Accurate Decentralized Application Identification via Encrypted Traffic Analysis Using Graph Neural Networks

Meng Shen, *Member, IEEE*, Jinpeng Zhang, Liehuang Zhu, *Member, IEEE*, Ke Xu, *Senior Member, IEEE*, and Xiaojiang Du, *Fellow, IEEE*

***Abstract*— Decentralized Applications (DApps) are increasingly developed and deployed on blockchain platforms such as Ethereum. DApp fingerprinting can identify users' visits to specific DApps by analyzing the resulting network traffic, revealing much sensitive information about the users, such as their real identities, financial conditions and religious or political preferences. DApps deployed on the same platform usually adopt the same communication interface and similar traffic encryption settings, making the resulting traffic less discriminative. Existing encrypted traffic classification methods either require handcrafted and fine-tuning features or suffer from low accuracy. It remains a challenging task to conduct DApp fingerprinting in an accurate and efficient way. In this paper, we present GraphDApp, a novel DApp fingerprinting method using Graph Neural Networks (GNNs). We propose a graph structure named Traffic Interaction Graph (TIG) as an information-rich representation of encrypted DApp flows, which implicitly reserves multiple dimensional features in bidirectional client-server interactions. Using TIG, we turn DApp fingerprinting into a graph classification problem and design a powerful GNN-based classifier. We collect real-world traffic datasets from 1,300 DApps with more than 169,000 flows. The experimental results show that GraphDApp is superior to the other state-of-the-art methods in terms of classification accuracy in both closed- and open-world scenarios. In addition, GraphDApp maintains its high accuracy when being applied to the traditional mobile application classification.**

***Index Terms*— Decentralized applications, encrypted traffic classification, deep learning, graph neural networks, blockchain.**

Manuscript received June 24, 2020; revised September 30, 2020 and December 11, 2020; accepted January 4, 2021. Date of publication January 11, 2021; date of current version February 12, 2021. This work was supported in part by the National Key Research and Development Program of China under Grant 2020YFB1006101, in part by the Beijing Nova Program under Grant Z201100006820006, in part by the NSFC Projects under Grant 61972039, Grant 61932016, and Grant 61872041, in part by the Beijing Natural Science Foundation under Grant 4192050, in part by the Zhejiang Lab Open Fund under Grant 2020AA3AB04, in part by the China National Funds for Distinguished Young Scientists under Grant 61825204, and in part by the Beijing Outstanding Young Scientist Program under Grant BJJWZYJH01201910003011. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Issa Traore. *(Corresponding author: Liehuang Zhu.)*

Meng Shen is with the School of Cyberspace Security, Beijing Institute of Technology, Beijing 100081, China, and also with the Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: shenmeng@bit.edu.cn).

Jinpeng Zhang is with the School of Computer Science, Beijing Institute of Technology, Beijing 100081, China (e-mail: zhangjinpeng@bit.edu.cn).

Liehuang Zhu is with the School of Cyberspace Security, Beijing Institute of Technology, Beijing 100081, China (e-mail: liehuangz@bit.edu.cn).

Ke Xu is with the Department of Computer Science, Tsinghua University, Beijing 100084, China, also with the Beijing National Research Center for Information Science and Technology (BNRist), Beijing 100084, China, and also with the Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: xuke@mail.tsinghua.edu.cn).

Xiaojiang Du is with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122 USA (e-mail: dxj@ieee.org).

Digital Object Identifier 10.1109/TIFS.2021.3050608

I. Introduction

With the recent development of blockchain technology, the number of decentralized applications (DApps) are increasing dramatically. Unlike regular mobile or web applications deployed on centralized servers, DApps run their backend codes on a decentralized peer-to-peer (P2P) network without the control of a single entity. Among the blockchain platforms on which DApps are hosted, such as EOS, NEO, Stellar, and Tron, we focus on Ethereum [1] in this paper, as it attracts the largest developer community, where more than 3,200 DApps are deployed on Ethereum and the number of daily active users has reached nearly 110,000 by 2020 [2].

The prosperity of DApps is attributed greatly to their resistance to censorship [3], [6]. Compared with the traditional mobile or web applications, DApps are completely open-sourced and autonomously managed without a single authority to manage all codes and data. Once a piece of information in DApp is added to the underlying blockchain, it gets stored permanently and thus cannot be removed or modified. In addition, blockchain technology naturally provides anonymity to each participant and thus can potentially protect identity privacy of DApp users. These features make DApps trustable to censorship and governance.

Although these service-level enchantments provide users with safety and security, the network traffic generated when individual users visit DApps can still reveal plenty of sensitive information of users. Passive adversaries (e.g., campus network administrators, residential network service providers, or malicious eavesdroppers) could conduct *DApp fingerprinting* to identify the specific DApps that a user visits by analyzing the resulting network traffic. An adversary can infer users' financial conditions from their usage of gambling DApps or learn their religious preferences and political views from their visits to social DApps. DApp fingerprinting can also be conducted by governors to deanonymize DApp users or even block access attempts to certain DApps without affecting visits to the rest DApps on the same platform (e.g., Ethereum).

A question naturally arises in this scenario is *to what extent DApps can be identified from the traffic generated by user visits*. Due to the adoption of SSL/TLS protocols in DApps, traditional traffic classification methods (e.g., Deep Packet Inspection) lose efficacy. Encrypted traffic analysis is not a new research area and a number of methods have been proposed for website fingerprinting [18], [26], mobile application fingerprinting [27], and user action identification [9], but few efforts have been made on DApp fingerprinting.

It remains a challenging task to *accurately* and *efficiently* identify DApps via traffic analysis. Unlike regular mobile applications or websites, DApps deployed on Ethereum implement the same frontend interface, adopt similar settings of SSL/TLS protocols, and share the same decentralized blockchain network for running their backend codes and managing their data. As a result, the traffic of different DApps has lots of common features, leading to low accuracy of existing fingerprinting methods employing SSL/TLS packet flags [15], [23] or packet length statistics [16], [17], [27] (Section IV-A). Efficiency is also of importance to construct a DApp fingerprinting method. Existing studies that employ machine learning classifiers usually resort to hand-crafted features (e.g., fusing multiple dimensional features [25]), or even require compute-intensive feature processing (e.g., dynamic time warping of packet time series [17]). It is more desirable to simplify the sophisticated feature selection process and reduce the computational overhead for training classifiers.

In this paper, we extend our previous work [25] and propose GraphDApp, a DApp fingerprinting method using Graph Neural Networks (GNNs). We are motivated by an observation that each traffic *flow*, consisting of a series of packets resulting from client-server interactions, can be represented by an information-rich graph structure named Traffic Interaction Graph (TIG). The benefit of TIG lies in that it is capable of reserving as much information as the original flow, such as packet direction, length, ordering, and bursts without selecting features explicitly by hand.

With TIG, we convert the encrypted flow classification problem into a graph classification problem. As deep learning has shown its superiority over traditional machine learning techniques, we design a GNN-based classifier using multi-layer perceptions. The classifier can automatically extract features from input TIGs and distinguish different graph structures by mapping them to different representations in the embedding space. We conduct extensive experiments using real-world datasets to evaluate the performance of GraphDApp in the closed- and open-world settings.

Compared with our previous work [25], the *novelty* of this paper includes the new graph-based representation of encrypted flows and the new classifier using GNNs. We further extend performance evaluation by conducting a comprehensive comparison with the state-of-the-art on real-world traffic datasets. The additional contributions beyond the original paper [25] are as follows:

1) We propose Traffic Interaction Graph (TIG) to represent each individual encrypted flow, where vertices in a TIG represent packets and edges represent the packet-level interactions between a pair of client and server. We also provide quantitative measures to demonstrate the advantages of representing flows using TIGs over the traditional packet length sequence.
2) We design GraphDApp, a powerful GNN-based classifier using Multi-Layer Perceptions (MLPs) and a fully-connected layer. It maps TIGs of different DApp flows to different representations in the embedding space and does not require hand-crafted features so that the classification can be conducted in an effective and accurate way.
3) We collect real-world traffic datasets from 1,300 DApps on Ethereum with more than 169,000 flows. We demonstrate the accuracy and efficiency of GraphDApp in closed- and open-world settings. Compared with the state-of-the-art methods, GraphDApp has the highest classification accuracy with the shortest training time. In addition, it is also applicable to traditional mobile application classification.

To the best of our knowledge, this is the *first* study that addresses the encrypted traffic classification problem using graph classification techniques. The rest of this paper is organized as follows. Section II introduces the background of DApps and summarizes the related work. Section III presents the rationale of representing DApp flows using graph structures, and Section IV builds the GNNs-based classifier. Section V evaluates the performance of GraphDApp and makes a comprehensive comparison with the state-of-the-art methods. After a brief discussion of GraphDApp in Section VI, Section VII concludes this paper.

II. Problem Description and Related Work

In this section, we describe the DApp fingerprinting problem, briefly review existing studies, and emphasize the differences between our work and the existing methods.

A. Problem Description

DApp fingerprinting is a traffic analysis attack that aims at identifying users' visits to a certain collection of *monitored* DApps. To be able to perform this attack, an adversary needs to observe the traffic when a victim is visiting DApps. Following the common assumption on threat model in existing literature, we assume that the potential adversary is *local* and *passive*, which means that the adversary exists in the local network of victims (e.g., campus network administrators) and performs merely traffic collection without any active attacks such as traffic hijacking, as shown in Fig. 1.

In this paper, we focus on DApps deployed on the popular platform Ethereum, as it has the largest developer community. Most DApps on Ethereum provide browser plug-ins so that they can be visited via browsers (e.g., Chrome). The frontend of a DApp implements user interfaces uniformly defined by Ethereum to render pages. The backend is *smart contracts* connecting to Ethereum blockchain network, which is completely different from that of a web application that connects to a centralized database.

To initialize a visit, a DApp client (short for *client*) sends a request to the server equipped with the corresponding smart

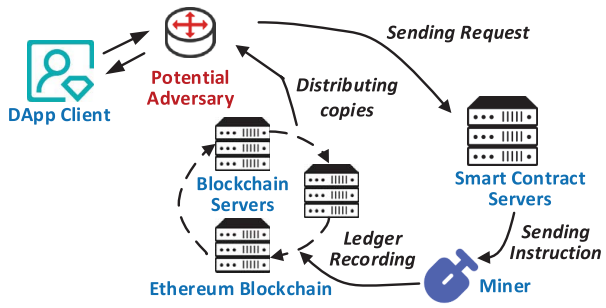


Fig. 1. Typical Workflow of DApps on Ethereum.

contracts, whose IP address can be obtained by DNS query. The smart contracts are the functionalities that determine the outcomes for each operation in DApp conducted by clients. All data records of these operations and outcomes are then packaged in the form of *blocks* by the miners on the underlying blockchain platform and stored on the distributed ledger. The client also gets a list of blockchain servers from the smart contract server, through which it can acquire updated information from the ledger to render webpages.

SSL/TLS protocols are used to encrypt the transmission data between clients and the blockchain platform. There are mainly two layers: the Handshake layer and the Record layer [23]. The former is used to negotiate secure parameters of an SSL/TLS session, while the latter is responsible for transferring encrypted data under the secure parameters.

Different from traditional mobile and web applications, DApps on Ethereum implement the same frontend interface, adopt similar settings in SSL/TLS implementation, running their backend codes and storing their data in the same decentralized blockchain network. These common features make the resulting traffic of different DApps less discriminative [25].

B. Related Work

Recent studies resort to machine learning techniques to deal with encrypted traffic classification. Here, we only review those that are closely related to our work, which can be roughly classified into three categories as shown in Table I.

1) Web Application Classification Methods: The goals of this category mainly focus on *website* fingerprinting and *webpage* fingerprinting. Website fingerprinting aims to identify the traffic generated from visits of certain websites. In practice, homepages are usually used as representatives of the corresponding websites. Panchenko *et al.* [17] leveraged the accumulated sum of packet lengths as features that were fed to a Support Vector Machine (SVM) classifier. Wang *et al.* [28] used a large feature set and adopted a k-Nearest Neighbor (k-NN) model to classify different websites. Recent studies attempted to use deep neural networks, such as Convolutional Neural Networks (CNNs) [18], [19], [26] and Long-Short Term Memory (LSTM) [19] to construct more accurate fingerprints.

The *webpage* fingerprinting aims to construct traffic fingerprints at the granularity of webpage, which can be viewed as a fine-grained version of website fingerprinting. Existing studies leverage either time information [10] or packet length information [20], [21] to build effective classifiers.

2) Mobile Application Classification Methods: The goals of this category mainly contain mobile application classification and user action identification. Taylor *et al.* [27] proposed Appscanner that combined statistical features of packet length with the random forest classifier to identify applications on smartphones. Several studies resorted to SSL/TLS flags in encrypted traffic and applied Markov Models to classify different smartphone applications [15], [23], [24]. User action identification attempts to recognize certain user operations on smartphone applications [9], [11], such as sending mail and replying a message in social network applications.

3) DApp Fingerprinting Methods: DApps are emerging as a new service paradigm that relies on the underlying blockchain platform. In our previous study [25], we obtained several key observations. First, statistical features of packet length are similar among DApps, e.g., the statistics of a DApp named Kitty are similar to those of Origin Protocol. Second, the adoption of the same blockchain platform makes the SSL/TLS message types of different DApps quite similar. The observations indicate that the classifiers built based merely on these features lose their efficacy for DApp fingerprinting. As a result, we proposed a feature fusion method, which leveraged mixed features of packet length, timestamp and burst, to build a classifier using Random Forest [25].

4) Summary: The limitations of existing methods lie in two aspects. *First*, the methods using traditional machine learning classifiers (e.g., [17], [22], [25], [27], [28]) rely on carefully-selected features, which become less discriminative for DApp classification and require a time-consuming process of feature extraction (see Section V). *Second*, the methods using deep neural networks (e.g., [19], [26]) generally employ packet direction sequence or packet length sequence as their input, which are not expressive enough to achieve high accuracy.

C. Differences From Existing Work

Our goal is to achieve high classification accuracy of DApp flows and avoid the exhausting process of feature selection. This paper tackles these challenges from a novel angle. Our observations on DApp traffic interactions motivates the graph representation of DApp flows, which naturally converts DApp fingerprinting into a graph classification problem. Then, GNNs are employed to build a powerful classifier that automatically extracts multi-dimensional features from the raw input. The evaluation results show that the proposed method is superior to the state-of-the-art methods metioned in Section V. We also demonstrate that the proposed method can be generalized to other classification problems, such as the traditional mobile application classification.

III. Graph Structure Abstraction for DApp Flows

In this section, we first elaborate on the rationale for representing DApp flows by graph structure before formally describing the graph construction process. We also present case studies to illustrate the differences of the graphs abstracted from different DApps.

TABLE I
SUMMARY OF ENCRYPTED TRAFFIC ANALYSIS METHODS

Category	Goal	Refs.	Classifier	Traffic Features
Web Application Classification	Website Fingerprinting	[17]	SVM	Accumulative sum of packet lengths
		[28]	k-NN	Features of packet length, account, ordering and burst
		[18, 19, 26]	CNNs	Packet direction/length sequences
		[19]	LSTM	Packet direction sequence
	Webpage Fingerprinting	[20, 21]	Random Forest	Cumulative downlink packet length
		[10]	k-NN	Dynamic time warping distance of downlink packets
Mobile Application Classificaiton	Mobile App Identification	[27]	Random Forest	Statistical features of packet sequences
		[15, 23, 24]	Markov Model	State transition in SSL/TLS handshake
	User Action Identification	[9]	Random Forest	Dynamic warping distance & hierarchical clusters
		[11]	Transfer Learning	Payload of packet sequences
DApp Classificaiton	DApp Fingerprinting	[25]	Random Forest	Mixed features of packet length, timestamp and burst
		This paper	GNNs	Graph representation of traffic interaction features

A. Traffic Interaction Graph

According to the problem description in Section II, we can collect all the traffic (i.e., a sequence of incoming and outgoing packets) when a client visits a certain DApp. Since the packet sequence usually consists of multiple flows, we first partition the whole sequence into individual flows, where each *flow* is defined as a series of packets with the same 5-tuple (i.e., source/destination IP addresses, source/destination ports, and protocol). The definition of flows is also commonly used in the literature. More details of flow partitioning will be described in Section V-B.

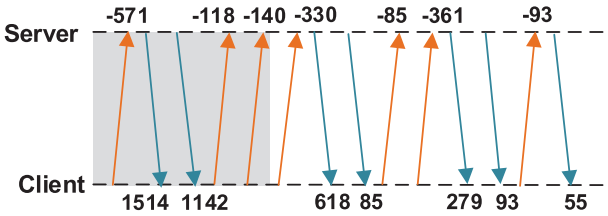
We take a specific flow as an example to illustrate the client-server interactions in terms of packet lengths and directions, as shown in Fig. 2(a). From the perspective of client, we set the length of upstream packets as *negative* and that of downstream packets as *positive*. The gray and white blocks in Fig. 2(a) represent the Handshake stage and the Record stage in SSL/TLS protocols, respectively. It is worth noting that the timestamp of packets is not considered in this paper, as it is easily affected by varying network conditions and the alignment of time series for different flows is proved to be time-consuming [20].

In the existing literature [17], a flow P consisting of n packets is usually represented by a packet length vector. It can be expressed as $P = (p_1, p_2, \ldots, p_n)$, where p_i is the signed length of the i-th packet (i.e., negative for upstream and positive for downstream). However, this vector has limited representational capacity of the client-server interactions. Inspired by information-rich graph structures, we attempt to represent a traffic flow by a *graph*, which is referred to as *Traffic Interaction Graph* (TIG).

We first describe TIG construction process with the example in Fig. 2 before presenting the formal definition of TIG.

Vertex. Given a specific flow, a vertex of the corresponding TIG represents a packet in the flow. Each vertex is also associated with the length of the packet. To reflect packet direction, we maintain the signed value of each packet length.

When vertices are determined, edges should be added to connect these vertices together. A flow typically contains multiple bursts, where a *burst* is usually defined as a sequence



(a) Packet-level Client-Server Interaction for DApp Flows

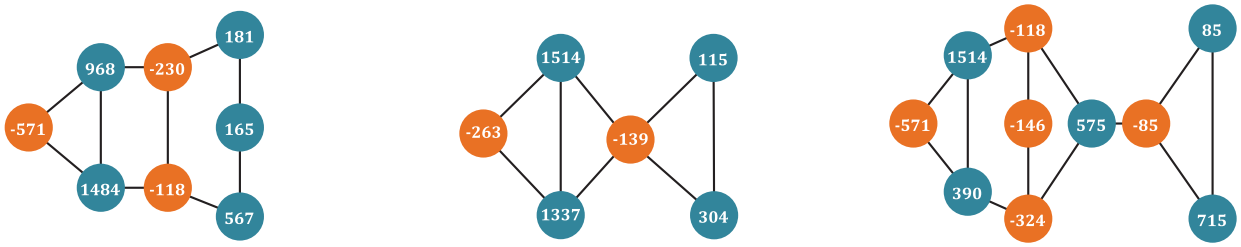
(b) Traffic Interaction Graph (TIG) for DApp Flows

Fig. 2. Graph-based Representation of DApp Flows.

of consecutive packets transmitted along the same direction during a short time interval [7]. Here, we misuse the terminology burst to define a series of consecutive packets transmitted along the same direction, even if there is only a single packet. As shown in Fig. 2(a), the arrows in orange and blue represent different bursts in the flow.

Edges. There are two types of edges in TIG: *intra*-burst edges and *inter*-burst edges. Intra-burst edges sequentially connects the consecutive vertices (i.e., packets) in each burst. For instance, for the burst [-118, -140, -330], two edges are used to connect the three vertices. Inter-burst edges are used to connect a burst with its preceding burst, i.e., starting from the second burst, the first and the last vertices in each burst are connected to the corresponding first and last vertices in its preceding burst. Note that *at most* one edge can be added between each pair of vertices in case that only one packet exists in a burst. Figure 2(b) shows the resulting TIG corresponding to the flow in Fig. 2(a).

Definition 1 (Traffic Interaction Graph): A TIG is represented by a three-tuple, TIG $= (V, E, L)$, such that:



(a) TIG of an Aigang flow (b) TIG of an Aragon flow (c) TIG of an AaveProtocol flow

Fig. 3. TIG Samples of Three Different DApps.

- V is a set of vertices. Each vertex $v \in V$ is associated with a signed non-zero integer $l_v \in L$ representing the packet direction and length.
- E is a set of edges. Each edge $e \in E$ is either an intra-burst edge connecting two consecutive vertices in a burst or an inter-burst edge connecting the starting or ending vertices of two consecutive bursts.
- L is the set of non-zero integers, where the absolute value of each integer is limited by the summation of packet header length and Maximum Transmission Unit (MTU).

B. TIG Samples of DApps

With an intuitive description of TIG in the last subsection, we formally present the construction process of TIG, as depicted in Algorithm 1.

The construction process considers the packet length sequence of a specific flow as the input and starts with an initialization of the vertex set V and edge set E. Then, it constructs the vertex set (lines 2-3) and obtain the bursts (line 4). After that, it iteratively adds intra-burst edges (lines 5-8) and inter-burst edges (lines 9-13). Finally, the resulting G is the desired TIG.

To get a better understanding of the representation capacity of TIG, we select 3 kinds of DApps (i.e., Aigang, Aragon and AaveProtocol) and visualize the corresponding TIGs constructed with Algorithm 1, as shown in Fig. 3. Since it is impossible to enumerate the TIGs for all the flows, we randomly select a flow for each DApp and use Matplotlib [4] to draw the corresponding TIG. It is clear to observe the differences in terms of graph structure: Aigang owns spindle-shaped TIGs for its flows, Aragon has simple and compact TIGs, while AaveProtocol exhibits complex and fish-shaped TIGs.

C. The Benefits of TIG

The following explains why TIG is a powerful representation of DApp flows. In general, TIG can extract features of DApp flows from four aspects, each of which has been proven to be valuable for traffic classification.

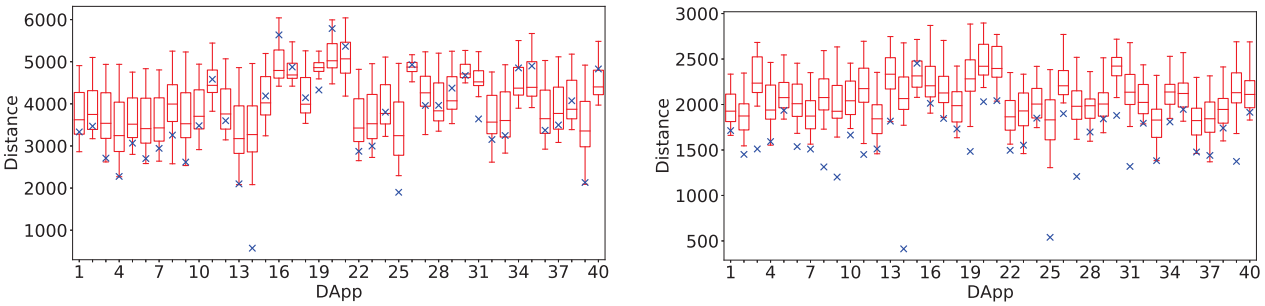
1) **Packet direction information.** The direction information is revealed by the sign of the vertex in TIG, where positive values indicate downstream packets while negative for upstream packets.
2) **Packet length information.** The packet length sequence, as well as its mathematical variants, are often used as the key features in encrypted traffic classification [17]. Vertices in TIG are associated with the corresponding packet lengths, which can be naturally used by classifiers.
3) **Packet burst information.** The vertices of the same layer in TIG represent the packets composing an individual burst. The burst-level behaviors for different applications can vary significantly and thereby act as discriminative features learned by classifiers.
4) **Packet ordering information.** TIG can represent the order of packets from the beginning of SSL/TLS session negotiation to the end of application data transmission. In addition, TIG also reflects the interaction between the server and the client.

Algorithm 1 Construction of Traffic Interaction Graph

Input: A packet length sequence $P = (p_1, \ldots, p_N)$
Output: The corresponding traffic interaction graph $G = (V, E)$

1: Initialize V and E as empty sets
2: **for** $p_i \in P$ **do**
3: Add a vertex with a length value of p_i in V
4: Separate V into bursts $B = (b_1, \ldots, b_K)$ according to packet direction
5: **for** $b_i \in B$ **do**
6: **if** $len(b_i) > 1$ **then**
7: **for** $v_j \in b_i$ **do**
8: Add an edge between v_j and v_{j+1} in E
9: **for** $b_i \in B$ **do**
10: **if** $len(b_i) = 1$ and $len(b_{i+1}) = 1$ **then**
11: Add an edge between b_i and b_{i+1} in E
12: **else**
13: Add two edges between b_i and b_{i+1} in E
14: **return** G

We resort to quantitative measures to demonstrate that TIG is more informative than other representations. An *ideal* representation should make a flow similar to flows from the same DApp while distinctive from those of different DApps. We select *packet length sequence* as a baseline representation for comparison, as it is commonly used in the literature [17]. We employ graph edit distance [5] and the Euclidean distance as the similarity metrics for TIGs and packet length sequences, respectively. Note that a smaller distance indicates more similarity.



(a) Euclidean Distance with Packet Length Sequences

(b) Graph Edit Distance with TIGs

Fig. 4. Similarity Measurements with Metrics of Euclidean Distance (a) and Graph Edit Distance (b) among 40 DApps.

We randomly select 100 flows from each of the 40 DApps in Table III and calculate the pairwise distance of flows. When calculating the graph edit distance, we follow the setting in [5], i.e., the node substitution cost is set to the Euclidean distance, the node insertion and deletion cost is set to 90, and the edge insertion and deletion cost is set to 15.

The distance measurements are shown in Fig. 4. For each DApp, the blue marker represents the average distance between flows in the same DApp (i.e., intra-class distance), and each boxplot represents typical distances from other DApps (i.e., inter-class distances), i.e., the maximum values, 75th percentile, 50th percentile, 25th percentile, and the minimum values. We can obtain the following observations by comparing Figs. 4(a) and 4(b):

1) With packet length sequence, only 4 DApps have their intra-class distances smaller than the minimum value of their inter-class distances; while TIG makes 21 DApps have this property.
2) With packet length sequence, there are 15 DApps whose intra-class distance are larger than the median (i.e., the 50th percentile) of inter-class distances; while there is only one such case with TIG.

Through quantitative measures, we demonstrate that TIG-based representation exhibits high similarity for flows of the same DApp and is more discriminative across different DApps.

IV. The Proposed GraphDApp

With the help of TIG, the classification of DApp flows is turned into a graph classification problem. GNNs become a natural solution to this problem, as they can automatically extract features from input graphs and distinguish between these graph structures [29]. In this section, we present the design details of the proposed GNN-based classifier named GraphDApp.

A. GraphDApp Overview

The overview of training GraphDApp is presented in Fig. 5. In the training process, we first collect encrypted traffic of DApps and parse the packet sequences into flows. Using Algorithm 1 described in Section III, we can obtain the corresponding TIGs for all the DApp flows. Then, the TIGs are fed into GNNs to learn a powerful classifier. We employ MLPs and a fully-connected layer to construct GraphDApp, as they do not require selection and fine-tuning features by hand and can effectively classify different graph structures [29]. In the testing stage (not shown in Fig. 5), the well-trained classifier is used to label the TIGs abstracted from unknown DApp flows.

TABLE II
List of Notations

Notation	Meaning
G	A TIG as the input of GNNs
v, e	A vertex and an edge in G
f	Neighbor feature aggregation function in MLPs
$\mathcal{N}(v)$	The set of nodes adjacent to node v
h_v	Feature representation of node v
h_G	Feature representation of graph G
χ	Feature space in GNNs
$g^{(k)}$	The k-th layer of MLPs
$\mathbb{R}^n$	A subset of a n-dimensional countable set
H_G	Representation of G in a latent space
y_{ic}	Actual label vector
$\hat{y}_{ic}$	Predicted label vector by GNNs
$\mathcal{L}$	The loss function in GNNs

B. GNN Architecture

Given a set of TIGs $\{G_1, \ldots, G_N\} \subseteq \mathcal{G}$ and their labels $\{y_1, \ldots, y_N\} \subseteq \mathcal{Y}$, GNNs aim to learn a representation vector h_G that can predict the label of each TIG, i.e., $\hat{y}_N = g(h_G)$. The main body of GNNs used in GraphDApp consists of MLPs and a fully-connected layer, where MLPs are used to extract features from the TIGs for training while the fully-connected layer is used to make a classification decision. GNNs try to gradually reduce the value of a loss function $\mathcal{L}$, which quantitatively measures the differences between the predicated and the actual labels. The notations used in this paper are shown in Table II.

1) Multi-Layer Perceptions: In GraphDApp, MLPs are used to learn the representation vector h_G for each TIG G. We employ n layers of perceptions, and each layer acts as a recursive neighborhood aggregation scheme, in which a feature vector h_v of each node $v \in G$ is calculated by aggregating the features of its neighbors. Thus, the feature vector h_v can store the feature information of its neighbor nodes in the graph.

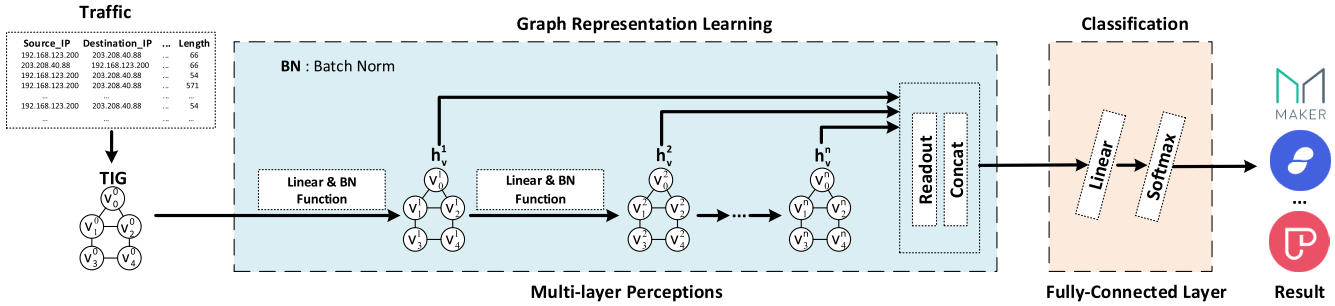


Fig. 5. Structure of Neural Networks in GraphDApp.

More specifically, each layer of MLPs consists of a *Linear* function and a *BatchNorm* function. The Linear function performs linear transformation that is required to learn the weights and biases in the learning process. The BatchNorm function is used to make sure that the input of each layer of MLPs has the same distribution during the training process. We use a *dropout* function to avoid over-fitting when neural network forward propagates. Finally, the representation of the entire graph is obtained through a *concatenation* of the *Readout* function across all the layers of MLPs. We defer the design detail of MLPs in the next subsection.

2) Fully-Connected Layer: Following MLPs, GraphDApp uses a linear function to perform linear transformation on the output data of MLPs. Again, a dropout function is used to avoid over-fitting. For each TIG G_i, we refer to the output of the linear function as its feature representation vector h_{G_i}. To facilitate the following prediction process, we need to map h_{G_i} into a new latent space $H_{G_i} \in \mathbb{R}^C$, where C is the amount of distinct elements in $\mathcal{Y}$ (i.e., the number of DApps to identify). Then, a *softmax* function is used to obtain the predicted probability vector $\hat{y}_{ic}$ indicating the likelihood that G_i belongs to each type of DApp, as shown in Eq. (1).

$$\hat{y}_{ic} = Softmax\left(H_{G_i}\right) \tag{1}$$

3) Loss Function: GraphDApp leverages the cross entropy function as the loss function, as it is commonly used in multi-classification problem to calculate the loss between the predicted labels and the ground truth, as defined in Eq. (2),

$$\mathcal{L} = -\frac{1}{|X|}\sum_{i=1}^{|X|}\sum_{c=1}^{C} y_{ic} log\left(\hat{y}_{ic}\right) \tag{2}$$

where $|X|$ is the number of instances (i.e., TIGs) for training and y_{ic} is the ground truth label. In our multi-classification scenario, it has better convergence properties than the Mean Squared Error (MSE) loss function.

4) Optimizer: We adopt the Adam optimizer in GraphDApp. Adam is a stepwise optimization algorithm based on a stochastic objective function of adaptive low-order moment estimation. It is an effective stochastic optimization method that requires only first-order gradients and little memory.

C. Design of MLPs

The design of MLPs is motivated by the fundamental goal of a GNN-based model. Ideally, GraphDApp should distinguish different graph structures, which means that it should be able to map different graphs to different representations in an embedding space. This implies that the ability also solves the *graph isomorphism* problem, where non-isomorphic graphs should be mapped to different representations [8].

Recent advances [29] in theoretical analysis of GNNs characterize the expressiveness of GNNs by the Weisfeiler-Lehman (WL) graph isomorphism test, which is a powerful heuristic to address the graph isomorphism problem. It has been proved that GNNs are *at most* as representational as the WL test in distinguishing different graph structures [8]. Following the theoretical basis for establishing a GNN as powerful as WL, we should make sure that the sufficient conditions should be satisfied, i.e., node feature aggregation and the graph-level readout in GNNs should be *injective* [29].

To map different graphs to different representations, GNNs should have an injective neighbor aggregation method. The update method of node feature vectors [29] is expressed as shown in Eq. (3),

$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)\right) \tag{3}$$

where $h_v^{(k)}$ is the feature vector of node v at the k-th iteration (i.e., layer), $\mathcal{N}(v)$ is the set of nodes adjacent to node v. On each layer, the function f is responsible for aggregating neighbor nodes of v and the function ϕ updates node features. In addition, ϕ and f should be injective.

Note that these aggregation functions should work on the multiset that allows multiple instances of each distinct element. Some popular injective functions, such as mean aggregation, are not injective on multisets. As a concrete example, the *summation* function is injective over multisets [18]. Many aggregation schemes can potentially satisfy the injectivity over multisets. Theorem 1 provides a simple and concrete formula in many of these aggregation schemes [29].

Theorem 1 (Aggregation [29]): Assume Ω is countable. There exists a function $f : \Omega \to \mathbb{R}^n$ so that for infinite choices of ε, including all irrational numbers, $h(c, X) = (1+\varepsilon)\cdot f(c) + \sum_{x\in X} f(x)$ is unique for each pair (c, X), where $c \in \Omega$ and $X \subset \Omega$ is a multiset of bounded size. Moreover, any function g over such pairs can be decomposed as $g(c, X) = \varphi\left((1+\varepsilon)\cdot f(c) + \sum_{x\in X} f(x)\right)$ for some function φ.

This theorem shows that for any $(c, X) \neq \left(c', X'\right)$ with $c, c' \in \Omega$ and $X, X' \subset \Omega$, $h(c, X) \neq h\left(c', X'\right)$ holds, if ε is an irrational number. Here we provide an intuition of the correctness of Theorem 1, where a rigorous proof can be found

in [29]. In general, there are two cases with $(c, X) \neq (c', X')$: 1) $c = c'$ but $X \neq X'$, and 2) $c \neq c'$. In the first case, the injectivity of $f(x)$ makes $\sum_{x \in X} f(x) \neq \sum_{x \in X'} f(x)$ and thus the conclusion holds. In the second case, since the left part of $h()$ is irrational and the right part is rational, the differences of two parts for c and c' never counteract with each other and thus the conclusion holds.

MLPs are able to model and learn f and φ in Theorem 1 according to the universal approximation theorem [13], [14]. Since MLPs can represent the composition of functions, we model $f^{(k+1)} \circ \varphi^{(k)}$ for one MLP. As the summation of the input features with one-hot encoding is naturally injective, MLP is not required in the first iteration. Then, GNNs update node representations as shown in Eq. (4):

$$h_v^k = MLP^k \left(\left(1 + \epsilon^{(k)}\right) \cdot h_v^{k-1} + \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)} \right) \quad (4)$$

where ϵ is a learnable parameter.

The node representation of subtree structure becomes more refined and global with the increment of the number of iterations. It is crucial to have a sufficient number of iterations for achieving good discrimination. To extract enough information form the neural networks, information from all the iterations (i.e., layers) of the model should be reserved. We use an architecture similar to Jumping Knowledge to achieve this goal as shown in Eq. (5),

$$h_G = Con\left(Readout\left(\left\{h_v^{(k)} | v \in G\right\}\right) | k \in [0, K]\right) \quad (5)$$

where graph representation in each layer is obtained by *Readout* and then concatenated across all layers of MLPs.

With all the designs above, GraphDApp captures the similarity of graph nodes and structures through training, which will be demonstrated by experimental results in the next section.

V. Performance Evaluation

In this section, we are devoted to evaluating the effectiveness of GraphDApp. We first introduce the experimental settings and the datasets for evaluation before tuning the hyperparameters of GraphDApp and comparing the performance of GraphDApp with the state-of-the-art in closed-world and open-world scenarios. In addition, we also demonstrate the applicability of the proposed method on the classification of traditional mobile applications.

A. Preliminary

1) Methods in Evaluation: To fully understand the performance of GraphDApp, we leverage 6 typical methods for comparison, which are briefly described as follows. To make a fair comparison, all the methods are fine tuned to achieve their best accuracy on the datasets used for evaluation.

- Markov Model (MARK), which uses Markov chains to model the sequences of message types in SSL/TLS sessions and feeds the state transition features into machine learning classifiers. The Maximum Likelihood principle is used to identify the application from which encrypted flows are generated [15].

TABLE III
Datasets for Closed-World Evaluation

Category	Application (Number of Flows)
Exchanges	Joyso (395), Bancor (290), Idex (3119), Lescovex (4464), Uniswap (1924)
Development	Oxdrop (5016), Golem (2207)
Finance	UQUID (7290), Aave Protocol (4004), Compound (1198), Makerdao (128)
Gambling	Bestdice (8965)
Goverance	Aragon (431), Kleros (919), IoTeXVotingPortal (425)
Identity	SelfKey (5215), LikeCoin (3049), SpringRole (11839)
Insurance	Aigang (3064)
Marketplaces	Multiven (6139), Reviewhunt (953), OpenSea (435), Originprotocol (799)
Media	Ompx (4689), Chainy (5474), Captain Bitcoin (1938)
Game	EthTown (3256), Kitties (215), Pepes (752), Etheremon (922), Fomo2Moon (1028)
Property	Ethername (5918), Foam (38412), ULSDocuments (7047)
Social	SteemSTEP (3570), Canwork (901), Ono (490), Lordless (581), Matchpool (1283)
Wallet	Greymass (6756)

- AppScanner (APPS), which captures statistical features from a sequence of packets, such as mean, minimum, maximum, standard deviation of incoming packets, outgoing packets, and bi-directional packets. It uses a Random Forest classifier to identify encrypted or unencrypted traffic flows from mobile applications [27].
- Feature Fusion (FEAF), which is the method proposed in our previous conference version [25]. It fuses different dimensional features (e.g., packet length, burst, packet timestamps) by kernel functions and uses a Random Forest classifier to identify encrypted flows from DApps.
- DeepFingerprinting (CNN+D), which uses the information of packet direction to construct Convolutional Neural Networks (CNNs) to classify the encrypted traffic of different websites [26].
- CNN+L, which uses the same CNN structure as CNN+D but takes the packet length sequence as input to build classifiers.
- LSTM+L, which takes packet length sequence as input and uses a double-layer LSTM [19] to learn feature representations and a fully-connected layer for classification.

2) Cross-Validation: The 10-fold cross-validation is used to evaluate the performance of each method. We randomly divided the dataset into 10 mutually exclusive subsets of similar sizes. Then, we conducted 10 times of training and tests, using 9 subsets as the training set and the rest one as the test set each time. The average value of the 10 tests is used as the final result. All the experiments are conducted on a server with an Intel Core Duo 3.60GHz and 16GB memory.

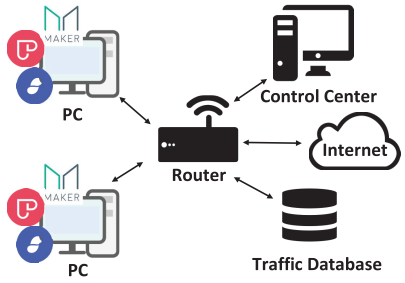


Fig. 6. Process of Network Traffic Collection.

B. Dataset Collection

Figure 6 shows the dataset collection process in our experiments. The network traffic capture tools are deployed on the routers of laboratories located in different university campuses in China. When users visit a certain DApp through a Chrome browser on PCs, the resulting network traffic is captured by WireShark and saved on data servers. All visits use Chrome, as it is the designated browser by some DApps. Network flows are then exported to a Comma Separated Value (CSV) file, each row of which contains the information obtained from the packets, including time, source/destination IP addresses, ports, protocols, packet lengths, and TCP/IP flags. The encrypted payload of each packet is not used for classification.

Closed- and Open-World Settings: In the closed-world setting, an adversary aims at identifying victims' visits to a certain collection of *monitored* DApps, which can be viewed as a multi-classification problem. The open-world setting considers a more realistic scenario, where victims not only visit the monitored DApps, but also visit a larger amount of *unmonitored* DApps. The goal is to identify monitored DApps from unmonitored DApps, which is usually treated as a binary-classification problem [26].

To construct the closed-world dataset, we select the top 40 DApps on Ethereum with the most users [2] as the *monitored* DApps. The categories include social communication applications, finance, online shopping, etc. The number of flows for each monitored DApp is summarized in Table III. In total, we collect 155,500 flows.

To build the background traffic for open-world evaluation, we randomly select 1,260 DApps on Ethereum as the *unmonitored* DApps and collect 14,000 flows in total. Following the strategy in the literature, each unmonitored DApp is accessed only once [26].

C. Parameter Tuning of GraphDApp

In this subsection, we will introduce the hyperparameters used in GraphDApp and evaluate the trade-offs between classification accuracy and efficiency. Note that all the experiments are conducted in the closed-world setting.

1) Hyperparameter Selection: An important step in training GNNs is to tune the hyperparameters, which adjusts the trade-offs between variance, bias and classification performance. Due to the large number of training instances and hyperparameters in GraphDApp, it is a challenging task to find the optimal setting of hyperparameters. Therefore, we search the hyperparameters from an interval and select the best combination. More specifically, at the training stage, we change each hyperparameter to estimate the gradient of the parameters and decide whether the hyperparameters should be increased or decreased. After completing this process, we select the best top-k values for each parameter and use them as the candidates for selecting the final best combination of all the hyperparameters. We use *accuracy* as the performance metric, which is defined as the proportion of all the DApps flows that are classified correctly.

TABLE IV
HYPERPARAMETERS SELECTION FOR GRAPHDAPP

Hyperparameters	Search Range	Final
Optimizer	[Adam, RMSProp, SGD]	Adam
Learning Rate	[0.0001, ..., 0.02]	0.0005
Training Epochs	[5, ..., 40]	10
Batch Size	[30, ..., 300]	150
Activation Functions	[Tanh, Relu, Elu]	Relu
Dropout	[0, ..., 0.3]	0.025
Number of MLP layers	[1, ..., 10]	3
Number of Hidden Units	[0, ..., 100]	64
Graph Pooling Type	[Sum, Average]	Sum
Neighbor Pooling Type	[Sum, Average, Max]	Sum

TABLE V
IMPACT OF THE NUMBER OF TRAINING EPOCHS ON CLASSIFIER TRAINING TIME (CTT), TESTING ACCURACY, AND DIFFERENCE BETWEEN THE TRAINING AND TESTING ACCURACY OF GRAPHDAPP

Epochs	1	2	4	10	15	20
CTT (s)	25.27	39.18	67.45	164.80	213.19	278.51
Testing Accuracy	0.8305	0.8713	0.8836	0.8922	0.8957	0.8963
Difference	0.0053	0.0024	0.0062	0.0094	0.0111	0.0110

The hyperparameter search ranges and the selected values are shown in Table IV. GraphDApp is set with the learning rate of 0.0005, the training epoch of 10 and the batch size of 150. It uses MLPs to extract features and a fully-connected layer to output the predicted results.

Next, we investigate the impact of parameters on trade-offs between classification accuracy and time consumption.

2) Epochs: Table V shows the Classifier Training Time (CTT) and testing accuracy with different epochs. GraphDApp can reach a testing accuracy of around 0.83 with only one epoch (25.27s). In general, increasing the number of epochs helps improve classification accuracy. When the number is larger than 10, the accuracy increment becomes trivial while CTT increases significantly. Therefore, we use 10 epochs to achieve a better balance between accuracy and training efficiency.

The difference between training accuracy and testing accuracy of neural networks is usually used to determine whether the classifier is over-fitting [26]. Table V also shows that GraphDApp keeps the difference less than 0.02, indicating that it can avoid over-fitting. The reason is we use dropout and BatchNorm to prevent over-fitting: The dropout function randomly selects hide units and temporarily removes them from training the neural networks, and BatchNorm normalizes

TABLE VI

IMPACT OF NUMBER OF PACKETS USED IN EACH FLOW ON FEATURE EXTRACTION TIME (FET), CLASSIFIER TRAINING TIME (CTT) AND ACCURACY OF GRAPHDAPP

# Packets	6	10	15	20	25	30
FET (s)	8.74	13.23	17.07	19.73	22.96	23.47
CTT (s)	86.79	98.43	107.23	134.72	164.80	196.03
Accuracy	0.8532	0.8853	0.8876	0.8889	0.8922	0.8935

TABLE VII

IMPACT OF DATASET SCALE ON FEATURE EXTRACTION TIME (FET), CLASSIFIER TRAINING TIME (CTT) AND ACCURACY OF GRAPHDAPP

Proportion (%)	20	40	60	80	100
FET (s)	4.60	9.38	14.33	19.79	22.96
CTT (s)	144.58	149.35	156.25	162.68	164.80
Accuracy	0.8764	0.8873	0.8980	0.8914	0.8922

the fully-connected layer and the output of the MLPs, which helps accelerate learning process and reduce over-fitting.

3) Packet Number: The number of packets in each flow used to construct the corresponding TIG also has an impact on the performance of GraphDApp. To take a deeper look at the time consumption, we roughly divide the training process into two stages: feature extraction (i.e., constructing TIGs from flows) and classifier training (i.e., learning graph representation from TIGs), as shown in Fig. 5.

The FET and CTT of GraphDApp, as well as its accuracy, with a varying number of packets used in each flow are shown in Table VI. When only the first 6 packets of each flow are used, the accuracy of GraphDApp can reach 0.8532. With the increase of the number of intercepted packets, both FET and CTT become larger accordingly but are with different growth trends: a slowdown in the growth of FET and an accelerated growth of CTT, because a larger amount of packets leads to more complex TIGs, requiring much more time to learning their representations in the classifier training stage. At the same time, the accuracy is also improved when provided with more packets to construct TIGs. We set the packet number as 25 to achieve higher accuracy with moderate time overhead.

4) Dataset Scale: To investigate the impact of dataset scale on the performance of GraphDApp, we vary the proportion of our original dataset (see Table III). Given a proportion value, e.g., 20%, we randomly select 20% flows of each DApp and constitute a new dataset, over which the 10-fold cross-validation strategy is applied to evaluate the time consumption and accuracy. The results are summarized in Table VII.

GraphDApp can achieve 0.8764 accuracy when only 20% samples are used, indicating that it can learn sufficient graph representations from limited training data. When provided with more training samples, GraphDApp can further improve its accuracy. An interesting observation is that a lager dataset does not lead to a significant increase of CTT, which shows that graph representation learning is not severely affected by redundant TIGs. In the following experiments, we use the entire dataset in Table III unless otherwise specified.

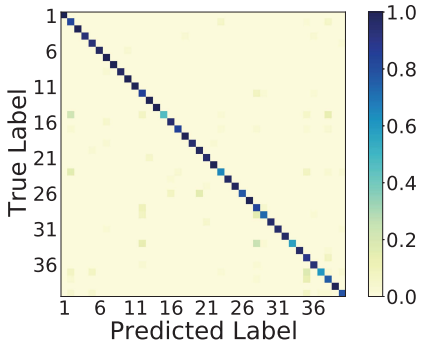


Fig. 7. Closed-world: Confusion Matrix of GraphDApp.

D. Closed-World Evaluation

In this subsection, we investigate the effectiveness and time efficiency of different classifiers in the closed-world setting. In this scenario, all the visits to DApps are limited to the monitored DApps (i.e., those listed in Table III) and DApp identification is a multi-classification problem.

1) Accuracy: We also use *accuracy* to measure the performance of all the methods. The average accuracy and the standard deviation in cross-validation are exhibited in Table VIII. We can obtain several major observations:

1) GraphDApp outperforms the rest methods, with the highest accuracy (0.8922) and the smallest standard deviation (0.0011). Figure 7 shows the confusion matrix of GraphDApp. The accuracy of 5 DApps reaches 1.0 and the accuracy of 67.5% DApps reaches above 0.9, while the accuracy of only 3 DApps is lower than 0.6.
2) Traditional machine learning methods with statistical features (e.g., FEAF and APPS) are comparable to deep learning methods with packet length sequences (e.g., CNN+L). MARK results in low accuracy, because the Markov state transition features used in MARK are quite similar for flows from the same platform (i.e., Ethereum).
3) Packet length sequence cannot work well with LSTM, as the accuracy of LSTM+L is around 20% less than CNN+L. The reason is most packets of DApps are transmitted with the fixed maximum length, making the temporal information in packet length sequence less distinguishable. It is consistent with the conclusion in previous literature [19].

The results demonstrate the superiority of representing traffic flows using TIGs and transforming the traffic classification problem into the graph classification.

2) Time Overhead: We evaluate the time cost of all the methods from two aspects: training time and testing time. Tables IX and X present the results of different methods in the closed-world scenario.

Training Time: In general, the training process of each method in comparison can be divided into two stages: feature extraction from encrypted flows and classifier training using these features. The time on each stage and the total time are reported in Table IX. Note that all the methods are trained on the same dataset.

TABLE VIII
CLOSED-WORLD: ACCURACY OF THE STATE-OF-THE-ART METHODS

Method	MARK	FEAF	APPS	LSTM+L	CNN+D	CNN+L	GraphDApp
Accuracy	0.1656±0.0017	0.8155±0.0023	0.7956±0.0025	0.5902±0.0018	0.7090±0.0013	0.7938±0.0033	0.8922±0.0011

TABLE IX
TRAINING TIME OF DIFFERENT METHODS

Methods	FET (s)	CTT (s)	Total (s)
FEAF	3,470.51	345.08	3,815.59
APPS	1,538.82	108.26	1,647.08
LSTM+L	16.83	414.44	431.27
CNN+D	13.90	763.95	777.85
CNN+L	16.83	765.40	782.23
GraphDApp	22.96	164.80	187.76

TABLE X
TESTING TIME OF DIFFERENT METHODS

Methods	FET (ms)	Prediction Time (ms)	Total (ms)
FEAF	22.32	0.08	22.40
APPS	9.89	0.10	9.99
LSTM+L	0.11	0.09	0.20
CNN+D	0.09	0.13	0.22
CNN+L	0.11	0.13	0.24
GraphDApp	0.15	0.24	0.39

FEAF takes the longest time in training, because it needs to first calculate features from packet length, timestamps and bursts using a kernel function and then merge features by selecting features with relatively high contribution before finally training the Random Forest classifiers. APPS calculates statistical features from packet length, which is a time-consuming process. Even though it takes the least time to train classifiers, APPS eventually becomes the second most time-consuming method.

Compared with the deep learning methods (i.e., LSTM+L, CNN+D and CNN+L), GraphDApp takes slightly longer time to construct TIGs from flows. However, it has more powerful learning capabilities and needs less iterations to achieve the desirable accuracy. Thus, GraphDApp has the highest time efficiency in terms of the total training time.

Testing Time: The testing time refers to the time spent on classifying an unknown flow by a well-trained classifier, which includes two parts: feature extraction time to obtain required features from the flow and prediction time to label this flow.

In general, the testing time of all the methods are no more than 0.03 seconds. FEAF takes the longest time to make a decision, as it needs more time to fuse multi-dimensional features from a flow. Similarly, the time of feature extraction makes APPS the second most time-consuming method in testing. Compared with FEAF and APPS, the deep learning methods are much more efficient in labeling an unknown flow. GraphDApp has a more complex neural network structure, so it takes slightly longer time for prediction.

3) Summary: The closed-world evaluation demonstrates that GraphDApp can achieve the most accurate classification results with the shortest training time.

E. Open-World Evaluation

In this subsection, we will evaluate the performance of different methods in a more realistic open-world scenario. Existing studies usually treat this problem as a *binary* classification [12], [26]. Following this standard model, the classifier in open-world evaluation aims at distinguishing between flows of the *monitored* and *unmonitored* DApps. Note that if a flow is determined to be monitored, we can further use the multi-class classification (i.e., the closed-world setting) to identify to which monitored DApp it belongs.

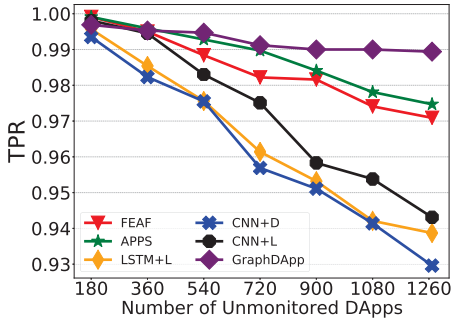
1) Dataset: We randomly select an average of 800 flows for each monitored DApp from those listed in Table III and obtain a total of 32,000 *monitored* flows.

To conduct cross-validation, we partition the *unmonitored* flows at the granularity of DApps. Given a certain number of unmonitored DApps, e.g., 1,260, we divide these DApps into 10 subsets, where each subset has 126 unique unmonitored DApps. Each time 9 subsets are used for training and the rest subset is used for testing. By this way, we can make sure that there is no overlap between the training and testing datasets in terms of unmonitored DApps. In other words, all the unmonitored DApps for testing are not learned in advance.

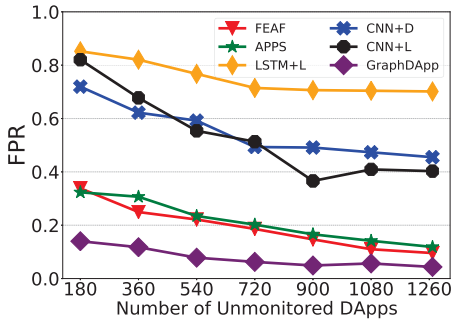
2) Criteria: We use the prediction probability output by the classifier to label unknown flows. If a flow of monitored DApp has a prediction probability greater than a certain threshold, we will record it as True Positive (TP), or False Negative (FN) otherwise. Similarly, if a flow of unmonitored DApp is correctly labeled, it will be considered True Negative (TN), or False Positive (FP) otherwise.

To evaluate the performance of classifiers on the two-class classification, we use True Positive Rate (TPR), False Positive Rate (FPR), precision, recall and Area Under Curve (AUC) as criteria. TPR is calculated by $TP/(TP + FN)$ and FPR equals to $FP/(FP + TN)$. Precision and recall are defined as $TP/(TP + FP)$ and $TP/(TP + FN)$, respectively. AUC refers to the area under the ROC curve, which is a performance indicator to measure the pros and cons of a classifier. With these criteria, we can evaluate to what extent a method can solve the open-world binary classification problem.

3) Results: Figure 8 shows the TPR and FPR of different methods with a varying number of *unmonitored* DApps. In general, TPR and FPR of all the methods decrease with the increase of the number of unmonitored DApps. When the number of unmonitored DApps reaches 720, the TPR and FPR of GraphDApp tend to be stable, which are 0.99 and 0.05, respectively. In contrast, TPR of other methods still tends to decrease when the number of unmonitored DApps reaches to 1,260, as more flows of monitored DApps are mislabeled.



(a) True Positive Rate



(b) False Positive Rate

Fig. 8. Open-world: Impact of the Number of Unmonitored DApps on TPR and FPR.

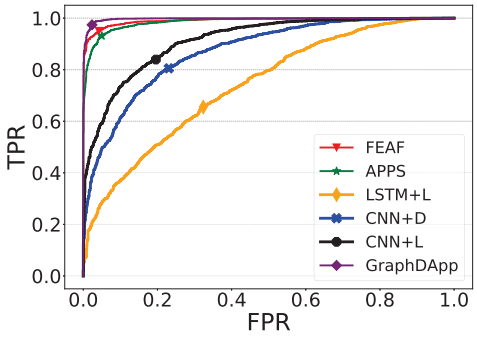


Fig. 9. Open-world: ROC Curves.

The reason is TIGs are stable and discriminative representations of DApp flows, which can accurately identify flows of monitored DApps from all the background flows, while the other representations (e.g., statistical features and packet length sequences) are not distinguishable enough to differentiate between monitored and unmonitored DApp flows. The results show that GraphDApp outperforms the rest methods, especially when the background noise (i.e., the unmonitored DApps) becomes larger.

Next, we fix the number of unmonitored DApps as 1,260 to evaluate the performance of different methods. In this set of experiments, the number of monitored and unmonitored flows are 32,000 and 14,000, respectively.

The ROC curve helps us to investigate the trade-offs between TPR and FPR for each classifier [18]. Taking GraphDApp for example, we can optimize GraphDApp either for high TPR or for low FPR. Figure 9 shows the ROC curves of different methods. The ROC curve of GraphDApp completely

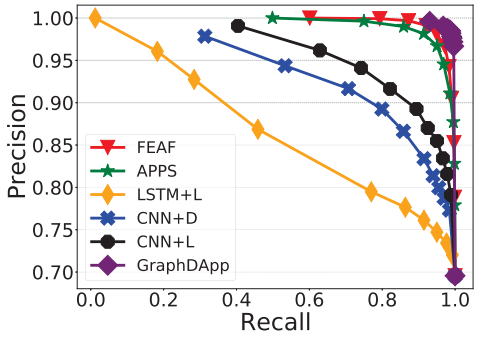


Fig. 10. Open-world: Precision-Recall Curves.

TABLE XI
OPEN-WORLD: AUC FOR STATE-OF-THE-ART METHODS

Method	FEAF	APPS	LSTM+L	CNN+D	CNN+L	GraphDApp
AUC	0.9908	0.9856	0.7648	0.8724	0.8993	0.9973

covers the curves of other methods, which can also be quantitatively measured by the AUC values in Table XI. The AUC value of GraphDApp is the largest among all the methods, which can reach 0.9973 and is quite close to the maximum value of 1.

Figure 10 presents the precision-recall curves for different methods in the open-world scenario. We calculate the precision and recall by varying the prediction threshold from 0.0 to 1.0 with a step of 0.1. This curve indicates that each classifier can be tuned to achieve different goals. For example, if we want to *reliably* recognize the monitored flows, a classifier can be optimized to achieve higher precision at the cost of lower recall; whereas if we aim at identifying as many potential visits to the monitored DApps as possible, the classifier can be parameterized to pursue higher recall.

As it can be seen from Fig. 10, GraphDApp is highly effective for any threshold. When the threshold is set to 0.1 (i.e., the recall of all methods is 1.0), the precision of GraphDApp can reach 0.96, while the precision of other methods is lower than 0.8. When we adjust the threshold to 1.0 to achieve high precision (i.e., approaching 1.0), the recall of GraphDApp can reach 0.93, which is much higher than that of other methods (all below 0.6). The precision-recall curve also demonstrates that GraphDApp outperforms the other methods.

F. Evaluation on Mobile Application Identification

Although GraphDApp is designed to identify encrypted traffic of specific DApps, it is still applicable to the classification of mobile applications, which is of great interest to recent studies [15], [24], [27]. To evaluate the generalizability of GraphDApp on mobile application classification, we conduct experiments using the datasets in our previous work [23]. This dataset contains the encrypted traffic of 15 popular mobile applications, such as Facebook, Twitter and Alipay. There are more than 54,000 flows in the datasets. Note that only dowstream flows are involved. As a closed-world evaluation, we also use *accuracy* as the metric to provide the overall performance of all methods.

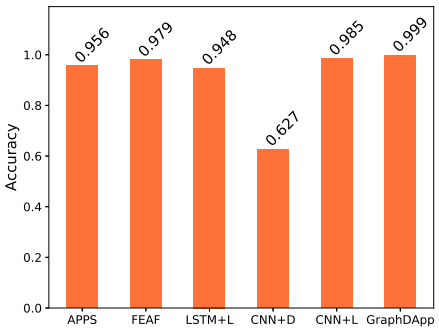


Fig. 11. Classification Results with Different Methods on Mobile Applications.

Figure 11 presents the classification accuracy with different methods. APPS, FEAF, LSTM+L, CNN+L have relatively high accuracy, indicating their features are discriminative enough to distinguish these mobile applications. Compared with the results shown in Table VIII, the accuracy of CNN+D significantly reduces to 0.627. The reason lies in that CNN can only extract limited information (i.e., one-way packet direction sequence) from the downstream flows in the dataset. The accuracy of GraphDApp reaches nearly 1.0. Although the graph structure of these flows reduces to sequential lines due to the absence of upstream packets, TIGs can also extract enough information for building powerful GNN-based classifiers. GraphDApp outperforms the other methods in terms of classification accuracy, indicating that GraphDApp can also be applied for mobile application fingerprinting.

VI. Discussion

There are mainly two limitations with our method. The *first* limitation is that GraphDApp needs a relatively long time to label an unknown flow. This can be solved by appropriately reducing the number of packets in TIGs to shorten feature extraction time, and employing less MLP layers and hidden units to accelerate the prediction speed. The *second* limitation is that, as a fingerprinting solution, when the fingerprints of an application changes, the accuracy will decrease accordingly. To address this problem, we can regularly update the TIGs of the application and fine-tune the parameters in the classifier. In the future work, we will further investigate techniques to make GraphDApp more adaptable to traffic changes.

VII. Conclusion

In this paper, we proposed GraphDApp, which can identify encrypted traffic of DApps using GNNs. We constructed the TIGs of encrypted flows based on the packet length and packet direction and turned the DApp identification into a graph classification problem. Then, we employed multi-layer perceptions to build a GNN-based classifier. Experiments are conducted to evaluate our method on real traffic datasets collected from DApps on Ethereum, including 40 monitored DApps and 1,260 unmonitored DApps. The experimental results show that GraphDApp can improve the accuracy by at least 10% over the state-of-the-art. We also demonstrated the effectiveness of GraphDApp on mobile application fingerprinting. In the future work, we plan to make GraphDApp more adaptable to traffic changes and further improve its time efficiency.

References

[1] *Ethereum*. Accessed: Oct. 1, 2019. [Online]. Available: https://www.ethereum.org/
[2] *State of the Dapps*. Accessed: Oct. 1, 2019. [Online]. Available: https://www.stateofthedapps.com/dapps?page=1
[3] *Countries and Territories*. Accessed: Oct. 5, 2019. [Online]. Available: https://freedomhouse.org/countries/freedom-world/scores
[4] *Matplotlib*. Accessed: Oct. 20, 2019. [Online]. Available: https://matplotlib.org/
[5] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau, "An exact graph edit distance algorithm for solving pattern recognition problems," in *Proc. Int. Conf. Pattern Recognit. Appl. Methods*, Lisbon, Portugal, vol. 1, Jan.2015, pp. 271–278.
[6] A. A. Niaki *et al.*, "ICLab: A global, longitudinal Internet censorship measurement platform," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 214–230.
[7] K. Al-Naami *et al.*, "Adaptive encrypted traffic fingerprinting with bi-directional dependence," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, Los Angeles, CA, USA , Dec. 2016, pp. 177–188.
[8] J. Cai, M. Fürer, and N. Immerman, "An optimal lower bound on the number of variables for graph identifications," *Combinatorica*, vol. 12, no. 4, pp. 389–410, 1992.
[9] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, "Analyzing Android encrypted network traffic to identify user actions," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 1, pp. 114–125, Dec. 2016.
[10] S. Feghhi and D. J. Leith, "A Web traffic analysis attack using only timing information," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 8, pp. 1747–1759, Aug. 2016.
[11] E. Grolman *et al.*, "Transfer learning for user action identication in mobile apps via encrypted trafc analysis," *IEEE Intell. Syst.*, vol. 33, no. 2, pp. 40–53, Mar. 2018.
[12] J. Hayes and G. Danezis, "K-fingerprinting: A robust scalable website fingerprinting technique," in *Proc. 25th USENIX Secur. Symp.*, Austin, TX, USA, vol. 16, Aug. 2016, pp. 1187–1203.
[13] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Netw.*, vol. 4, no. 2, pp. 251–257, 1991.
[14] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Netw.*, vol. 2, no. 5, pp. 359–366, Jan. 1989.
[15] M. Korczynski and A. Duda, "Markov chain fingerprinting to classify encrypted traffic," in *Proc. IEEE Conf. Comput. Commun.*, Toronto, ON, Canada, Apr. 2014, pp. 781–789.
[16] M. H. Mazhar and Z. Shafiq, "Real-time video quality of experience monitoring for HTTPS and QUIC," in *Proc. IEEE Conf. Comput. Commun.*, Honolulu, HI, USA, Apr. 2018, pp. 1331–1339.
[17] A. Panchenko *et al.*, "Website fingerprinting at Internet scale," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, Feb. 2016, pp. 1–5.
[18] S. Rezaei and X. Liu, "Deep learning for encrypted traffic classification: An overview," *IEEE Commun. Mag.*, vol. 57, no. 5, pp. 76–81, Dec. 2019.
[19] V. Rimmer, D. Preuveneers, M. Juarez, T. V. Goethem, and W. Joosen, "Automated website fingerprinting through deep learning," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, Feb. 2018, pp. 1–15.
[20] M. Shen, Y. Liu, S. Chen, L. Zhu, and Y. Zhang, "Webpage fingerprinting using only packet length information," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Shanghai, China, May 2019, pp. 1–6.
[21] M. Shen, Y. Liu, L. Zhu, X. Du, and J. Hu, "Fine-grained webpage fingerprinting using only packet length information of encrypted traffic," *IEEE Trans. Inf. Forensics Security*, early access, Dec. 23, 2020, doi: 10.1109/TIFS.2020.3046876.
[22] M. Shen, Y. Liu, L. Zhu, K. Xu, X. Du, and N. Guizani, "Optimizing feature selection for efficient encrypted traffic classification: A systematic approach," *IEEE Netw.*, vol. 34, no. 4, pp. 20–27, Jul. 2020.
[23] M. Shen, M. Wei, L. Zhu, and M. Wang, "Classification of encrypted traffic with second-order Markov chains and application attribute bigrams," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 8, pp. 1830–1843, Aug. 2017.

[24] M. Shen, M. Wei, L. Zhu, M. Wang, and F. Li, "Certificate-aware encrypted traffic classification using second-order Markov chain," in *Proc. IEEE/ACM 24th Int. Symp. Qual. Service (IWQoS)*, Beijing, China, Jun. 2016, pp. 1–10.
[25] M. Shen, J. Zhang, L. Zhu, K. Xu, X. Du, and Y. Liu, "Encrypted traffic classification of decentralized applications on ethereum using feature fusion," in *Proc. Int. Symp. Qual. Service*, Phoenix, AZ, USA, Jun. 2019, p. 18.
[26] P. Sirinam, M. Imani, M. Juarez, and M. Wright, "Deep fingerprinting: Undermining website fingerprinting defenses with deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Toronto, ON, Canada, Oct. 2018, pp. 1928–1943.
[27] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "AppScanner: Automatic fingerprinting of smartphone apps from encrypted network traffic," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Saarbrücken, Germany, Mar. 2016, pp. 439–454.
[28] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg, "Effective attacks and provable defenses for website fingerprinting," in *Proc. 23rd Secur. Symp.*, San Diego, CA, USA, Aug. 2014, pp. 143–157, 2014.
[29] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *7th Int. Conf. Learn. Represent.*, New Orleans, LA, USA, May 2019, pp. 1–4.

Meng Shen (Member, IEEE) received the B.Eng. degree from Shandong University, Jinan, China, in 2009, and the Ph.D. degree from Tsinghua University, Beijing, China, in 2014, all in computer science. He is currently an Associate Professor with the Beijing Institute of Technology, Beijing. His research interests include data privacy and security, blockchain applications, and encrypted traffic classification. He has authored over 50 papers in top-level journals and conferences, such as ACM SIGCOMM, IEEE JSAC, and IEEE TIFS. He has guest edited special issues on emerging technologies for data security and privacy in IEEE NETWORK and IEEE INTERNET-OF-THINGS JOURNAL. He received the Best Paper Runner-Up Award at IEEE IPCCC 2014 and IEEE/ACM IWQoS 2020. He was selected by the Beijing Nova Program 2020 and was the winner of the ACM SIGCOMM China Rising Star Award 2019.

Jinpeng Zhang received the B.Eng. degree in computer science from Northwest A&F University, Shanxi, China, in 2017. He is currently pursuing the master's degree with the Department of Computer Science, Beijing Institute of Technology. His research interests include anonymity networks and traffic analysis.

Liehuang Zhu (Member, IEEE) is currently a Professor with the Department of Computer Science, Beijing Institute of Technology. He is selected into the Program for New Century Excellent Talents in University from Ministry of Education, P.R. China. His research interests include the Internet of Things, cloud computing security, Internet and mobile security.

Ke Xu (Senior Member, IEEE) received the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, where he serves as a Full Professor. He has published more than 200 technical papers and holds 11 U.S. patents in the research areas of next-generation Internet, blockchain systems, the Internet of Things (IoT), and network security. He is a member of ACM. He has guest-edited several special issues in IEEE and Springer Journals. He is an Editor of IEEE IoT JOURNAL. He is also the Steering Committee Chair of IEEE/ACM IWQoS.

Xiaojiang Du (Fellow, IEEE) received the B.S. and M.S. degrees in electrical engineering from Tsinghua University, Beijing, China, in 1996 and 1998, respectively, and the M.S. and Ph.D. degrees in electrical engineering from the University of Maryland College Park, in 2002 and 2003, respectively. He is a tenured Professor with the Department of Computer and Information Sciences, Temple University, Philadelphia, USA. His research interests are wireless communications, wireless networks, security, and systems. He has authored over 400 journal and conference papers in these areas, as well as a book published by Springer. He has been awarded more than $5 million US dollars research grants from the US National Science Foundation (NSF), Army Research Office, Air Force, NASA, the State of Pennsylvania, and Amazon. He won the best paper award at IEEE GLOBECOM 2014 and the best poster runner-up award at the ACM MobiHoc 2014. He serves on the editorial boards of three international journals. He is a Life Member of ACM.