Pham, Thai-Dien; Ho, Thien-Lac; Truong-Huu, Tram; Cao, Tien-Dung; Truong, Linh

MAppGraph: Mobile-App Classification on Encrypted Network Traffic using Deep Graph Convolution Neural Networks

Published in:
Proceedings - 37th Annual Computer Security Applications Conference, ACSAC 2021

DOI:
10.1145/3485832.3485925

Published: 06/12/2021

Document Version
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

Please cite the original version:
Pham, T.-D., Ho, T.-L., Truong-Huu, T., Cao, T.-D., & Truong, L. (2021). MAppGraph: Mobile-App Classification on Encrypted Network Traffic using Deep Graph Convolution Neural Networks. In *Proceedings - 37th Annual Computer Security Applications Conference, ACSAC 2021* (pp. 1025–1038). ACM.
https://doi.org/10.1145/3485832.3485925

MAppGraph: Mobile-App Classification on Encrypted Network Traffic using Deep Graph Convolution Neural Networks

Thai-Dien Pham*
Tan Tao University, Vietnam
dien.pham1702002@std.ttu.edu.vn

Thien-Lac Ho*
Tan Tao University, Vietnam
lac.ho1702005@std.ttu.edu.vn

Tram Truong-Huu†
Singapore Institute of Technology
truonghuu.tram@singaporetech.edu.sg

Tien-Dung Cao
Tan Tao University, Vietnam
dung.cao@ttu.edu.vn

Hong-Linh Truong
Aalto University, Finland
linh.truong@aalto.fi

ABSTRACT

Identifying mobile apps based on network traffic has multiple benefits for security and network management. However, it is a challenging task due to multiple reasons. First, network traffic is encrypted using an end-to-end encryption mechanism to protect data privacy. Second, user behavior changes dynamically when using different functionalities of mobile apps. Third, it is hard to differentiate traffic behavior due to common shared libraries and content delivery within modern mobile apps. Existing techniques managed to address the encryption issue but not the others, thus achieving low detection/classification accuracy. In this paper, we present MAppGraph, a novel technique to classify mobile apps, addressing all the above issues. Given a chunk of network traffic generated by a mobile app, MAppGraph constructs a communication graph whose nodes are defined by tuples of IP address and port of the services connected by the app, edges are established by the weighted communication correlation among the nodes. We extract information from packet headers without analyzing encrypted payload to form feature vectors of the nodes. We leverage deep graph convolution neural networks to learn the diverse communication behavior of mobile apps from a large number of graphs and achieve a fast classification. To validate our technique, we collect traffic of a hundred mobile apps on the Android platform and run extensive experiments with various experimental scenarios. The experimental results show that MAppGraph significantly improves classification accuracy by up to 20% in various metrics compared to recently developed techniques and demonstrates its practicality for security and network management for mobile services.

CCS CONCEPTS

• **Security and privacy** → **Mobile and wireless security**.

KEYWORDS

Mobile-App Classification, Graph Convolution Neural Networks, Deep Learning, Traffic Engineering, Network Security

ACM Reference Format:
Thai-Dien Pham, Thien-Lac Ho, Tram Truong-Huu, Tien-Dung Cao, and Hong-Linh Truong. 2021. MAppGraph: Mobile-App Classification on Encrypted Network Traffic using Deep Graph Convolution Neural Networks. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3485832.3485925

1 INTRODUCTION

The proliferation of communication technologies including 5G enables a remarkable growth of smart and mobile devices. It is widely predicted that the number of Internet of Things (IoT) devices including smartphones will attain a few tens of billions by 2023 [18]. Besides daily functionalities such as financial payments, voice-over-IP (VoIP) and video live streaming, today's mobile applications are increasingly being used for many real-time and high bandwidth services such as virtual reality or augmented reality games. The security of mobile devices and their applications has a great impact on the security of the underlying networks, which are more and more complex and vulnerable. Mobile service operators need to protect not only their networks but also application and platform services deployed in their infrastructures. However, they do not have control over the applications installed on each device in their network. Thus, addressing the problem of mobile and network security becomes more challenging. A possible solution that operators can still retain the capability of detection and monitoring of active apps is to observe the network traffic behavior of each device. Though this may lead to the problem of user privacy as mobile apps can reveal a lot of sensitive information about users, it provides a non-intrusive approach without requiring host (device) access.

Network traffic classification has been extensively studied in the literature [29]. Most of existing techniques focused on the traffic of traditional computer networks [10, 16, 24, 25, 35, 36, 39, 46]. Adopting such techniques to handle network traffic generated by modern apps in smartphones faces several challenges. First, according to [5]

*Both authors contributed equally to the work.
†This work has been done when T. Truong-Huu was with Institute for Infocomm Research (I²R), Agency for Science, Technology and Research (A*STAR), Singapore.

ACSAC '21, December 6–10, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3485925

more than 80% of mobile traffic is encrypted or adopts Transport Layer Security (TLS), thus it might not be possible to classify traffic using payload-based methods that analyze certain fields of application layer protocols. Second, port-based classification methods fail to classify mobile traffic as apps deliver their data predominantly using HTTPS and send data back and forth using text formats such as XML or JSON. Some information such as the number of files or file size are not available to adopt web page classification. Third, user behavior changes dynamically over time depending on the used functionalities. The traffic captured in a short period (e.g., 5 minutes) of a mobile app may not represent its complete traffic behavior. For instance, a user surfs `Facebook` to update all the new feeds for 10 minutes and then switches to `Facebook Watch` or `Facebook Live` to watch available videos. Even though the user uses the same app, the traffic behavior is much different between updating new feeds from `Facebook` servers and streaming videos from different sources (e.g., YouTube). Given the traffic of an app captured at different times, it is challenging to identify a unique signature that can be used to classify the app. Last but not least, to optimize the performance, modern mobile apps are designed and developed using micro-services that share the same libraries and use content delivery networks (CDNs) and third-party services [50]. For such apps, using domain name resolution or IP address lookup for app classification may not be useful. Yet, with local cache, DNS and TLS exchanges may not be observed at all. This raises the need for an effective approach to mobile-app classification that can deal with encrypted traffic, and be aware of diverse functionalities and the nature of sharing third-party services of mobile apps.

Existing works on mobile-app classification mostly overcome the challenge of encrypted traffic but fail to address the remaining issues. For instance, AppScanner [40] used a flow-based detection approach that extracts side-channel features from the packet header and computes statistical features to train a machine learning model for mobile app classification. As apps use CDNs or shared services, similar flow characteristics will be observed, thus confusing the classification model [40, 41, 44]. Examples of shared services include crash analytics, mobile advertisement (ad) networks, social networks. These services are often embedded through libraries that are used by many apps: e.g., `googleads.g.doubleclick.net`, `lh4.googleusercontent.com` and `android.clients.google.com`. Motivated by this issue, the authors of FlowPrint [44] proposed to construct the fingerprint of an app by considering the communication graph between the mobile device and other destinations (e.g., CDNs and third-party services) and the associated attributes such as destination IP, destination port and TLS-certificates. At the inference stage, the fingerprint collected in the past is compared with the new one to determine the app. However, due to the challenge in building the communication graphs for all possible behavior of an app, the author only considered a short communication period (e.g., 5 minutes). Thus, it may not work well if the user changes her usage behavior or uses different functionalities of the app.

In this paper, we develop MAppGraph, a novel technique for mobile app classification addressing the above challenges. We collect network traffic of mobile apps at different times, resulting in a large amount of network traffic for each app to be processed. Each chunk of traffic (e.g., within 5 minutes) forms a communication graph where nodes are defined by the tuples of IP address and port of the destination services, which are accessed by the apps during that time window. We adopt the cross-correlation approach [32] to establish the edges between nodes and compute their weight. We extract information from packet headers such as packet size, inter-arrival time of packets, etc., and derive statistical features, which are used as attributes of the graph nodes to represent traffic behavior of the communication between the app and a service.

With a large amount of traffic collected, one mobile app will have a set of graphs, each representing traffic behavior at different time instants. Learning or identifying the fingerprint of each mobile app through this big set of graphs is challenging. The emergence of graph neural networks (GNN), specifically deep graph convolution neural networks (DGCNN) [49], has demonstrated the capability of learning complex behavior and extracting high-level features from big data to create the signature or fingerprint of analyzed data. DGCNN can be naturally adapted to address the problem of mobile-app classification and fingerprinting based on network traffic, which can be represented as graphs. DGCNN can be used for both supervised and unsupervised learning problems depending on the problem objective. To this end, in MAppGraph, we exploit the capability of DGCNN in a supervised learning manner for the mobile-app classification problem and keep the unsupervised learning for future work. We collect traffic of 101 mobile apps, which are popular in Google Play. For each app, more than 30 hours of traffic were collected, resulting in more than 600 GB of traffic stored in PCAP files. We run extensive experiments to demonstrate the effectiveness of MAppGraph. We compare its performance with recent techniques, specifically AppScanner [40] and FlowPrint [44] as well as multi-layer perceptron (MLP). The main contribution of the paper is summarized as follows:

- We develop a method for processing network traffic and generating graphs with node features and edge weights that better represent the communication behavior of mobile apps.
- We develop a DGCNN model that is able to learn the communication behavior of mobile apps from a large number of graphs and achieve a fast mobile-app classification.
- We collect traffic for 101 mobile apps and run extensive experiments to demonstrate the effectiveness of MAppGraph.
- We enhance AppScanner and FlowPrint and use as baselines for performance comparison with MAppGraph.

We have made both our prototype and a portion of dataset available at https://soeai.github.io/MAppGraph.

The rest of the paper is organized as follows. Section 2 presents existing literature. Section 3 presents some background on deep graph neural networks and their application to the problem of mobile-app classification. Section 4 presents the details of MAppGraph. Section 5 present the experiments and analysis of results before we conclude the paper in Section 6.

2 RELATED WORK

Network-centric techniques that are based on traffic analysis have been used extensively in various problems such as network management and cybersecurity in both traditional computer networks and mobile (IoT) networks. In traditional computer networks, network-centric techniques have been used extensively in cybersecurity

problems such as network anomaly detection and attack detection [22, 28, 43]. Those works focused on collecting basic traffic features at the data link layer and network layer of the TCP/IP network model and calculating additional statistical features such as mean and standard deviation of packet rate, packet size, inter-arrival time between packets, number of packets in a flow, flow rate, flow duration, and number of flows in a session. Several works such as [2, 3] also use traffic analysis-based techniques for malware detection, applying in the scenarios where malware samples attack the victim from a remote source, e.g., botnet traffic.

In IoT networks, traffic analysis-based techniques have been used for multiple purposes. In [42], the authors developed a distributed device fingerprinting technique to exploit the presence of common IoT devices and detect new devices across smart home networks and enterprise networks. In [4], the authors showed that sensitive information can be inferred from the IoT devices in smart homes. The authors also demonstrated that users' usage patterns can be identified by analyzing the traffic of user devices, MAC address, DNS queries and traffic rates. As using traffic can identify IoT devices, in [38], the authors developed a multivariate Gaussian distribution, which is trained on device traffic in a supervised learning approach, resulting in a machine learning model used to fingerprint device and authentication. As they used machine learning, the performance of the model may degrade due to changes in domains, i.e., from one smart home to another smart home. The authors applied transfer learning [30] to minimize the performance degradation and adapt to new domains. Similar works have been presented in [26, 27], which also proposed to extract traffic features from the network, transport and application layers to train machine learning models for device fingerprinting and security enforcement. In [26], the authors also proposed to gather the data from external resources such as Alexa top website ranking[1] and geo-location of IP addresses. All of the works aimed to identify IoT devices as soon as possible after they join the network. Thus, in [27], the authors aimed to minimize the number of traffic packets required to successfully identify the devices. Nevertheless, too few packets may not be sufficient to represent the device traffic pattern.

Mobile-app classification based on network traffic has been studied in the literature [9, 11–13, 40, 44]. In [11, 12], the authors proposed to used machine learning and deep learning for mobile-app classification based on HTTP headers. In [9, 13], the authors developed the techniques for creating fingerprints automatically. However, those approaches are based on deep packet inspection, which will not work when traffic is encrypted. AppScanner [40] uses statistical features of packet sizes in the TCP streams to train Support Vector Machine (SVM) and Random Forest (RF) to learn traffic patterns of mobile apps and classify known apps during the inference phase. Similar to AppScanner, BIND [1] uses supervised learning techniques to learn traffic behavior of mobile apps based on statistical features of TCP streams combined with temporal features to achieve higher performance compared to AppScanner. Unfortunately, to optimize app performance, most mobile apps nowadays share the same third-party services such as Content Delivery Networks, making their TCP streams much similar and harder to recognize apps. Another supervised learning technique has been proposed in [31], which showed that mobile apps can also be identified even though the traffic is anonymized through Tor.

Recently, FlowPrint [44] has been proposed. The authors considered the spatial correlation of traffic flows between the mobile apps and other destinations such as app servers and third-party services and developed a technique to create mobile-app fingerprints for app identification. However, a mobile app may have multiple fingerprints associated with various functionalities. Within a short duration (e.g., 5 minutes), a user may not use all the functionalities of the app so that its created fingerprint represents all its traffic behavior. Inspired by FlowPrint with an aim to address the diverse behavior of mobile apps, we leverage the capability of deep graph convolution neural networks, which can represent both statistical traffic features by the node attributes and communication correlation among nodes by the edges. With multiple graphs captured at different times, we train a deep learning model to classify mobile apps regardless of the user behavior and functionalities used.

3 BACKGROUND

3.1 Graph for Data Representation and GNN

Graphs and their methods deal with abstract concepts such as relationships and interactions. They provide an intuitively visual way of thinking about these concepts. With the prevalence of graph-structured data, there is an increasing need for graph representation learning [8]. Mathematically, a graph $\mathcal{G}$ is defined by a tuple $(\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of nodes, $\mathcal{E}$ is the set of edges linking the nodes. Each node, $v \in \mathcal{V}$ is associated with a d-dimensional vector of features, $x_v \in \mathbb{R}^d$ and denote the set of all features of nodes as matrix $\mathcal{X} := \{x_v : v \in \mathcal{V}\} \in \mathbb{R}^{d \times |\mathcal{V}|}$. For every edge that connects nodes $u, v \in \mathcal{V}$, we denote $e_{u,v} \in \mathbb{R}$ to be a weight of the edge $(u, v) \in \mathcal{E}$. In Figure 1, we present an example of graph-structured data. Analyzing graph-structured data is a challenging task due to the fact that graphs do not exist in a Euclidean space and they do not have a fixed form even though two graphs may have the same adjacency matrix. In [6], Bruna *et al.* used convolutional neural networks to represent graphs in the spectral domain. In [49], Zhang *et al.* further improved the graph representation with a new deep neural network architecture, namely Deep Graph Convolution Neural Network (DGCNN), that can keep much more vertex information and learn from the global graph topology. Such generalization aims to learn meaningful embeddings (i.e., vector representations) of nodes and/or (sub)graphs. Such embeddings can be used in various downstream tasks, such as node classification [45], link prediction [37], and graph classification [49]. Such models are usually evaluated on chemical and social domains [47]. In this work, we adopt graph neural networks for cybersecurity to represent mobile traffic in a meaningful manner and achieve better performance for the mobile-app classification problem. As DGCNN has demonstrated its superior performance compared to existing approaches [49] in the graph classification problem, we adopt DGCNN and carry out optimization on network architectures and learning parameters.

3.2 DGCNN for Graph Classification

In general, graph classification involves differentiating graph instances of different classes and predicting the label of an unknown

[1] Alexa Top Site: https://www.alexa.com/topsites; accessed: May 2021

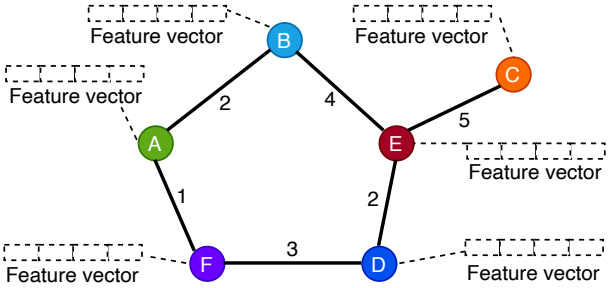


Figure 1: An Example of Graph-structured Data.

graph. Mathematically, given $\mathcal{D} := \{(\mathcal{G}_1, y_1), (\mathcal{G}_2, y_2), ...\}$ where $y_i \in \mathcal{Y}$ is the class label of the graph $\mathcal{G}_i \in \mathbb{G}$, the goal of graph classification is to learn a mapping $f : \mathbb{G} \rightarrow \mathcal{Y}$ that maps graphs to the set of class labels and predicts the class labels of unknown graphs. This task requires a graph representation vector distinctive enough to distinguish graphs of different classes. There exist several graph neural network architectures that can be used for graph classification such as DiffPool [48], DGCNN [49] and 2STG+ [14]. The common point of those architectures is that they consist of three sequential stages in the graph analysis process:

- First, a given graph is passed through the graph convolution layers that extract local features and define a consistent vertex ordering;
- Second, the pooling layers will process the output of the first stage and build a vector representation of the entire graph with predefined order and size;
- Third, they employ traditional convolutional or dense (fully connected) layers that take the output of the second stage and make the prediction.

For the sake of completeness, we present below the details of each stage of the graph analysis process. We refer the reader to [49] for further detailed mathematical proofs of the approach.

3.2.1 Graph Convolution Layers. Given a graph $\mathcal{G} := (\mathcal{V}, \mathcal{E})$, let $\mathcal{A}$ be the adjacency matrix of $\mathcal{G}$ such that $\mathcal{A}$ is a symmetric binary matrix with the assumption that the graph has no self-loops. The node feature matrix is defined as $\mathcal{X} \in \mathbb{R}^{n \times d}$ where $n = |\mathcal{V}|$. A graph convolution layer computes node latent representation as $Z = \sigma(\tilde{D}^{-1}\tilde{\mathcal{A}}\mathcal{X}W)$ where $\tilde{\mathcal{A}} := \mathcal{A} + I$ is the adjacency matrix with added self-loops, $\tilde{D}$ is the diagonal degree matrix of the graph such that $\tilde{D}_{i,i} := \sum_j \tilde{\mathcal{A}}_{i,j}$, $W \in \mathbb{R}^{d \times d}$ is a matrix of trainable graph convolution parameters that are shared among nodes, σ is a non-linear activation function, and $Z \in \mathbb{R}^{n \times d}$ is the output activation matrix.

Intuitively, nodes in a graph are defined by their neighbors and connections. Thus, the latent representation of a node is also affected by that of its neighbors. The graph convolution layers reflect this rationale by starting from node features ($\mathcal{X}W$), allow information to propagate between neighboring nodes by the product of node features and adjacency matrix ($\tilde{\mathcal{A}}\mathcal{X}W$). To compute multi-scale node latent representation, multiple graph convolution layers can be stacked. The final node latent representation is computed as $Z^{l+1} = \sigma(\tilde{D}^{-1}\tilde{\mathcal{A}}Z^lW^l)$ where $l = 0 \ldots L-1$, $Z^0 := \mathcal{X}$, $Z^l \in \mathbb{R}^{n \times d_l}$ is the output of the l^{th} graph convolution layer; d_l is the number

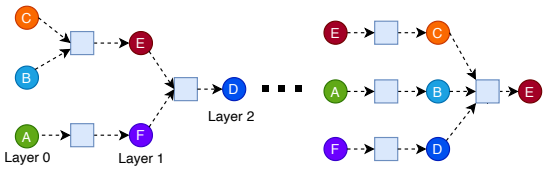


Figure 2: Information Propagation Between Nodes with Multiple Convolution Layers.

of the output channels of the l^{th} layer (i.e., the number of features of each graph node extracted at the l^{th} layer); $W^l \in \mathbb{R}^{d_l \times d_{l+1}}$ is trainable parameters of the l^{th} layer. In Figure 2, we illustrate how the features of a node are propagated to its neighbors during the convolution process given the graph example depicted in Figure 1. After the completion of the convolution process through all convolution layers, we obtain node latent representation that can learn from the entire graph topology.

3.2.2 Pooling Layers. Given node latent representation obtained from the first stage through graph convolution layers, the pooling layers aim at combining latent representation of all the nodes into a vector with a predefined order and size. The output of the pooling layers is a latent representation of a graph. There exist several pooling approaches such as global pooling [15] and hierarchical pooling [21, 48]. Global pooling is the simplest approach that takes the mean and max of the final node latent representation. Hierarchical pooling step-by-step reduces the number of nodes (after one or multiple graph convolution layers) either by merging similar nodes to super-nodes [48] or selecting the most significant nodes [21] until a single super-node is found. In DGCNN [49], the authors developed SortPooling algorithm for sorting nodes by the sum of node features at the L^{th} layer, which is the last layer in the graph convolution stage. If two nodes have the same value at the l^{th} layer, the sum of node features at the $(l-1)^{\text{th}}$ layer is used until ties are broken. Since the number of nodes in each graph is heterogeneous, pooling layers also perform truncation or extension of the graph latent representation to a predefined size, which is then fed to the third/last stage of the graph classification process. Given a predefined size of the graph latent representation (say k), if there are more than k values in the graph latent representation vector, truncation is performed. Otherwise, zero-padding is performed. The value of k is defined heuristically based on the input data. For instance, k is defined such that 90% of graph nodes will be used to construct the graph latent representation vector so as to avoid loss of node features in the final graph latent representation.

3.2.3 Neural Network Layers. After pooling layers, each graph is represented by a latent vector. To further learn the local pattern of graphs, one or multiple 1-dimensional convolution layers associating with `MaxPooling` layers are applied before using fully-connected layers followed by a `softmax` layer to predict the class.

3.3 Network Traffic Collection

We assume that mobile traffic collection can be done by network operators using available network monitoring tools [20], which capture the traffic without interfering with app functionalities. As our technique does not require the payload, it can be discarded to

reduce the storage space needed. To prepare the training dataset for the classifier, a large amount of traffic needs to be collected and processed. While this task can be done in an offline manner, the inference phase needs to be done in real-time so as to react to the abnormal behavior of mobile apps. After being analyzed by the classifier, the traffic samples collected during the inference phase can also be discarded or stored to enrich the training dataset.

4 MOBILE-APP CLASSIFICATION USING DEEP GRAPH CONVOLUTION NEURAL NETWORKS

We now present MAppGraph, a novel technique that adopts DGCNN to cybersecurity for the problem of mobile-app classification. We first present the most important task that is the construction of graphs of mobile-app traffic behavior and node feature extraction. We then present the DGCNN architecture used in our work.

4.1 Construction of Traffic Behavior Graphs

4.1.1 Construction of Graph Topology. Given that network traffic of mobile apps is captured with a predefined time window (T_{window}), say 5 minutes, the construction of a traffic behavior graph involves node and edge determination to correlate nodes and weight computation for the edges. It should be noted that the longer the time window, the more the traffic is captured and the diverse the app behavior is observed. Obviously, the longer the time window, the longer the time needed before the app is classified. This may affect the security of the networks or mobile service platform as the app can exhibit malicious behavior that needs to be detected as soon as possible. At first glance, a naive solution is to construct a traffic behavior graph by considering the IP addresses (including the IP address of the device) as graph nodes and connecting those nodes based on traffic flows exchanged among them. However, this makes all the graphs in the star form where the device IP address is the central node that has connections to all the remaining nodes. Furthermore, as mobile apps share third-party services, the star-formed graphs of many apps will be similar with the same number of nodes.

To overcome this issue and produce a better graph representation of the traffic behavior of mobile apps, we define a graph node by a tuple of a destination IP address and the port number of the service that the app connects to. Multiple third-party services may be deployed under the same server (i.e., the same IP address) but with different port numbers. For instance, SMTP Email and DNS services may have the same IP address but using different port numbers (587 for the secure email service and 53 for the DNS service). It is to be noted that third-party services are usually deployed in multiple servers (with different IP addresses) to provide load balancing. The requests from mobile apps will be distributed to a specific service based on their load. For this reason, using a rule-based classification approach with the destination IP address may not work well. Nevertheless, the port number does not change and the number of destination nodes remain unchanged.

The challenging issue is how to determine whether two nodes are correlated such that an edge between them needs to be established. To achieve this goal, we adopt the temporal correlation concept presented in [44], which is in turn based on the cross-correlation [32] between activities of all node pairs. Given the traffic captured in a time window that results in the list of graph nodes determined as discussed above, the time window is further divided into multiple slices with a predefined slice duration (t_{slice}, say 10 seconds). Let T denote the number of slices. During each slice, we consider a node (i.e., a pair of a destination IP address and a port number) is active if the mobile app has sent or received at least one traffic packet to the service deployed at the destination IP address and the port number. Let $a_i(t)$ be a binary variable indicating whether node i is active at time slice t or not such that $a_i(t) = 1$ if node i is active and $a_i(t) = 0$, otherwise. The cross-correlation between node i and node j during T slices is defined as $C_{i,j} = \sum_{t=1}^{T} a_i(t) \cdot a_j(t)$.

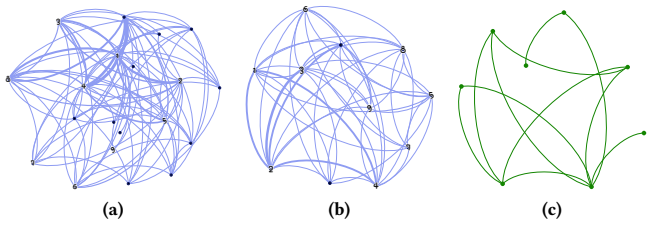


Figure 3: Examples of Traffic Behavior Graphs of Facebook (a and b) and Spotify (c) Constructed by Our Approach.

The cross-correlation between two nodes ($C_{i,j}$) is high if they have a lot of activities in the same time slices. Otherwise, $C_{i,j}$ is low or even equal to zero if there is no correlation between them. Using cross-correlation, we can establish edges among nodes and set the weight for the edges accordingly. If $C_{i,j} \neq 0$, an edge between node i and node j is established with the weight $C_{i,j}$. To avoid the feature bias when feeding graphs to DCGNN for training and prediction, we can normalize the cross-correlation to the range $[0, 1]$ using a min-max scaler. Using the graph construction technique presented above, we achieve the graph representation of communication behavior of mobile apps as shown in Figure 3 with three example graphs of Facebook and Spotify. The thickness of the edge between two nodes indicates the weight of the edge (i.e., the cross-correlation between the nodes).

4.1.2 Extraction of Node Features. We need to construct a feature vector for each node in the graphs. Since a mobile app connects to various services, each being represented by a node in the graph as a tuple of an IP address and a port, the behavior of traffic from the mobile device to the server of each service may be different in terms of various traffic features such as packet size, number of packets, flow duration, etc. To generalize our technique (MAppGraph) to both encrypted and unencrypted traffic, we extract information only from packet headers without analyzing packet payloads. Apart from packet features, we also consider flow features such as the number of flows, mean number of packets in each flow and mean flow size in bytes. In this work, we consider only TCP and UDP flows and we rely on Wireshark to collect traffic features. In practical scenarios, an online traffic feature extraction tool such as [43] is needed to process the traffic stream on the fly. It is also to be noted that a flow is different from a TCP session, which is defined with TCP flags and can last for a long duration. As defined in [43], a flow is a sequence of packets that have the same tuple of source IP,

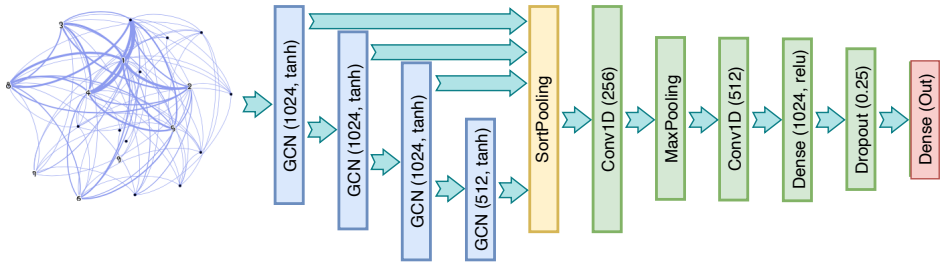


Figure 4: Architecture and Parameters of MAppGraph Model for Mobile App Classification.

destination IP, source port destination port and protocol, and the inter-arrival time between two consecutive packets is shorter than a predefined threshold (e.g., 3 seconds). From the basic features, we derive statistical features that are useful for DGCNN to learn the communication behavior of mobile apps. There is a total of 63 features, which are extracted and derived from the traffic. We classify these features into 4 categories.

- Aggregated features: These are the features computed as sum, total count, max and min values of basic features over the time window of traffic capture (e.g., 5 minutes). These include the total number of packets, the total number of bytes and the total number of flows.
- Temporal features: Temporal features include the flow duration, the mean and standard deviation of flow duration.
- Statistical features: These are the mean, median absolute deviation (MAD), variance, skew, kurtosis and standard deviation (in the short stand. dev.) of packet size (in bytes), number of packets, number of bytes, flow size (in bytes) over the time window of traffic capture (e.g., 5 minutes). We separate incoming and outgoing packets and extract features before aggregating them all together to provide more statistical features.
- Categorical features: Categorical features such as transmission protocol and IP address need to be factorized before being fed to MAppGraph. Each IP address is factorized and normalized into 4 features, each representing a component of the address. For instance, the IP address 223.12.45.68 is factorized as 4 features $(223/255, 12/255, 45/255, 68/255)$.

In Table 1, we summarize the list of traffic features used for attributing graph nodes. Instead of performing feature selection and using only important features for attributing graph nodes, we leverage the capability of deep learning that can learn latent characteristics of data and feed all the possible features to MAppGraph. On one hand, this implicitly relieves the effort of feature engineering and selection. On the other hand, this provides maximum information that can be collected from network traffic to achieve the highest performance in mobile app classification.

4.2 MAppGraph Model Architecture

In Figure 4, we present the architecture and detailed parameters of the MAppGraph model obtained after a parameter tuning process. We employ 3 graph convolution layers, each having the size of 1024. The `SortPooling` layer that follows the graph convolution layers has the size of 512. The activation function used in graph convolution layers and the `SortPooling` layer is `tanh`. In the classification stage, we employ two traditional 1-dimensional convolution layers, between them a `MaxPooling` layer is integrated. The output of the 1-dimensional convolution layers is flattened before being fed to a fully-connected layer with a size of 1024 with a `ReLU` activation function. Before the `softmax` layer for classification, a `Dropout` layer with a dropout probability of 0.25 is used to prevent the model from overfitting. We use Adam optimizer [19] during the training of MAppGraph.

Table 1: Traffic features used for attributing graph nodes

No.	Feature	Category
1	Number of incoming packets	Aggregate
2	Max of incoming packet size	Aggregate
3	Min of incoming packet size	Aggregate
4	Mean of incoming packet size	Statistical
5	MAD of incoming packet size	Statistical
6	Stand. dev. of incoming packet size	Statistical
7	Variance of incoming packet size	Statistical
8	Skew of incoming packet size	Statistical
9	Kurtosis of incoming packet size	Statistical
10-18	$10-90$ percentile of incoming packet size	Statistical
19-37	Features 1-18 for outgoing packets	
38-56	Features 1-18 on both types of packets	
57	Mean flow size (in bytes)	Statistical
58	Mean flow duration (in seconds)	Temporal
59	Stand. dev. of flow duration (in seconds)	Temporal
60	Total number of flows	Aggregate
61	Mean number of packets in each flow	Statistical
62	Transmission protocol	Categorical
63	IP address of the node	Categorical

5 EXPERIMENTS

5.1 Data Collection

There exist several datasets that contain encrypted traffic of mobile apps such as ReCon [34], Cross Platform [33] and ANDRUBIS [23]. These datasets contain the traffic of a large number of mobile apps on various platforms such as Android and iOS, i.e., up to 512 mobile apps. However, the duration of traffic capture for each mobile app is not long enough (the longest average duration is 339.4 seconds [44]) for observing diverse behavior and various functionalities of the apps. We note that existing work such as [44] needs at least 300 seconds of traffic capture to create fingerprints of an app. Furthermore, these pre-processed datasets do not provide all features required in

our technique. Thus, to capture the traffic of diverse user behavior and various functionalities of mobile apps, we decided to carry out our own traffic data collection.

We focused on Android apps, which can run on various brands of smartphones such as Samsung, Xiaomi, Google Pixel, etc. Our research team along with a recruited team of 10 students (see Appendix A) carried out traffic collection within 6 months with 8 smartphones. We setup a controlled data collection environment, in which the smartphones are connected to a WiFi router where we mirror the traffic to a desktop with sufficient storage space. All the mobile apps are accessible within the university campus through the university network. On each mobile device, apart from the core Android apps of the OS, only one app (among 101 mobile apps mentioned in Appendix B) is in use. All the traffic generated from the mobile device is labeled with the in-use app. Many mobile devices can be used at the same time to collect traffic from different apps. We recognize them based on the IP address of the mobile devices.

During the project lifetime, multiple data collection sessions have been carried out. In each session (around 3-4 hours), the volunteering students (the number of students who join in each session varies) will get a smartphone (provided by the research team) and access one of the apps in the list (presented in Appendix B). With this randomness, we believe the collected traffic reflects the common user behavior in using common apps (of course maybe some apps will not be accessed by the user or the user behavior change will access some personal services) and all users may not have the same usage behavior. Nevertheless, we believe it is a very challenging problem in controlled experiments using mobile apps when users volunteer to do the pilot. This is out of our control.

As a result, we managed to collect the traffic for 101 mobile apps, which are popular (testified by the number of installs) in Google Play. For each app, we intentionally captured more than 30 hours of traffic stored in PCAP files, each belonging to a particular app. In Table 6 of Appendix B, we present all the mobile apps that we have collected their traffic. With the PCAP files, we performed graph construction for each mobile app with a sliding window of $T_{\text{window}} = 5$ minutes (if not stated otherwise). To provide more graphs to train the MAppGraph model, we set an overlapping duration window to 3 minutes. This results in at least 800 graphs for each mobile app in our experiments. In Figure 5, we present data pre-processing to construct graphs and the workflow for experiments and performance comparison. We randomly split these graphs into training and test sets with a ratio 80 : 20.

5.2 Performance Metrics and Comparison

We use conventional performance metrics of machine learning for a multi-class classification problem such as Precision, Recall, F1-Score and Accuracy. We compare the performance of MAppGraph with three following techniques.

5.2.1 Multilayer Perceptron (MLP). We aggregate features of graph nodes into a single vector. Due to the heterogeneity in the number of nodes in the graphs, we fixed the number of nodes, which have the highest number of traffic packets to construct the feature vectors. The selected nodes are sorted in the descending order based on the number of traffic packets to construct the feature vector. We define this parameter as N in the experiments presented hereafter.

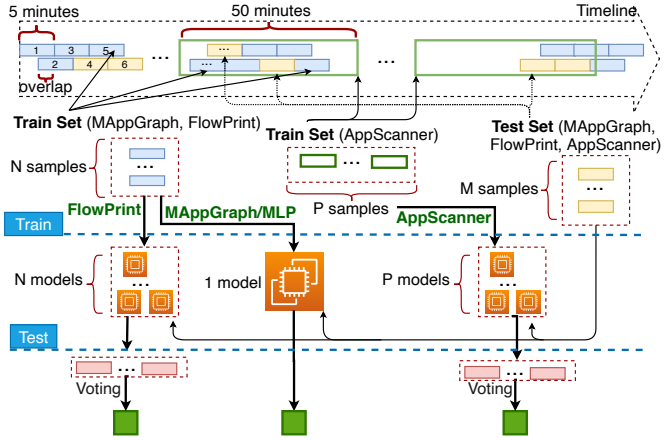


Figure 5: Preparation of Training and Testing Datasets.

As we used only node features, the correlation among nodes is not used in the models. The architecture of MLP consists of 3 hidden layers, each having 1024 hidden nodes. After each dense layer, a batch normalization layer is added to normalize data before feeding to the next dense layer. We used ReLU for MLP.

5.2.2 AppScanner. AppScanner [40] is a flow-based mobile app classification technique, in which a capture of 50 minutes of mobile app traffic is analyzed and flow features are extracted to train a machine learning model (Random Forest and Support Vector Machine) to classify traffic flows of mobile apps. The authors extracted 40 traffic features, which are also used in our work presented in Table 1. For a fair comparison, we faithfully used the source codes of AppScanner downloaded from its website[2] and run with our collected data. As shown in Figure 5, each AppScanner model requires 50 minutes of traffic for each app. For all the apps used in our experiments, every app has at least 16 traffic chunks with a duration of 50 minutes. Thus, we considered 16 AppScanner models to evaluate the performance. Besides presenting the performance of each individual model, we used a naive voting scheme to use all the 16 models for classification. Given a mobile app, the class predicted by the highest number of models is considered as the final prediction of the app. We denote this technique as *Enhanced AppScanner* in the experiments.

5.2.3 FlowPrint. FlowPrint [44] considers the cross-correlation to classify or detect mobile apps. With the traffic collected in a time window (e.g., 5 minutes), instead of using machine learning, FlowPrint defines app fingerprint as "the set of network destinations that form a maximal clique in the correlation graph". Given two fingerprints, the authors use the Jaccard similarity [17] to compare the similarity between them. If the similarity is larger than a predefined threshold then the two fingerprints are considered to belong to the same app. Similar to AppScanner, FlowPrint uses only 5 minutes of traffic capture to create fingerprints for each app. In our experiments, with more than 30 hours of traffic, we can create up to 544 traffic chunks of 5 minutes for each. As shown in Figure 5, we use a naive voting scheme to determine which mobile app a test traffic sample belongs to. Given a test traffic sample, the

[2] AppScanner: https://github.com/Thijsvanede/AppScanner

obtained fingerprints will be compared with all the pre-computed fingerprints obtained from the 544 traffic chunks. The mobile app that has the highest number of pre-computed fingerprints similar to the test fingerprints will be the final prediction. We faithfully used the source codes of FlowPrint downloaded from the website of its authors[3] to run the experiments. We also denoted the adopted FlowPrint as *Enhanced FlowPrint* in the experiments.

For the hyper-parameters of MLP and MAppGraph, we trained the models with 150 training epochs. The initial learning rate is 10^{-4} with a decay of 0.9 after every 10 training epochs. Training the MAppGraph model can be done in an offline manner. In a practical deployment, a pre-trained model is used in the production while another model is trained in parallel to reflect any changes in mobile app behavior (e.g., version upgrading or being attacked). Advanced training methods such as incremental learning [7] can also be applied to reduce the training time of the model when new data is collected. To ensure reproducibility, we conducted each experiment over multiple random-seeded runs. The experiments were carried out on a customized desktop with AMD Ryzen Threadripper 2950X 16-core processor @ 3.5GHz, 64 GB of RAM and 2 Nvidia GeForce RTX 2080Ti GPUs, each having 11 GB of memory.

5.3 Analysis of Results

5.3.1 Overall Performance Comparison. We now present the performance comparison of MAppGraph with MLP, AppScanner and FlowPrint. In Table 2, we present the performance of all the techniques in terms of Precision, Recall, F1-Score and Accuracy on our dataset. The results show that MAppGraph has the best performance. Compared to the worst performance results produced by Enhanced AppScanner, MAppGraph significantly improves the performance in all the metrics by up to 20%. Our experiments also confirm the fact that FlowPrint outperforms AppScanner as discussed in [44]. Interestingly, MLP has a better performance compared to Enhanced AppScanner. This demonstrates that using flow-based detection or classification of mobile apps is not an appropriate approach as most of the apps nowadays share the same third-party services. This makes traffic flows between the mobile apps and the servers of the third-party services have similar behavior, thus being indistinguishable among the apps. Even though MLP does not process graphs with nodes and edges among nodes, the way we select the graph nodes whose features are used to train the model implicitly takes into account the communication correlation of the mobile app and various third-party services used by the app.

It is interesting to show that the result obtained with Enhanced AppScanner based on the voting scheme is much better compared to the performance of individual models. Table 3 presents the performance of AppScanner obtained with individual models. The experimental results show that individual models of AppScanner have stable performance (i.e., testified by the low standard deviation of all performance metrics) throughout different traffic chunks used to train the models. Enhanced AppScanner significantly improves the performance by up to 25% and 20% compared to the worst and best individual models, respectively. This demonstrates the diversity of traffic behavior of mobile apps when users use various functionalities, which pose challenges for the detection and classification techniques.

[3]FlowPrint: https://github.com/Thijsvanede/FlowPrint

Table 2: Overall Performance Comparison

Technique	Precision	Recall	F1-Score	Accuracy
MLP	0.9081	0.9075	0.9074	0.9075
Enhanced AppScanner	0.8634	0.7938	0.7828	0.7938
Enhanced FlowPrint	0.8759	0.8341	0.8275	0.8341
MAppGraph	**0.9364**	**0.9346**	**0.9347**	**0.9346**

Table 3: AppScanner Performance with Individual Models

Model No.	Precision	Recall	F1-Score	Accuracy
1	0.7527	0.6660	0.6468	0.6660
2	0.7476	0.6483	0.6304	0.6483
3	0.7791	0.6757	0.6556	0.6757
4	0.7405	0.6468	0.6299	0.6468
5	0.7779	0.6577	0.6460	0.6577
6	0.7358	0.6652	0.6479	0.6652
7	0.7281	0.6493	0.6308	0.6493
8	0.7354	0.6559	0.6398	0.6559
9	0.7322	0.6485	0.6323	0.6485
10	0.7590	0.6515	0.6381	0.6515
11	0.7666	0.6515	0.6418	0.6515
12	0.7306	0.6527	0.6419	0.6527
13	0.7873	0.6563	0.6571	0.6563
14	0.7618	0.6415	0.6357	0.6415
15	0.7420	0.6346	0.6258	0.6346
16	0.7592	0.6633	0.6534	0.6633
Mean	0.7522	0.6541	0.6408	0.6541
Stan. Dev.	0.0187	0.0101	0.0097	0.0101
Enhanced AppScanner	**0.8634**	**0.7938**	**0.7828**	**0.7938**

Compared to Enhanced FlowPrint which also considers the cross-correlation among app servers and third-party services by constructing communication graphs, MAppGraph improves the performance by up to 7%. This improvement is a result of the combination of advanced deep learning techniques and consideration of the diverse behavior of mobile apps. On one hand, using DGCNN (with multiple graph convolution layers) allows the classification model to learn the communication behavior of mobile apps better from the graph topology and node attributes. On the other hand, MAppGraph takes into account the diversity of mobile app behavior by training a single DGCNN model on multiple graphs. This is an advantage of our technique compared to FlowPrint, which has to compare the fingerprints obtained from a test traffic sample with all pre-computed fingerprints (there are at least 544×101 fingerprints computed by FlowPrint in our experiments). This fingerprint comparison technique is not practical as there is a large number of mobile installed by users in reality. Nevertheless, in Figure 6, we present the performance of Enhanced FlowPrint with respect to the number of traffic chunks (of 5 minutes) of each app used in the voting scheme. The results show that the performance improvement is significant when we increase the number of traffic chunks of each app used for inference from 1 to 20. However, using

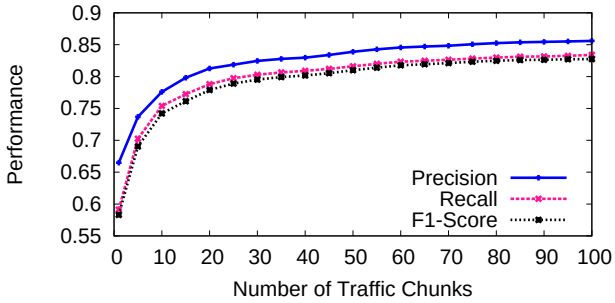


Figure 6: Performance of Enhanced FlowPrint w.r.t Number of Traffic Chunks of Each App Used for Classification.

more than 20 traffic chunks of each app only results in a slight performance improvement that may not be worth compensating for the inference time/cost. It is worth mentioning that inference with MAppGraph is performed by simply applying linear operations (i.e., matrix multiplication) of graphs (i.e., node attributes and edge weights) with the parameters of a single DGCNN model.

5.3.2 Impact of Number of Graph Nodes used to Train Models. As discussed in Section 3, the pooling layers perform truncation or extension of the graph latent representation to predefined size (k). This implicitly corresponds to the number of graph nodes (denoted as N) whose features are used to train the model in case of MLP (i.e., k is a multiple of N). The rationale behind this experiment is that with a high value of N, all the graphs that have fewer nodes must use zero paddings for feature vectors (in the case of MLP) or the latent representation vectors (in case of MAppGraph). These zero-valued features may mislead the learning of the models. On the other hand, if we use a small number of nodes, we may lose useful information from the nodes that are discarded, thus affecting the performance of the models as well. In Figure 7, we present the histogram of the number of nodes in the graphs of our dataset with T_{window} set to 5 minutes. The histogram shows that most of the graphs have around 10 nodes. 90% of graphs have fewer than 35 nodes and 86% of graphs have fewer than 30 nodes. In Figure 8, we present the performance of the models with respect to four values of N. We obtained expected results that performance degrades when fewer nodes are used. The performance increases to an optimum value of N before decreasing again when a large number of nodes are used. It is interesting mentioning that the optimal value of N is different between MLP and MAppGraph. The reason could be the fact that MAppGraph needs more information about the graph topology to differentiate mobile apps. Nevertheless, in all the experimental scenarios, we observed that MAppGraph has a better performance compared to MLP. We note that FlowPrint considers the entire graphs for determining app fingerprints. Thus, we do not present FlowPrint in this experiment.

5.3.3 Impact of Time Window Duration of Traffic Collection for Graph Construction. In this experiment, we evaluate the impact of the time window (T_{window}) duration needed for traffic capture to construct the communication graphs of mobile apps. In the experiments presented above, we used a time window of 5 minutes for traffic capture. However, it would be better if the model can classify the apps with shorter traffic capture, leading to better benefits such

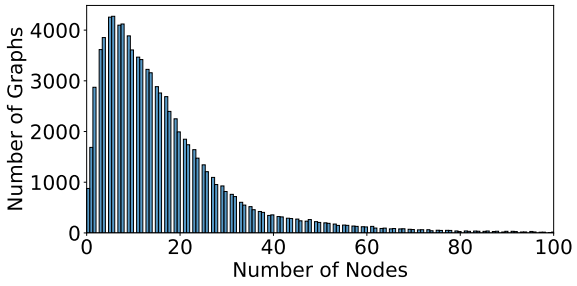
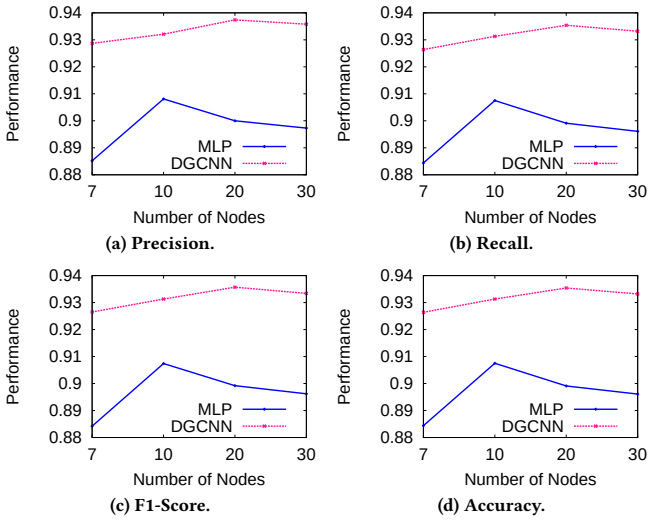


Figure 7: Histogram of Number of Nodes in Graphs.



(a) Precision. **(b) Recall.** **(c) F1-Score.** **(d) Accuracy.**

Figure 8: Performance of MLP and MAppGraph w.r.t. Number of Nodes in Graphs.

as lower computational resources required, a quick reaction in case of security breaches. In Table 4, we present the performance of the proposed technique with respect to the duration of the time window required to capture traffic.

As expected, the performance of all the techniques decreases when we use a shorter traffic capture window. The results show that when we reduce the traffic capture from 5 minutes to 1 minute, the performance of MAppGraph decreases by 7%. We believe that the gain obtained when reducing the traffic capture duration (e.g., faster app classification and detection, less storage and computational resources required) is more significant compared to the loss in performance. In practice, this parameter can be configured by the network operators based on their desired performance and objective. Nevertheless, the trends of performance among the techniques do not change such that the proposed technique (MAppGraph) always performs the best followed by MLP and Enhanced FlowPrint.

It is to be noted that when the time window duration is short (the cases of $T_{\text{window}} \leqslant 2$), we do not apply overlapping. The rational behind is twofold. First, with short capture duration, we managed to generate sufficient data to split into train and test for performance evaluation. Second, it is practically fast enough to detect the applications. In case where the capture duration is long (e.g., 5

Table 4: Impact of Time Window (in minutes) of Traffic Collection on Performance of Classification Models

T_{window}	Technique	Precision	Recall	F1-Score	Accuracy
5	MLP	0.9081	0.9075	0.9074	0.9075
	Enhanced FlowPrint	0.8759	0.8341	0.8275	0.8341
	MAppGraph	**0.9364**	**0.9346**	**0.9347**	**0.9346**
4	MLP	0.8905	0.8894	0.8896	0.8894
	Enhanced FlowPrint	0.8559	0.8296	0.8258	0.8296
	MAppGraph	**0.9181**	**0.9174**	**0.9171**	**0.9174**
3	MLP	0.8705	0.8671	0.8681	0.8671
	Enhanced FlowPrint	0.8546	0.8175	0.8179	0.8175
	MAppGraph	**0.8977**	**0.8932**	**0.8935**	**0.8932**
2	MLP	0.8648	0.8625	0.8631	0.8625
	Enhanced FlowPrint	0.8447	0.7945	0.7978	0.7945
	MAppGraph	**0.8759**	**0.8738**	**0.8734**	**0.8738**
1	MLP	0.8478	0.847	0.8466	0.8470
	Enhanced FlowPrint	0.7785	0.6535	0.6711	0.6535
	MAppGraph	**0.8698**	**0.8683**	**0.8679**	**0.8683**

Table 5: Impact of Slice Duration on Performance of Classification Models

t_{slice}	Technique	Precision	Recall	F1-Score	Accuracy
1	Enhanced FlowPrint	0.7773	0.7325	0.7176	0.7325
	MAppGraph	**0.9342**	**0.9327**	**0.9328**	**0.9327**
5	Enhanced FlowPrint	0.8414	0.8112	0.8065	0.8112
	MAppGraph	**0.9353**	**0.9335**	**0.9337**	**0.9335**
10	Enhanced FlowPrint	0.8759	0.8341	0.8275	0.8341
	MAppGraph	**0.9364**	**0.9346**	**0.9347**	**0.9346**

minutes), sliding windows will increase the frequency of classification and detection. For instance, with sliding windows of 2 minutes and $T_{\text{window}} = 5$ minutes, within 10 minutes, 3 samples will be collected and analyzed instead of 2 samples in case of non-overlapping. This could help users (e.g., network administrator) who deploy the proposed approach, quickly detect and class the apps.

5.3.4 Impact of Slice Duration on Cross-Correlation in Graph Construction. In this experiment, we evaluate the impact of the slice duration that is used to determine the communication correlation among destination services connected by a mobile app. The shorter the slice duration, the fewer the edges in the graphs. On the other hand, the longer the slice duration, the graphs become fully connected. In both scenarios, the communication correlation may affect the capability of MAppGraph in learning the communication behavior of mobile apps. In Table 5, we present the performance of the techniques with different values of slice duration. It is to be noted that the slice duration is used to compute the weight of graph edges. Thus, it does not affect the performance of MLP that only uses traffic features of the nodes in the graphs (i.e., the destination services that mobile apps connect to). Thus, we do not show MLP in this experiment. Interestingly, the results show that the performance increases along with the increase in the slice duration. While FlowPrint incurs a big performance gap (up to 13%) between the two scenarios: $t_{\text{slice}} = 1$ second and $t_{\text{slice}} = 10$ seconds, this does not happen to MAppGraph. The difference between the two scenarios is less than 1%. This relieves the effort for determining optimal slice duration to obtain the best performance.

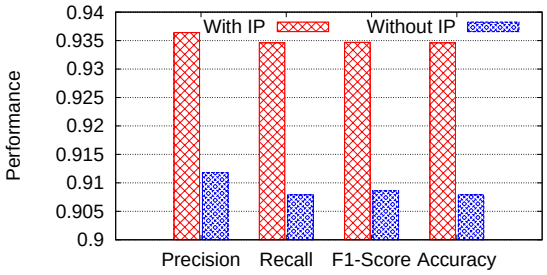


Figure 9: Performance of MAppGraph with and without using IP Addresses in Feature Vectors.

5.3.5 Performance with and without using the IP Addresses in Feature Vectors. As we discussed previously, IP addresses of destination services may change due to load balancing. In this experiment, we train MAppGraph without using the IP addresses of destination services in the feature vectors. In Figure 9, we present the performance comparison of MAppGraph with and without IP addresses in feature vectors. The results show that the performance slightly decreases when IP addresses are not used to train MAppGraph. However, we believe that this performance is acceptable as the model does not need to be retrained when deploying in a different network domain of the destination services. It is worth mentioning that the performance of MAppGraph without using IP addresses as a feature still significantly better than that of FlowPrint, thus demonstrating the effectiveness of the proposed technique.

5.3.6 Classification of Mobile Apps with Similar Functionalities. In this experiment, we evaluate the performance of MAppGraph when apps have similar functionalities. We created 2 datasets, each having 17 apps. The first dataset includes 17 apps related to audio and music players such as Spotify and SoundCloud, which should have similar traffic features such as packet size, flow size, etc. The second dataset includes the apps with different functionalities. We trained two DGCNN models and tested them on the two datasets, denoted as SIM-APP and DIFF-APP, respectively. In Figure 10, we present the performance of these two models. As expected, SIM-APP has lower performance compared to DIFF-APP. The model (DIFF-APP) trained on the dataset with different functionalities attains 0.9750 for all performance metrics. However, the performance degradation when similar apps are present in the dataset is not significant (i.e., 4%). This shows that considering cross-correlation among the services used by the apps into graphs and combining with traffic features (i.e., information extracted from packet headers) as attributes of graph nodes allow us to accurately differentiate mobile apps even though they have similar functionalities.

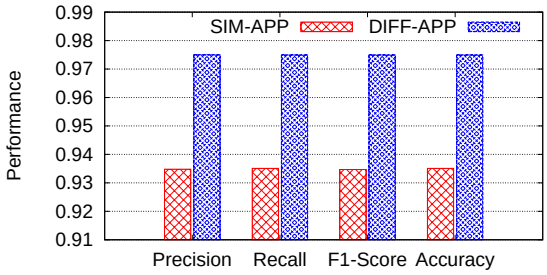
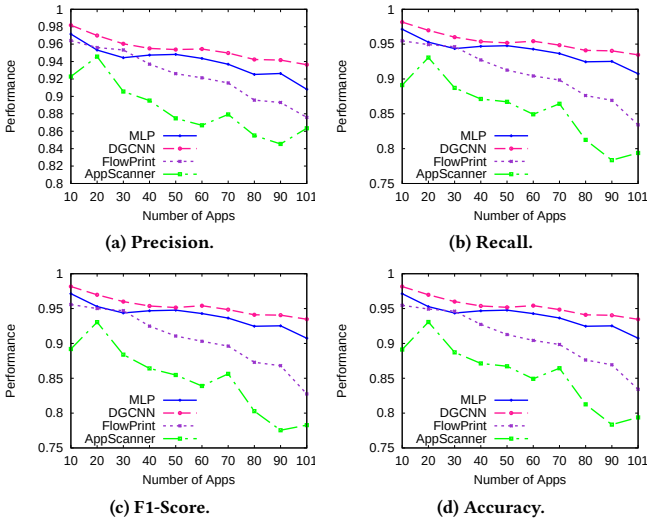


Figure 10: Classification of Apps with Similar and Different Functionalities.



(a) Precision. **(b) Recall.**

(c) F1-Score. **(d) Accuracy.**

Figure 11: Performance of Techniques w.r.t Number of Apps.

5.3.7 Performance with Different Number of Apps. We evaluate the performance of MAppGraph with different number of apps. We randomly selected a number of apps from the original dataset to train and test the model. The results presented in Figure 11 show that the performance of the techniques degrades along with the increase in the number of apps. This is expected as the higher the number of apps, the higher the possibility that the more apps have similar behavior. The results also indicate that the performance of MLP, AppScanner and FlowPrint degrades quickly, resulting in a big margin between the two scenarios (i.e., the smallest and highest number of apps). Whereas, the performance of the DGCNN model slightly decreases and still achieves high performance even with the highest number of apps in the dataset. This demonstrates the effectiveness of MAppGraph.

5.3.8 Discussion on Detection of Unseen/unidentified Apps. While the proposed technique mainly applies to the problem of mobile app classification, it can also be adopted for the problem of detection of unseen or unidentified apps. Note that the output of the softmax layer (i.e., the output layer of the DGCNN architecture) for class decision is a probability and the class with the highest probability is selected. This probability is considered as the confidence of the model to decide whether the traffic sample belongs to the selected app. We can compare the confidence with a predefined threshold (e.g., 0.5). If the probability is smaller than the threshold, we can confirm that the app is unseen or unidentified. A more advanced approach that uses an unsupervised learning approach such as graph clustering could also be employed. Such an unsupervised learning approach does not need a large labelled dataset, thus relieving us from data labelling effort. We make this as our future work.

6 CONCLUSION

In this paper, we presented MAppGraph, a novel technique for mobile app classification that can deal with encrypted traffic, dynamic communication behavior and implementation nature of mobile apps. We developed a technique to process mobile traffic and construct communication behavior graphs that considers the cross-correlation among the services connected by the apps and traffic features, which are useful for differentiating mobile apps. We developed a DGCNN model that is able to learn the diverse communication behavior of mobile apps from a large number of graphs. We collected traffic for 101 Android apps, each with more than 30 hours of traffic for the experiments. We carried out extensive experiments with various scenarios and compared the performance of MAppGraph with a traditional deep learning model (MLP) and two state-of-the-art techniques (AppScanner and FlowPrint). The experimental results show that MAppGraph outperforms the baseline techniques with a performance improvement of up to 20% in terms of Precision, Recall, F1-Score and Accuracy. With high performance and fast execution, MAppGraph enables better mobile security by using it in anomaly detection, automated vulnerability patching of mobile apps, etc. as well as in network management such as dynamic resource allocation and traffic engineering.

Acknowledgments. This work was partially supported by Tan Tao University Foundation for Science and Technology Development under the Grant No. TTU.RS.19.102.023.

REFERENCES

[1] Khaled Al-Naami et al. 2016. Adaptive Encrypted Traffic Fingerprinting with Bi-Directional Dependence. In *Proc. 32nd Annual Conference on Computer Security Applications (ACSAC '16)*. Los Angeles, CA, USA, 177–188.
[2] Blake Anderson et al. 2018. Deciphering malware's use of TLS (without decryption). *J Comput Virol Hack Tech* 14 (Aug. 2018).
[3] Blake Anderson and David McGrew. 2016. Identifying Encrypted Malware Traffic with Contextual Flow Data. In *2016 ACM Workshop on Artificial Intelligence and Security*. Vienna, Austria, 35–46.
[4] Noah J. Apthorpe, Dillon Reisman, Srikanth Sundaresan, Arvind Narayanan, and Nick Feamster. 2017. Spying on the Smart Home: Privacy Attacks and Defenses on Encrypted IoT Traffic. *CoRR* abs/1708.05044 (2017).
[5] Bram Bonne. 2021. An Update on Android TLS Adoption. https://android-developers.googleblog.com/2019/12/an-update-on-android-tls-adoption.html. Online; accessed 30 April 2021.
[6] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann Lecun. 2014. Spectral networks and locally connected networks on graphs. In *International Conference on Learning Representations (ICLR2014)*. Banff, Canada.
[7] Francisco M. Castro, Manuel J. Marín-Jiménez, Nicolás Guil, Cordelia Schmid, and Karteek Alahari. 2018. End-to-End Incremental Learning. In *15th European Conference on Computer Vision (ECCV 2018)*. Munich, Germany, 241–257.
[8] Fenxiao Chen, Yun-Cheng Wang, Bin Wang, and C.-C. Jay Kuo. 2020. Graph representation learning: a survey. *APSIPA Transactions on Signal and Information Processing* 9 (2020), e15. https://doi.org/10.1017/ATSIP.2020.13
[9] Yi Chen, Wei You, Yeonjoon Lee, Kai Chen, XiaoFeng Wang, and Wei Zou. 2017. Mass Discovery of Android Traffic Imprints through Instantiated Partial Execution. In *Proc. 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Dallas, Texas, USA, 815–828.

[10] Z. Chen, K. He, J. Li, and Y. Geng. 2017. Seq2Img: A sequence-to-image based approach towards IP traffic classification using convolutional neural networks. In *2017 IEEE International Conference on Big Data*. Boston, MA, USA, 1271–1276.

[11] Zhengyang Chen, Bowen Yu, Yu Zhang, Jianzhong Zhang, and Jingdong Xu. 2016. Automatic Mobile Application Traffic Identification by Convolutional Neural Networks. In *2016 IEEE Trustcom/BigDataSE/ISPA*. Tianjin, China, 301–307.

[12] Yeongrak Choi, Jae Yoon Chung, Byungchul Park, and James Won-Ki Hong. 2012. Automated Classifier Generation for Application-level Mobile Traffic Identification. In *2012 IEEE Network Operations and Management Symposium*. Maui, HI, USA, 1075–1081.

[13] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. 2013. NetworkProfiler: Towards automatic fingerprinting of Android apps. In *2013 Proceedings IEEE INFOCOM*. Turin, Italy, 809–817.

[14] Manh Tuan Do, Noseong Park, and Kijung Shin. 2020. Two-stage Training of Graph Neural Networks for Graph Classification. *arXiv e-prints* (Nov. 2020).

[15] David Duvenaud, , Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gomez-Bombarelli, Timothy Hirzel, Alan Aspuru-Guzik, and Ryan P. Adams. 2015. Convolutional Networks on Graphs for Learning Molecular Fingerprints. In *Twenty-ninth Conference on Neural Information Processing Systems*. Montreal, Canada.

[16] A. S. Iliyasu and H. Deng. 2020. Semi-Supervised Encrypted Traffic Classification With Deep Convolutional Generative Adversarial Networks. *IEEE Access* 8 (2020).

[17] Paul Jaccard. 1912. The Distribution of the Flora in the Alpine Zone. *New Phytologist* 11, 2 (Feb. 1912).

[18] Peter Jonsson, Stephen Carson, Jasmeet Singh Sethi, Mats Arvedson, Ritva Svenningsson, Per Lindberg, Kati Öhman, and Patrik Hedlund. 2017. *Ericsson Mobility Report.* Technical Report. Ericsson.

[19] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations (ICLR 2015)*. San Diego, CA, USA.

[20] Baris Kurt, Engin Zeydan, Utku Yabas, Ilyas Alper Karatepe, Gunes Karabulut Kurt, and Ali Taylan Cemgil. 2016. A Network Monitoring System for High Speed Network Traffic. In *2016 13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. London, UK.

[21] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. 2019. Self-Attention Graph Pooling. In *Proc. International Conference on Machine Learning*. Long Beach, USA.

[22] Jingyi Liao, Sin G. Teo, Partha Pratim Kundu, and Tram Truong-Huu. 2021. ENAD: An Ensemble Framework for Unsupervised Network Anomaly Detection. In *Proc. IEEE CSR 2021*. Virtual Conference.

[23] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. 2014. ANDRUBIS – 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. Wroclaw, Poland, 3–17.

[24] M. Lopez-Martin, B. Carro, A. Sanchez-Esguevillas, and J. Lloret. 2017. Network Traffic Classifier With Convolutional and Recurrent Neural Networks for Internet of Things. *IEEE Access* 5 (2017), 18042–18050.

[25] Mohammad Lotfollahi, Mahdi Jafari Siavoshani, Ramin Shirali Hossein Zade, and Mohammdsadegh Saberian. 2020. Deep packet: a novel approach for encrypted traffic classification using deep learning. *Soft Computing* 24 (Feb. 2020).

[26] Yair Meidan, Michael Bohadana, Asaf Shabtai, Juan David Guarnizo, Martín Ochoa, Nils Ole Tippenhauer, and Yuval Elovici. 2017. ProfilIoT: A Machine Learning Approach for IoT Device Identification Based on Network Traffic Analysis. In *Proc. Symposium on Applied Computing (SAC '17)*. Marrakech, Morocco.

[27] Markus Miettinen, Samuel Marchal, Ibbad Hafeez, N. Asokan, Ahmad-Reza Sadeghi, and Sasu Tarkoma. 2017. IoT SENTINEL: Automated Device-Type Identification for Security Enforcement in IoT. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. Atlanta, GA, USA.

[28] Akash Raj Narayanadoss, Tram Truong-Huu, Purnima Murali Mohan, and Mohan Gurusamy. 2019. Crossfire Attack Detection Using Deep Learning in Software Defined ITS Networks. In *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*. Kuala Lumpur, Malaysia.

[29] T. T. T. Nguyen and G. Armitage. 2008. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials* 10, 4 (2008), 56–76.

[30] Sinno Jialin Pan and Qiang Yang. 2010. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering* 22, 10 (2010), 1345–1359.

[31] Emanuele Petagna, Giuseppe Laurenza, Claudio Ciccotelli, and Leonardo Querzoni. 2019. Peel the Onion: Recognition of Android Apps Behind the Tor Network. In *Proc. International Conference on Information Security Practice and Experience*. Kuala Lumpur, Malaysia, 95–112.

[32] Lawrence R. Rabiner and Bernard Gold. 1975. *Theory and Application of Digital Signal Processing*. Prentice Hall, Hoboken, New Jersey, United States.

[33] Jingjing Ren et al. 2019. An International View of Privacy Risks for Mobile Apps. https://recon.meddle.mobi/papers/cross-market.pdf

[34] Jingjing Ren, Martina Lindorfer, Daniel Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. 2018. Bug fixes, improvements,... and privacy leaks–a longitudinal study of pii leaks across android app versions. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. San Diego, USA.

[35] S. Rezaei and X. Liu. 2019. Deep Learning for Encrypted Traffic Classification: An Overview. *IEEE Communications Magazine* 57, 5 (2019), 76–81.

[36] Shahbaz Rezaei and Xin Liu. 2019. How to Achieve High Classification Accuracy with Just a Few Labels: A Semisupervised Approach Using Sampled Packets. In *Proc. 19th Industrial Conference on Data Mining*. New York, USA, 28–42.

[37] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *European Semantic Web Conference*. Heraklion, Crete, Greece.

[38] Yaman Sharaf-Dabbagh and Walid Saad. 2016. On the Authentication of Devices in the Internet of Things. In *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. Coimbra, Portugal.

[39] Hongtao Shi, Hongping Li, Dan Zhang, Chaqiu Cheng, and Xuanxuan Cao. 2018. An efficient feature generation approach based on deep learning and feature selection techniques for traffic classification. *Computer Networks* 132 (2018).

[40] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic. 2016. AppScanner: Automatic Fingerprinting of Smartphone Apps from Encrypted Network Traffic. In *IEEE EuroS&P 2016*. Saarbruecken, Germany.

[41] Vincent F. Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. 2018. Robust Smartphone App Identification via Encrypted Network Traffic Analysis. *IEEE Transactions on Information Forensics and Security* 13, 1 (2018), 63–78.

[42] Vijayanand Thangavelu, Dinil Mon Divakaran, Rishi Sairam, Suman Sankar Bhunia, and Mohan Gurusamy. 2019. DEFT: A Distributed IoT Fingerprinting Technique. *IEEE Internet of Things Journal* 6, 1 (2019), 940–952.

[43] Tram Truong-Huu, Nidhya Dheenadhayalan, Partha Pratim Kundu, Vasudha Ramnath, Jingyi Liao, Sin G. Teo, and Sai Praveen Kadiyala. 2020. An Empirical Study on Unsupervised Network Anomaly Detection Using Generative Adversarial Networks. In *1st Security and Privacy on Artificial Intelligent Workshop (SPAI'20)*. Taipei, Taiwan.

[44] Thijs van Ede, Riccardo Bortolameotti, Andrea Continella, Jingjing Ren, Daniel J Dubois, Martina Lindorfer, David Choffnes, Maarten van Steen, and Andreas Peter. 2020. FlowPrint: Semi-Supervised Mobile-App Fingerprinting on Encrypted Network Traffic. In *Proc. Network and Distributed System Security Symposium (NDSS 2020)*. San Diego, CA, USA.

[45] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. Vancouver, Canada.

[46] W. Wang, M. Zhu, J. Wang, X. Zeng, and Z. Yang. 2017. End-to-end encrypted traffic classification with one-dimensional convolution neural networks. In *IEEE ISI 2017*. Beijing, China, 43–48.

[47] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*. New Orleans, Louisiana, United States.

[48] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. 2018. Hierarchical Graph Representation Learning with Differentiable Pooling. In *32nd International Conference on Neural Information Processing Systems*. Montréal, Canada, 4805–4815.

[49] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An End-to-End Deep Learning Architecture for Graph Classification. In *The Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*. New Orleans, USA.

[50] Yixue Zhao and Nenad Medvidovic. 2019. A Microservice Architecture for Online Mobile App Optimization. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. Montreal, QC, Canada.

A DATA COLLECTION AND ETHICS

Our data collection process involved a group of 10 students including the first two co-authors of the paper under the supervision of the 4^{th} co-author. We set up controlled experimental sessions, in each session (around 3-4 hours), the volunteering students (the number of students who joined in each session varies) get a smartphone provided by the research team and access one of the apps presented in Appendix B. The students were offered beverage and food during the experimental sessions. After each session, the research team took back the devices, extracted the logs and anonymized the data. We also used the mobile devices purchased in the scope of our project to perform traffic collection rather than using personal devices of the students. We also did not store personal information of students. Thus, the data collection team did not involve the institutional review board (IRB).

B LIST OF MOBILE APPS

Table 6: List of Mobile Apps

No.	Package Name	Number of Installs	No.	Package Name	Number of Installs
1	com.facebook.katana	>5 billion	52	com.sendo	>10 million
2	com.instagram.android	>1 billion	53	com.shopee.vn	>10 million
3	com.facebook.orca	>1 billion	54	com.ted.android	>10 million
4	com.netflix.mediaclient	>1 billion	55	gsn.game.zingplaynew1	>10 million
5	com.skype.raider	>1 billion	56	vn.tiki.app.tikiandroid	>10 million
6	com.snapchat.android	>1 billion	57	vn.vtv.vtvgo	>10 million
7	com.twitter.android	>1 billion	58	droidhang.twgame.restaurant	>5 million
8	com.dts.freefireth	>500 million	59	game.bida.vng	>5 million
9	video.like	>500 million	60	com.chotot.vn	>5 million
10	com.linkedin.android	>500 million	61	com.imbb.oversea.android	>5 million
11	com.pinterest	>500 million	62	com.guardian	>5 million
12	com.spotify.music	>500 million	63	mobi.fiveplay.tinmoi24h	>5 million
13	org.telegram.messenger	>500 million	64	com.hihuc.cchess.online	>1 million
14	us.zoom.videomeetings	>500 million	65	vcc.mobilenewsreader.kenh14	>1 million
15	com.innersloth.spacemafia	>100 million	66	myradio.radio.fmradio.liveradio.radiostation	>1 million
16	com.azarlive.android	>100 million	67	mobi.mangatoon.novel.portuguese	>1 million
17	sg.bigo.live	>100 million	68	com.live.party	>1 million
18	com.mobilemotion.dubsmash	>100 million	69	com.popsworldwide.popskids	>1 million
19	com.google.android.apps.meetings	>100 million	70	com.giaitri.tvviet	>1 million
20	com.yy.hiyo	>100 million	71	com.riotgames.league.wildriftvn	>1 million
21	com.lazada.android	>100 million	72	vieon.phim.tv	>1 million
22	com.soundcloud.android	>100 million	73	fr.playsoft.vnexpress	>1 million
23	com.starmakerinteractive.starmaker	>100 million	74	wsj.reader_sp	>1 million
24	com.sgiggle.production	>100 million	75	com.tencent.qqlivei18n.tw	>1 million
25	com.ss.android.ugc.trill	>100 million	76	com.toast.comico.vn	>0.5 million
26	com.tinder	>100 million	77	com.hahalolo.android.social	>0.5 million
27	tv.twitch.android.app	>100 million	78	com.vivavietnam.lotus	>0.5 million
28	com.zing.zalo	>100 million	79	xyz.wmfall.phim	>0.5 million
29	me.mycake	>50 million	80	com.viettel.tv360	>0.5 million
30	com.garena.game.kgvn	>50 million	81	vn.com.dantrinews.android	>0.1 million
31	com.huya.nimo	>50 million	82	io.pobble.sen.android	>0.1 million
32	com.nono.android	>50 million	83	com.music.mp3.trutinh.nhacvang	>0.1 million
33	com.reddit.frontpage	>50 million	84	app.sachnoi	>0.1 million
34	com.radio.fmradio	>50 million	85	vcc.mobilenewsreader.sohanews	>0.1 million
35	com.tencent.wesing	>50 million	86	com.gkim.thanhniennews	>0.1 million
36	org.wikipedia	>50 million	87	tivi.vina.thecomics	>0.1 million
37	com.zing.mp3	>50 million	88	com.tvonline.tivi24h	>0.1 million
38	com.baohay24h.app	>10 million	89	com.topcv	>0.1 million
39	com.epi	>10 million	90	vn.tuoitre.app	>0.1 million
40	bbc.mobile.news.ww	>10 million	91	com.vietnamworks.vietnamworks	>0.1 million
41	com.chess	>10 million	92	com.wewe.musicsounds	>0.1 million
42	com.cnn.mobile.android.phone	>10 million	93	com.miboo	>0.1 million
43	com.fplay.activity	>10 million	94	com.kaka.kakavideo	>0.05 million
44	sg.bigo.hellotalk	>10 million	95	com.habn.webtruyen	>0.05 million
45	com.iqiyi.i18n	>10 million	96	com.radione	>0.05 million
46	mobi.mangatoon.comics.aphone	>10 million	97	vn.diijam.jammer	>0.01 million
47	com.vng.mlbbvn	>10 million	98	com.craftbox.jwapp.android	>0.01 million
48	ht.nct	>10 million	99	smartapphome.tinhte	>0.01 million
49	fm.castbox.audiobook.radio.podcast	>10 million	100	com.bachtruyen	>0.01 million
50	com.vng.pubgmobile	>10 million	101	com.msa.audiobooks	>0.005 million
51	com.quora.android	>10 million			