any other apps concurrently executing on the phone; based on the approaches introduced in this paper, we can produce the final fingerprints of the testing apps.

4.1.2. Evaluation of fingerprints. To evaluate the system, we conducted four experiments. The mixture traffic is needed, and it should be collected separately. So we run these apps including *weibo*, *weixin*, *zedge*, and others on the smartphone concurrently. To make the collected traffic resembling the traffic in real network, these apps need to be run randomly by hand. The experiments we carried out are as follows:

The first experiment. The first experiment we conducted is that we use the fingerprints of the three apps to match their corresponding traffic that are used to extract the fingerprints. We find that in 3626 *weibo* messages, 6099 *zedge* messages and 1388 *weixin* messages, there are 2895, 5696, and 987 matches with fingerprints, respectively, which illustrate that most of the messages of the testing apps can be matched successfully with extracted fingerprints. The unsuccessful matches are due to the impact of messages generated from noise traffic. The results are shown in Figure 9.

The second experiment. In order to further verify the effectiveness of the fingerprints, we conducted the second experiment. We collected the three apps' pure raw traffic individually. But at this time, we need neither to run the apps many times nor to run all functions of the applications. We match the fingerprints we extracted previously with the traffic we collected this time, respectively. We observed that 600 out of 705 *weibo* messages are matched, 682 out of 731 *zedge* messages, and 290 out of 340 *weixin* messages were matched. The results are shown in Figure 10.

The third experiment. This experiment was conducted to validate the accuracy of fingerprints. To this end, we performed cross-verification among different apps. Concretely, we used the fingerprints from *zedge* and *weixin* to match the messages (705) from *weibo*. Similarly, we performed the matching between *weibo*, *weixin*, and *zedge*, and between *zedge*, *weibo*, and *weixin*. The outcomes are shown in Figure 11. The blue color represents the messages of the app whose fingerprints are used in this matching process, and the yellow and orange colors represent other messages of the remaining apps, which demonstrate the accuracy of the extracted fingerprints.

The fourth experiment. The fourth experiment uses the fingerprint of each of the three apps to separately match the mixed traffic of the three apps and other traffic we collected. The mixed traffic totally includes 2150 messages. According to the result of the matching step, we can infer that there are 402 messages belonging to *weibo*, 128 messages belonging to *weixin*, and 277 messages belonging to *zedge*. In addition, some unknown traffic is also included. The results are shown in Figure 12, and it shows that our method can effectively identify the fraction of app's traffic in network.

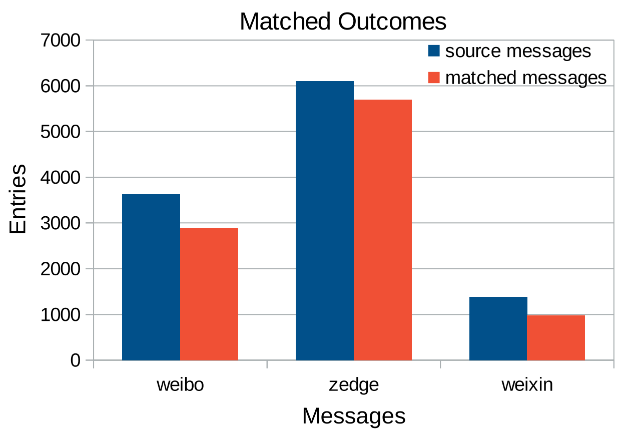

Figure 9. Matching with the traffic used extracting fingerprints.

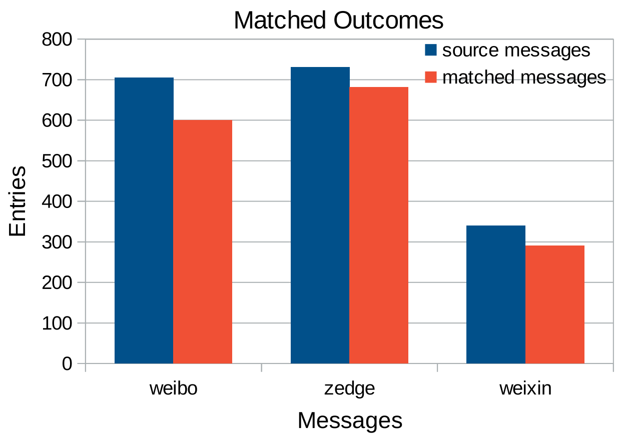

Figure 10. Matching with the new collected pure traffic.

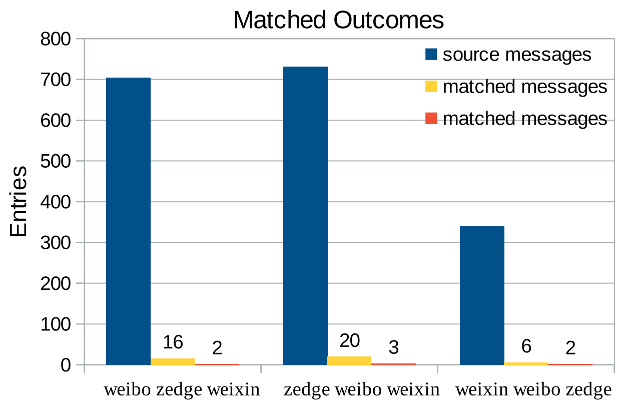

Figure 11. Matching with the fingerprints not belonging to the app.

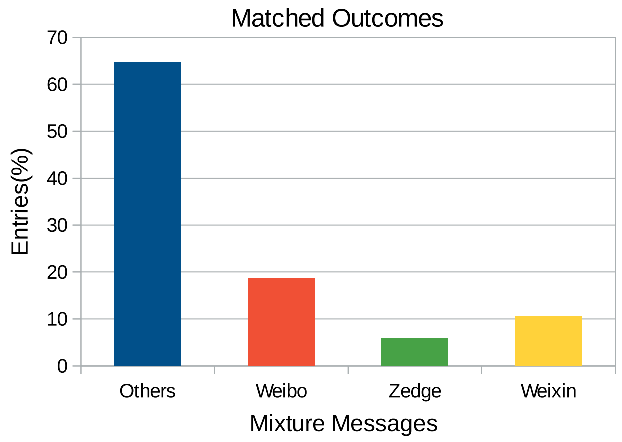

Figure 12. Matching with the mixed traffic.

The first experiment shows that our system can identify apps from the network trace, and in order to prove the accuracy of the experiments, we have tested three different apps. The second experiment prove the effectiveness of our extracted fingerprints. The third and the fourth experiments validate the accuracy of the fingerprints extracted. In summary, the results from these experiments indicate that our system can correctly recognize the apps occurring in the network traffic. The most important is that our method only depends on the network traffic and does not need to analyze the apps to classify them.

4.2. Performance

We conducted the performance evaluation from three aspects: execution time, complexity, and fault tolerance.

Execution time. First, we calculate the execution time of producing the fingerprints for the three apps. It takes about 62.10 s to extract the fingerprints for *weibo*, 63.64 s for *zedge*, and 61.04 s for *weixin*, which indicate the relatively low complexity in extracting fingerprints. On the other hand, we find that most of the execution time is spent on the NMF clustering algorithm. As for future work, we plan to leverage the parallel computing power of multi-core and GPU systems to improve the performance when dealing with larger dataset.

Second, we calculate the time of matching each app's fingerprints with the mixed traffic. The results are shown in Figure 13, which illustrates that the matching time increases with the increasing of the number of fingerprints and source messages (i.e., the raw traffic).

Complexity. Our system relies only on the raw network traffic to perform the fingerprints extraction with no need to access the source code or any specific information of the apps, as compared

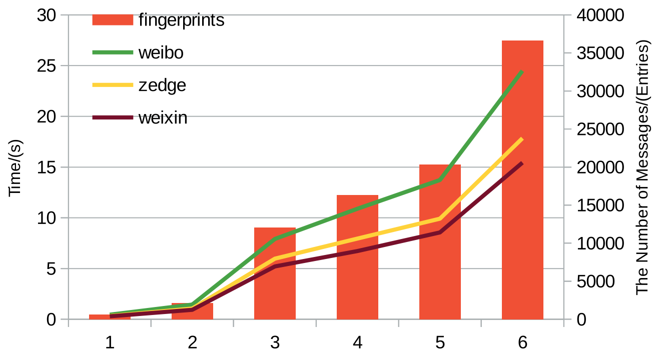

Figure 13. Time of matching.

with the system presented in [4] that classifies the Android apps' traffic to Ad and non-Ad traffic. For the Ad traffic, specific keywords from the manifest file of the apps are needed, which often incurs manual intervention and extra complexity. Our system directly extracts unique fingerprints from raw traffic without additional complexities.

Fault tolerance. First, in collecting the raw network traffic, it is inevitable to let some noise traffic such as the advertisement traffic sneak in. Second, in transforming raw packets to HTTP messages, we cannot completely prevent any errors because of protocol parsing, which would result in unnecessary messages occurring. Even with these potential noise data, our system can effectively extract fingerprints to identify mobile apps.

5. RELATED WORK

With the development of social network, the data generated from social network are becoming more and more huge. A large body of work have been conducted to analyze the social network. On the one hand, the analysis of social network concentrates on the relations among persons from survey or data obtained from research. These works include ONA surveys [11], which is based on the survey and data, Yed, which is based on Java program for drawing complex diagrams developed by the University of Tubingen [12], and the Proximity, an open-source software under development by the University of Massachusetts Knowledge Discovery Laboratory. They are specifically designed for social network analysis and other similar applications [13]. All of these methods focus on mapping social network.

On the other hand, for analysing the social network, many method based on network traffic classification have been put out. For classifying the network traffic and identifying the applications running in the network, they have proceed many research including the classification method based on the port number [14], the method based on payloads [15], and the clustering technique based on machine-learning [3, 16]. The approaches based on the port number are mainly designed for identifying different P2P protocols and are not suitable for other types of applications. The payload-based methods need the real payloads of packets that are often not accessible or are encrypted. In addition, there are some approaches proposed that focus on generating signature of Android apps using the statistical information such as packet sizes. All of these methods can classify apps traffic; however, these techniques do not work well for clustering mobile apps traffic and identifying them from the mixed network traces.

Considering the fact that a majority of the mobile apps' traffic is conveyed using the HTTP protocol, and can be distinguished from each other by their HTTP requests in the URLs and the hosts they connected, recently, many studies on Android app behaviors have been conducted [3, 16]. But they do not propose a systematic method to classify the mobile app traffic to perform app identification. NetworkProfiler [4] analyzed the Android app network behaviors to extract the fingerprints for Android apps, but it involves additional complexity such as manually analyzing specific information in apps' code to aid the fingerprints generation. AppPrint [17] is a system that learns fingerprints of mobile apps via comprehensive traffic observations. These fingerprints identify apps even in small traffic samples. Rieck and Laskov [18] presented a new clustering technique for network traffic, which can classify the network traffic with high accuracy. There has been much work on analyzing Android apps for malware detection. But most of these target at monitoring apps [19] or static analysis of application code [20]. None works well for our purpose of a simple and effective app identification system for mobile networks.

6. CONCLUSIONS

In this paper, we present the design and implementation of a system to extract the fingerprints from mobile apps. The fingerprints can be used to uniquely identify apps in mobile network traffic. The proposed approach is superior in terms of automation and efficiency as compared to previous methods. The experimental evaluations show that our system can identify the mobile apps from random and mixed network traffic with high precision. As for future work, we are planning to improve the scalability of our approach and enhance our system to handle real-time network traffic analysis and support fingerprints update.

ACKNOWLEDGEMENTS

This research was supported in part by the National Natural Science Foundation of China under grants 61272190 and 61173166, the Program for New Century Excellent Talents in University, and the Fundamental Research Funds for the Central Universities.

REFERENCES

1. Zhang Y, Arvidsson A. Understanding the characteristics of cellular data traffic. *SIGCOMM Computer Communication Review* September 2012; **42**(4):461–466.
2. Xiong G, Meng J, Cao Z, Wang Y. Research progress and prospects of network traffic classification. *Integrated Technology* January 2012; **1**(1):32–42.
3. Xu Q, Erman J, Gerber A, Mao Z, Pang J, Venkataraman S. Identifying diverse usage behaviors of smartphone apps. *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, ACM, New York, NY, USA, 2011; 329–344.
4. Dai S, Tongaonkar A, Wang X, Nucci A, Song D. NetworkProfiler: towards automatic fingerprinting of android apps. *INFOCOM13*, IEEE, Turin, Italy, 2013; 809–817.
5. Azim T, Neamtiu I. Targeted and depth-first exploration for systematic testing of android apps. In *ACM SIGPLAN Notices - OOPSLA '13*, Vol. 48 no.10. ACM: New York, NY, USA, 2013; 641–660.
6. Wireshark. (Available from: http://www.wireshark.org.) [Accessed on 9 November 2015].
7. Justniffer. (Available from: http://justniffer.sourceforge.net.) [Accessed on 9 November 2015].
8. Sally. (Available from: http://www.mlsec.org/sally.) [Accessed on 9 November 2015].
9. Holm S. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics* 1979; **6**(2):65–70.
10. Lee DD, Seung HS. Algorithms for non-negative matrix factorization. In *Advances in Neural Information Processing Systems 13*, Leen TK, Dietterich TG, Tresp V (eds). MIT Press: Denver, United States, 2001; 556–562.
11. Onasurveys. (Available from: http://www.onasurveys.com.) [Accessed on 9 November 2015].
12. Yed. (Available from: http://www.yworks.com/en/products_yed_about.htm.) [Accessed on 9 November 2015].
13. Proximity. (Available from: http://kdl.cs.umass.edu.) [Accessed on 9 November 2015].
14. Sen S, Spatscheck O, Wang D. Accurate, scalable in-network identification of P2P traffic using application signatures. *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, ACM, New York, NY, USA, 2004; 512–521.
15. Karagiannis T, Papagiannaki K, Faloutsos M. Blinc: Multilevel traffic classification in the dark. *SIGCOMM Computer Communication Review* August 2005; **35**(4):229–240.
16. Frank J, Mda-c NU. Artificial intelligence and intrusion detection: current and future directions. In *Proceedings of the 17th National Computer Security Conference*, Location: Baltimore, Maryland, 1994; 22–33.
17. Miskovic S, Lee GM, Liao Y, Baldi M. AppPrint: automatic fingerprinting of mobile applications in network traffic. *PAM 2015*, Springer, New York City, NY, 2015; 57–69.
18. Rieck K, Laskov P. Linear-time computation of similarity measures for sequential data. *Journal Machine Learning Research* June 2008; **9**:23–48.
19. Enck W, Gilbert P, Chun BG, Cox LP, Jung J, McDaniel P, Sheth AN. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communication of the ACM* March 2014; **57**(3):99–106.
20. Grace MC, Zhou W, Jiang X, Sadeghi AR. Unsafe exposure analysis of mobile in-app advertisements. *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, ACM, New York, NY, USA, 2012; 101–112.