AppSniffer: Towards Robust Mobile App Fingerprinting Against VPN

Sanghak Oh
Department of Electrical and Computer Engineering,
Sungkyunkwan University
sanghak@skku.edu

Minwook Lee
Department of Electrical and Computer Engineering,
Sungkyunkwan University
mwlee@skku.edu

Hyunwoo Lee
Department of Energy Engineering,
Korea Institute of Energy Technology (KENTECH)
hwlee@kentech.ac.kr

Elisa Bertino
Department of Computer Science,
Purdue University
bertino@purdue.edu

Hyoungshick Kim
Department of Electrical and Computer Engineering,
Sungkyunkwan University
hyoung@skku.edu

ABSTRACT

Application fingerprinting is a useful data analysis technique for network administrators, marketing agencies, and security analysts. For example, an administrator can adopt application fingerprinting techniques to determine whether a user's network access is allowed. Several mobile application fingerprinting techniques (e.g., FlowPrint, AppScanner, and ET-BERT) were recently introduced to identify applications using the characteristics of network traffic. However, we find that the performance of the existing mobile application fingerprinting systems significantly degrades when a virtual private network (VPN) is used. To address such a shortcoming, we propose a framework dubbed AppSniffer that uses a two-stage classification process for mobile app fingerprinting. In the first stage, we distinguish VPN traffic from normal traffic; in the second stage, we use the optimal model for each traffic type. Specifically, we propose a stacked ensemble model using Light Gradient Boosting Machine (LightGBM) and a FastAI library-based neural network model to identify applications' traffic when a VPN is used. To show the feasibility of AppSniffer, we evaluate the detection accuracy of AppSniffer for 150 popularly used Android apps. Our experimental results show that AppSniffer effectively identifies mobile applications over VPNs with F1-scores between 84.66% and 95.49% across four different VPN protocols. In contrast, the best state-of-the-art method (i.e., AppScanner) demonstrates significantly lower F1-scores between 25.63% and 47.56% in the same settings. Overall, when normal traffic and VPN traffic are mixed, AppSniffer achieves an F1-score of 90.63%, which is significantly better than AppScanner that shows an F1-score of 70.36%.

CCS CONCEPTS

• **Security and privacy** → **Mobile and wireless security**; • **Information systems** → *Traffic analysis*; • **Computing methodologies** → Artificial intelligence.

KEYWORDS

App fingerprinting, Traffic analysis, VPN, Mobile app

ACM Reference Format:
Sanghak Oh, Minwook Lee, Hyunwoo Lee, Elisa Bertino, and Hyoungshick Kim. 2023. AppSniffer: Towards Robust Mobile App Fingerprinting Against VPN. In *Proceedings of the ACM Web Conference 2023 (WWW '23), April 30–May 04, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3543507.3583473

1 INTRODUCTION

With the growing number of mobile devices, mobile applications (hereinafter referred to as *apps*) contribute to over half of the global web traffic [1]. More than 6.5 billion people are using their hands-on devices [9]. Therefore, it becomes a commonplace to use own mobile devices within enterprise networks [22], indicating that techniques and tools are necessary for managing network traffic generated from mobile apps.

Mobile app fingerprinting is one of the building blocks for managing and securing networks [46]. It is used to identify mobile apps based on traffic they generate, such as network addresses or application data. For security purposes, Internet Service Providers (ISPs) or enterprise network administrators can use this technique to identify specific apps and allow/block their network accesses according to the network's policies. To this end, there have been several studies [27, 43] to identify apps with high accuracy by fingerprinting based on application data (e.g., HTTP headers). However, such approaches are unusable as Transport Layer Security (TLS) [42] becomes widespread [30]; thus, it is challenging to identify mobile apps over encrypted traffic. To address the challenge, three types of mobile app fingerprinting techniques have been proposed, differing with respect to the information they use for generating fingerprints. Techniques of the first type, referred to as plaintext metadata-based techniques [8, 35, 45, 47], generate fingerprints based on plaintext metadata, such as certificates or IP addresses. Techniques of the second type, referred to as networking pattern-based techniques [37, 41, 46, 51], make fingerprints based on networking patterns like inter-arrival time between packets. Techniques of the third type, referred to as raw packet-based techniques [31–33], create fingerprints based on raw packets.

However, a significant drawback of those techniques is that their accuracy performance notably degrades when a virtual private network (VPN) is used, as we have experimentally verified (results are

WWW '23, April 30–May 04, 2023, Austin, TX, USA

ACM ISBN 978-1-4503-9416-1/23/04…$15.00
https://doi.org/10.1145/3543507.3583473

shown later in the paper). Since VPN protocols usually append an additional header and a message authentication code to packets, network patterns over a VPN are quite different from an app's network patterns when no VPN is used, making apps hard to detect. Consequently, malware can use a VPN to evade detection by fingerprinting systems that rely on network addresses and traffic patterns, thus adding another evasion mechanism to the ones already in use by malware [32, 46, 47].

In this paper, we present AppSniffer, a framework aimed at identifying blacklisted or suspicious apps, instead of blocking all VPN traffic. This approach is taken to avoid any negative impact on usability, such as the inability to use voice services like WiFi calling, which rely on VPNs. AppSniffer is designed to identify mobile apps, regardless of whether they are running with or without a VPN connection. AppSniffer uses a two-stage approach for app identification. In the first stage, it differentiates VPN traffic from normal traffic by using a simple but effective classifier based on the number of flows in the traffic. In the second stage, AppSniffer identifies the specific mobile apps from VPN traffic and normal traffic, based on the optimal model for each traffic type. We use FlowPrint [47] for normal traffic because it produces the best F1-score results. However, we develop a novel classifier, called ScanVPN, for VPN traffic because there is no effective model to identify mobile apps over VPNs. A sequence of packet lengths is the key feature for identifying mobile apps over a VPN because packet sizes do not dynamically change in most mobile apps, even when a VPN is used. To improve the performance of the mobile app fingerprinting, we build a stacked ensemble model with two machine learning models, namely Light Gradient Boosting Machine (LightGBM) [24] and a FastAI [20] library-based neural network. With ScanVPN, AppSniffer introduces a specific classifier per known VPN protocol (e.g., OpenVPN over TCP or WireGuard over UDP) trained on the corresponding VPN traffic. Then, for given VPN traffic, AppSniffer infers the VPN protocol, e.g., by inspecting the headers, used for the traffic and sends it to the corresponding classifier. Finally, the classifier identify mobile apps from the traffic. Note that we also provide an automated script to generate VPN traffic to train a classifier of known VPN protocol to make AppSniffer easily deployable.

Our key contributions are summarized as follows:

- We review the state-of-the-art mobile app fingerprinting techniques [32, 46, 47] over VPNs and analyze the root causes of the ineffectiveness of those techniques in identifying mobile apps over VPNs.
- We propose a framework dubbed AppSniffer to identify mobile apps regardless of whether a VPN is used as the underlying network. Also, we develop a novel classification model for app recognition over VPNs.
- To show feasibility of the framework, we collect network traffic from 150 popularly used Android apps with an automated script and implement AppSniffer.
- We conduct a comprehensive experiment on the systems and show that AppSniffer achieves F1-score of 90.63%, significantly better than the state-of-the-art methods when normal traffic and VPN traffic are mixed.

We release our dataset and source code in the public repository; they are available at https://github.com/network-traffic/AppSniffer.

2 BACKGROUND

This section presents background information on VPN protocols and mobile app fingerprinting techniques.

2.1 VPN Protocols

A VPN technique connects two entities across the Internet as if they were connected within a private network, leveraging encrypted tunneling schemes. VPN techniques are widely used for various reasons. For example, one may use a VPN to protect one's privacy or circumvent censorship. One can also establish a VPN with a company gateway to access the enterprise network from outside. VPN networks are also established between smartphones and mobile networks when users get call services [29].

A VPN protocol specifies 1) how to establish encrypted tunnels and 2) how to exchange data over the tunnels. Two parties, a VPN client and a VPN server, are involved in the protocol. A VPN client selects a VPN server and initiates the protocol to establish an encrypted tunnel with the VPN server. After the tunnel is established, to send packets to a public server (e.g., Google), the VPN client just needs to encrypt the packets, including application data as well as TCP/IP headers, and encapsulate the encrypted packets with the tunnel headers. The tunnel headers contain information about the encrypted packets (e.g., length), the VPN server's IP address, and port number. Finally, the VPN client sends the encapsulated packets to the VPN server. The VPN server first decapsulates the tunnel headers from the packets, decrypts the resulting packets, and forwards the decrypted packets to the public server.

In what follows, we first describe the PPTP VPN protocol [19] and then describe three widely used VPN protocols, namely IKEv2 [23]/IPsec [26], WireGuard [11], and OpenVPN.

PPTP-based VPN protocol. PPTP is a protocol to establish a tunnel that carries point-to-point protocol (PPP) [44] packets. In PPTP, an IP tunnel is established by using the GRE tunneling protocol [14], and the PPP packets are exchanged over the tunnel. As the GRE tunnel does not provide any security service, the Microsoft Point-To-Point Encryption (MPPE) protocol [39] is executed to negotiate the use of the RC4 encryption algorithm with a session key. SuperVPN[1] is a representative example using PPTP.

IKEv2/IPsec-based VPN protocol. In this type of VPN protocol, a tunnel is established after running the IKEv2 protocol [23] that relies on the Diffie-Hellman (DH) key exchange to establish a session key. Then, the Encapsulated Security Payload (ESP) protocol [25] is used to encrypt packets and assure their integrity with the Hash-based Message Authentication Code (HMAC). Note that ESP is over UDP. TurboVPN[2] is one of the solutions using IKEv2/IPsec.

OpenVPN protocol. In OpenVPN, a VPN client and its VPN server establish a session key using the TLS handshake protocol. The key is used to encrypt/decrypt raw IP packets in the data channel where both TCP and UDP can be used for its transport layer. In other words, the data channel is established as either TCP or UDP. Surfshark[3] is a representative example using OpenVPN.

WireGuard protocol. In WireGuard [11], a VPN client and its VPN server perform the elliptic curve-based DH key exchange to establish a session key. A session key is used to encrypt packets with an authenticated encryption scheme. Then, encrypted packets are sent over UDP. NordVPN[4] uses WireGuard.

[1] http://www.supervpn.net
[2] https://turbovpn.com
[3] https://surfshark.com
[4] https://nordvpn.com

Table 1: Performance of ET-BERT before and after excluding TSval and TSecr (in TCP option field) from the datasets in [32].

Dataset	ET-BERT (flow)				ET-BERT (flow) w/o TSval and TSecr			
	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
Cross-Platform (iOS) [47]	0.9882	0.9898	0.9885	0.9898	0.9117	0.9069	0.9055	0.9069
Cross-Platform (Android) [47]	0.9920	0.9913	0.9910	0.9913	0.7340	0.7666	0.7180	0.7666
ISCX-VPN-Service [12]	0.8538	0.8418	0.8463	0.8418	0.8402	0.8356	0.8365	0.8356
ISCX-VPN-App [12]	0.7655	0.7754	0.7637	0.7754	0.7316	0.6890	0.6840	0.6890
ISCX-Tor [28]	0.8552	0.8656	0.8555	0.8656	0.8552	0.8656	0.8555	0.8656
USTC-TFC [49]	0.9755	0.9803	0.9774	0.9803	0.9566	0.9633	0.9593	0.9633

Takeaway. We should note that once a VPN is used, VPN traffic is different from original traffic because the structure of packets is changed, especially in the following two aspects. First, the transport layer protocol might be changed. As most of the VPN protocols use UDP for its transport layer protocol, TCP/IP packets change to UDP/IP packets. Second, packet lengths increase. This is because the VPN protocols always prepend/append headers/trailers for each packet. For instance, if there is a 80 byte packet (40 byte for TCP/IP headers and the other for application data), the encapsulated packet for data would be 112 bytes (PPTP) and 140 bytes (WireGuard), respectively without optional fields and paddings.

2.2 Mobile App Fingerprinting Techniques

Depending on the features that fingerprinting techniques use, there are three types of techniques as follows:

Plaintext metadata-based techniques. These systems [8, 35, 45, 47] extract plaintext metadata (e.g., header fields of network packets) from network traffic and transform them into a form of a fingerprint. For example, FlowPrint [47] generates a fingerprint of a mobile app based on a cluster of destination IP addresses with which each mobile app communicates. The rationale of the approach is that each app communicates with a limited number of destination servers, and this pattern can properly represent the app itself. FlowPrint examines a set of destination IP addresses that a target mobile app communicates with and compares the set with known mobile apps' fingerprints to find the one where the highest number of destination IP addresses overlap.

Networking pattern-based techniques. The systems in this category [37, 41, 46, 51] extract networking patterns from the traffic and build a fingerprint based on them. Examples of features used for a fingerprint are the inter-arrival time between packets and packet lengths. For instance, AppScanner [46], a representative of networking pattern-based systems, makes a fingerprint of a mobile app based on a sequence of lengths of packets that the app generates. A classifier is generated based on fingerprints extracted from a training dataset and is used to identify a mobile app over the traffic.

Raw packet-based techniques. This type of systems [31–33] use raw packets generated by a mobile app as a fingerprint. For example, ET-BERT [32], a state-of-the-art system showing a high accuracy, extracts raw payload bytes from each packet and performs pre-training and fine-tuning of the BERT model based on the payload data. Surprisingly, ET-BERT shows that statistically meaningful patterns exist in both encrypted and plaintext payloads that allow one to identify mobile apps.

3 LIMITATIONS OF FLOW-BASED MOBILE FINGERPRINTING SYSTEMS AGAINST VPN

In this section, we discuss the limitation of each category of mobile fingerprinting systems when we consider an attacker that uses a VPN on its device to communicate with other parties.

Plaintext metadata-based techniques. The systems in this category would be ineffective in identifying mobile apps over a VPN because the plaintext features of traffic are changed when using the VPN. For instance, a fingerprint generated by FlowPrint [47] for the attacker's traffic would form a cluster of VPN server IP addresses instead of the original servers' IP addresses; thus, it would not match any known fingerprints computed without VPNs. Our experiments show that the F1-score of FlowPrint, trained only with normal traffic, is 0% when dealing with a fingerprint of an attacker because FlowPrint does not know the VPN server IP addresses (see Section 5.4).

Networking pattern-based techniques. Even though the networking patterns can be captured from VPN traffic, they might significantly differ from the patterns extracted from normal traffic because using a VPN can affect networking patterns (e.g., increased packet length or fragmentation) by encapsulating network packets. Consequently, the systems become inefficient when they attempt to identify mobile apps based on the patterns extracted from an attacker's traffic. We show that AppScanner [46] achieves an F1-score of 0% due to the VPN (see Section 5.4).

Raw packet-based techniques. Once an attacker uses a VPN tunnel, the form of packets over a VPN tunnel changes from the form of the original packets. For instance, TCP/IP packets can be changed to UDP/IP packets if the attacker selects the option of using a UDP tunnel to send the packets. This change can significantly affect the performance of the fingerprinting systems. For instance, we show that ET-BERT [32] fails to identify mobile apps when each app communicates in a VPN tunnel; ET-BERT achieves an F1-score of nearly 0% (see Section 5.4). We used a feature importance analysis on the Cross-Platform datasets [47] to analyze the reasons for ET-BERT's performance degradation for VPN traffic (see Appendix A). We found that Timestamp Value (TSval) and Timestamp Echo Reply (TSecr) [21] in the TCP option fields are the most important features that may affect the performance of mobile app fingerprinting. That is, ET-BERT's performance can significantly degrade when we exclude those fields (see Table 1). For example, in the Cross-Platform (Android) dataset [47], an F1-score of 0.9910 was reduced to 0.7180. Therefore, ET-BERT would fail to work on packets over a UDP tunnel because those optional fields are not used in such situations.

4 OVERVIEW OF APPSNIFFER

To address the limitation of the existing mobile app fingerprinting systems against VPN, we propose a framework dubbed AppSniffer to identify mobile apps based on network traffic regardless of whether they are being encapsulated over VPNs or not. In what follows, we first introduce our threat model and provide an overview of AppSniffer. Then, we present in detail each component of it.

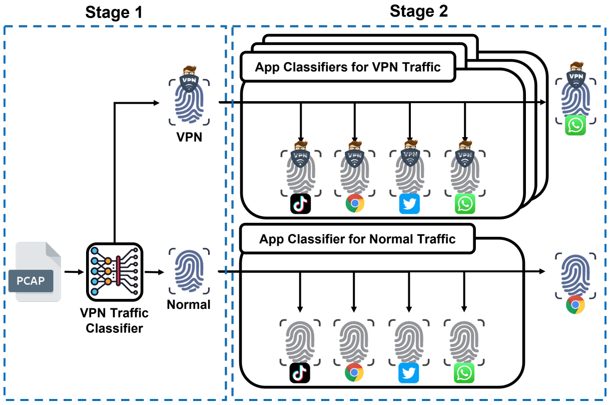
Stage 1
Stage 2
App Classifiers for VPN Traffic
VPN
PCAP
VPN Traffic Classifier
Normal
App Classifier for Normal Traffic

Figure 1: AppSniffer framework overview.

4.1 Threat Model

In our work, we use the same threat model used in previous mobile app fingerprinting studies [41, 47]. We assume that an app fingerprinting system is located at the edge of a network that the system aims to protect. The system can trace back traffic in its network but cannot inspect the traffic in other networks. In addition, we assume that the system can identify a mobile device that produces network traffic through MAC addresses or mobile device identification techniques [6, 40]. Lastly, we focus on a single app fingerprint. In other words, we assume that only packets from one mobile app are passed through the network at a given time.

The goal of an attacker in the protected network is to secretly communicate with its server by hiding the identity of the app that the attacker uses from mobile app fingerprinting systems. To achieve this goal, we assume that the attacker can only leverage network-level solutions such as VPN or a proxy but cannot directly manipulate the app traffic, which would be expensive.

4.2 High Level Description of AppSniffer

Our framework, AppSniffer, aims to precisely identify mobile apps when a VPN or a proxy is used to hide their identities. That is, AppSniffer recognizes apps with a high accuracy using their network traffic regardless of whether it is being sent over a VPN or not. Hereinafter, if the traffic is sent over a VPN, we call the traffic *VPN traffic*; otherwise, we call it *normal traffic*.

The main challenge is how to recognize mobile apps from VPN traffic. We observe that when a mobile app runs over a VPN, the traffic the app generates is quite different from that the app generates in a normal scenario. Therefore, a mobile app running over a VPN should be considered independently of the original app in a normal scenario. To address this issue, one may suggest using a multi-class classifier that identifies each mobile app with respect to given traffic. However, we surmise that such an approach would not work since patterns of VPN traffic depend not only on the networking patterns of the mobile apps but also on the VPN protocol in use (e.g., OpenVPN over TCP and WireGuard over UDP). Therefore, we do not combine the traffic of all different VPN protocols for a single classification model. Instead, we build the optimized classification model for each VPN protocol individually.

Figure 1 shows the design of AppSniffer consisting of two stages. For AppSniffer, we need to build $k + 1$ classifiers where one is for normal traffic and the other k classifiers are for k known VPN protocols, respectively. In the first stage, AppSniffer classifies whether some given traffic is VPN traffic or not. Then, in the second stage, AppSniffer first identifies the VPN protocol in use to select the optimal classification model for that protocol. When one of k known VPN protocols is used, AppSniffer can accurately identify the VPN protocol in use because of the packet header and VPN servers' IP addresses. Finally, AppSniffer identifies mobile apps using the optimal classification model for the protocol in use. The rationale behind this two-stage approach is to consider underlying VPN protocols that can significantly affect network traffic patterns of mobile apps in developing classifiers while getting benefits from existing approaches on normal traffic. Therefore, we introduce three components in AppSniffer, namely: 1) *VPN traffic classifier* used in the first stage, 2) k different *app classifiers for VPN traffic* and 3) *app classifier for normal traffic* used in the second stage.

In the second stage, AppSniffer cannot identify which VPN protocol is specifically used when an unknown VPN protocol is used. In such situations, AppSniffer only labels the given traffic as *unknown VPN traffic* but cannot provide the app's identity in detail.

4.3 Components of AppSniffer in Detail

Below we describe the three classifiers in detail.

VPN traffic classifier. The main purpose of the *VPN traffic classifier* is to determine whether some given traffic is VPN traffic or not. Our approach for this task is to focus on *the number of flows* generated by a mobile app, where a flow is defined as a five-tuple indicating the protocol in use, source/destination IP addresses, and source/destination ports. Mobile apps generally communicate with various destination servers (e.g., CDN servers and ad servers). However, if an attacker runs mobile apps over a VPN, the mobile traffic, encapsulated with the same IP addresses and ports, results in a single flow. Therefore, we develop a VPN traffic classifier that labels some given traffic into VPN traffic if the traffic contains only one flow; otherwise, the classifier labels it as normal traffic. We verify the feasibility of our approach by counting the number of flows from VPN traffic and normal traffic generated by a mobile app. Our analysis of 150 apps shows that all the VPN traffic consists of one flow, while the normal traffic has 33.53 flows on average (the minimum is 2 and the maximum is 353).

App classifiers for VPN traffic (C_{VPN}). These classifiers aim to recognize a mobile app over VPN traffic. Designing such classifiers requires addressing the following three challenges. First, we should decide how to prepare a training *dataset* for each classifier. Second, we should consider which *features* to use for identifying mobile apps. Finally, we should determine how to select a good *algorithm* with appropriate hyper-parameters for each classifier.

Our approach to identifying mobile apps over VPNs is to build the optimal classifier for each VPN protocol so that the classifier can label given traffic with the corresponding mobile app. A challenge of this approach is the lack of public datasets of VPN traffic generated by mobile apps. Therefore, we implement a script to collect VPN traffic from mobile apps in a scalable manner.

Based on the dataset, we select a sequence of packet lengths in the communication flows as the main feature. We also consider the direction of a packet by prepending '+' or '-' to each length in the sequence. For example, consider a VPN client that sends a 100-byte packet to a VPN server that responds with a 200-byte packet. We describe the sequence of packet lengths as $(+100, -200)$. We refer to the Adjusted Mutual Information (AMI) analysis [47], where a packet length is one of the highest-scored features. Other features in the top 10 ones are source/destination IP addresses and the averaged time interval between flows. However, we found those features inappropriate for our task because VPN server IP addresses are used instead of the original servers' IP addresses, and the flows are combined. To consider the context of a mobile app, we focus on a sequence of packets, not a single packet, for fingerprinting.

To select an appropriate model for our task, we use AutoGloun [13], the state-of-the-art Automated Machine Learning tool that automatically finds the best model with optimal hyper-parameters based on a given dataset. It searches a best-fit classifier from 8 types of models[5] and ensemble models. In ensembling, AutoGluon also supports bagging and multi-layer stacking. Finally, for VPN traffic, we develop the app classifier dubbed ScanVPN, which consists of a stacked ensemble model using Light Gradient Boosting Machine (LightGBM) and a FastAI library-based neural network.

App classifier for normal traffic (C_{Normal}). After passing through the VPN traffic classifier, normal traffic is fed to the app classifier for normal traffic. As existing mobile app fingerprinting systems (e.g., FlowPrint [47] or AppScanner [46]) show high accuracy, we leverage the best existing system for this component.

5 EXPERIMENTS

This section describes how we prepare the datasets used in our experiment and how we choose training and testing datasets to show the limitations of existing mobile app fingerprinting systems and evaluate the performance of AppSniffer compared with other systems.

5.1 Dataset

For our experiments, we generate a dataset of VPN and normal traffic from 100 mobile apps selected from the top-200 Android apps listed on the Google Play Store in the US[6] (details can be found in Appendix B). The apps are chosen when they generate enough flows to be identified by the state-of-the-art app fingerprinting systems (i.e., FlowPrint and AppScanner). Each of the 100 selected apps is run 50 times, with the traffic generated recorded in each trial, each lasting 20 seconds, as determined by our preliminary study (see Section 5.2). To increase the diversity of the dataset, we also collected additional 50 samples of 20-second-long traffic per app, generated by simulating user interactions using Monkey [7].

We collect VPN traffic from mobile apps running four VPN clients – SuperVPN, TurboVPN, NordVPN, and Surfshark – that execute different VPN protocols. We select the VPN clients based on the number of downloads in Google Play Store. Information about the four VPN clients is given in Table 2. We summarize our VPN datasets with statistics on the number of packets in each VPN flow in Table 3. We observe that for a unit of time, the flows in the TurboVPN dataset have the highest number of packets, and those on the Surfshark dataset have the least number of packets. We surmise that difference in the transport layer might be a reason since UDP communications are typically faster than TCP [7].

Table 2: Selected VPN apps.

VPN Provider Name	Version	# Download	VPN Type
SuperVPN	2.7.2	> 100 million	Free Provider
TurboVPN	3.7.4.2	> 100 million	Paid Provider
NordVPN	5.11.5	> 50 million	Paid Provider
Surfshark	2.8.1.8	> 50 million	Paid Provider

Table 3: Number of packets in each VPN flow.

Dataset	Transport	Mean (Std)	Median	Min	Max
SuperVPN	UDP	3412.36 (5990.19)	1469	28	57203
TurboVPN	UDP	4219.88 (9341.60)	1345	24	65758
NordVPN	UDP	2703.10 (4516.52)	1072	40	51355
Surfshark	TCP	1641.93 (3666.92)	442	38	35393

5.2 Experiment Setup

Time series cross-validation. To assess the performance of FlowPrint, AppScanner, and ET-BERT on VPN traffic and normal traffic, we first split samples of each app into five folds to perform cross-validation in time series. Our purpose is to see how the performance of a system changes as the size of training samples increases. Finally, we prepare three different scenarios called CV1, CV2, and CV3 (see Figure 2).

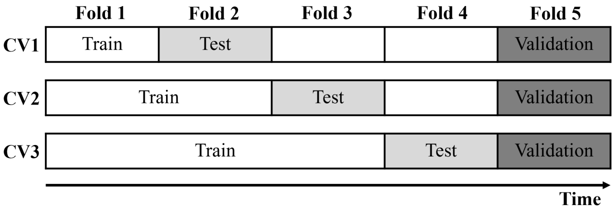

Figure 2: Five-fold time series cross-validation.

For metrics, we use four typical metrics (precision, recall, F1-score, and accuracy) to evaluate the performance of FlowPrint, AppScanner, and ET-BERT. Micro-average is used [18], which calculates the proportion-weighted mean value of each metric for the individual apps (see Equation 1) to handle imbalanced sizes of multiple classes.

$$score_{micro-avg} = W_0 \cdot score_{class_0} + \cdots + W_n \cdot score_{class_n} \quad (1)$$

Optimal execution time for app recognition. To collect apps' traffic, we face the challenge of determining the appropriate amount of time for the execution time as it may affect the performance of the systems. Therefore, we conduct experiments to find the optimal execution time when the performance of the systems becomes stable. To this end, we evaluate FlowPrint and AppScanner with respect to the F1-score of recognizing app, varying the execution time to 5, 10, 15, 20, 25, and 30 seconds respectively, which we report the results in Table 4. We find that the performance of the systems becomes stable after 20 seconds. Therefore, we decide to use 20 seconds for the execution time per sample.

Table 4: Summary of tested execution time optimization.

Execution Time	FlowPrint F1-score	AppScanner F1-score
5 sec	0.9642	0.7372
10 sec	0.9874	0.7385
15 sec	0.9826	0.7314
20 sec	0.9712	0.7953
25 sec	0.9651	0.7884
30 sec	0.9626	0.7805

[5] Random Forest, XGBoost, CatBoost, kNN, Logistic Regression, Light Gradient Boosting Machine (LightGBM), ExtraTrees, and Tabular Neural Network

[6] We select mobile apps from the list in accordance with Google's app ranking system on 21-December-2021

[7] https://developer.android.com/studio/test/other-testing-tools/monkey

Table 5: Comparison results of classifiers for normal traffic. NN stands for neural network, RF for Random Forest.

Dataset		LightGBM + FastAI NN + RF + CatBoost				FlowPrint				AppScanner				ET-BERT			
		Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
Normal	CV1	0.8426	0.8310	0.8348	0.8310	0.9815	**0.9756**	**0.9756**	**0.9756**	**0.9832**	0.5317	0.6707	0.5317	0.2205	0.1888	0.1637	0.1888
	CV2	0.8706	0.8584	0.8630	0.8584	**0.9878**	**0.9839**	**0.9843**	**0.9839**	0.9844	0.6198	0.7462	0.6198	0.5659	0.4442	0.4657	0.4442
	CV3	0.8796	0.8675	0.8719	0.8675	0.9822	**0.9754**	**0.9767**	**0.9754**	**0.9884**	0.6566	0.7766	0.6566	0.5413	0.4497	0.4692	0.4497

Data pre-processing. We perform pre-processing in two aspects. First, we remove background traffic (e.g., Dynamic Host Configuration Protocol (DHCP), Address Resolution Protocol (ARP)) that is not specific to apps after collecting the samples. Second, we mask values in some fields (e.g., timestamp) that can be specifically determined in experimental settings. We found that ET-BERT achieved high accuracy relying on Timestamp Value (TSval) and Timestamp Echo Reply (TSecr) [21] in the TCP option fields. However, such high accuracy is mainly due to the similar timestamps in the training and testing sets, which would be unacceptable in real-world environments. Other than TSval and TSecr, we also mask the Security Parameter Index (SPI) field in the header of Encapsulating Security Payload (ESP) in the TurboVPN (i.e., IPsec-based VPN protocol) dataset. SPI is an index to search for the specific security association (SA) that contains information about encryption algorithms and keys. The SPI value is randomly generated whenever the VPN tunnel is established. However, in some cases, the same SPI value is repeatedly used across different samples, which should not happen in practice. Therefore, we conclude that SPI is not an appropriate feature; thus, we also mask all the SPI values.

Experiment environment. For model training, we use a GPU server that consists of a Tesla V100-PCIe with 34GB memory and an Intel Xeon(R) E5-2687w v3 @3.10 GHz with 264GB RAM. We use a Google Pixel 4 phone for our experiments.

5.3 Implementation

Dataset generation script We implement a script for generating the VPN traffic for desired apps in an automated way. The script is written in Python 3 using bash shell commands. It works as follows: with Android Debug Bridge (ADB) [8], the script automatically installs, executes, and uninstalls apps. During the app execution, with tshark [9] (the terminal version of Wireshark [10]), the script captures the app traffic, while the mobile device is connected to the Internet.

VPN traffic classifier We implement the classifier using Python 3 with the tshark package to parse characteristics of flows from traffic files (i.e., Pcap files).

App classifier for VPN traffic We implement the classifier using Python 3 with AutoGluon [11] packages. For model selection, our strategy is to find the best model and optimize it to improve the F1-score. We also set up a fitting time of modeling as 10 minutes since the model fitting with unlimited time setup takes too much time, but the performance is not much higher than that of models with our fitting time.

5.4 Evaluation

In this section, we evaluate the performance of AppSniffer in mobile app identification with regard to VPN traffic and normal traffic. Our experimental results are shown in Table 5, Table 6, and Table 7 where the best results are highlighted in bold.

Optimization of AppSniffer. As AppSniffer allows using existing techniques for the app classifier for normal traffic (C_{Normal}), we experiment to understand the impact of different types of classifiers on the performance of AppSniffer. As classifiers, we use FlowPrint, AppScanner, and ET-BERT. In addition to them, we also build our own model using AutoGloun [13], which is generated as a stacked ensemble model consisting of LightGBM, a FastAI library-based neural network, Random Forest (RF), and CatBoost, which is similar to ScanVPN but trained on normal traffic. Note that ScanVPN was originally designed for identifying apps over VPNs where each app generates only one flow unless the VPN server is changed. Therefore, the flow can represent the corresponding app. However, in a normal scenario, each app generates several flows during execution. Therefore, we should choose a representative flow to identify the corresponding app. Our selection is a flow containing the maximum number of packets among the flows the app generates. Then, we use a sequence of packet lengths of the representative flow to build our model for normal traffic.

Table 5 shows that FlowPrint is the best in that it achieves the highest F1-score of 97.67% compared to the others (12.02% higher than the second), indicating that a set of destination IP addresses is a highly useful feature for identifying the apps for non-VPN traffic. Finally, we select FlowPrint for the app classifier for normal traffic (C_{Normal}). Note that the model built by AutoGloun shows the second-best performance, indicating that a sequence of packet lengths in the longest flow (i.e., the flow in which the most number of packets are included) can also be a useful feature for identifying mobile apps.

On the other hand, for the app classifier for VPN traffic (C_{VPN}), we evaluate ScanVPN and the existing app fingerprinting systems. Here, the models are all trained based on VPN traffic. For the experiment, we slightly modify the source code of AppScanner to make it work on UDP traffic because it originally focuses on HTTPS (TCP). Table 6 shows that ScanVPN is the best (F1-score between 84.66% to 95.49%) compared to others. Finally, we select ScanVPN for the app classifier for VPN traffic (C_{VPN}) component.

We note two observations on AppScanner as follows. First, we observe that AppScanner demonstrates high precision between 80.77% and 97.36% and low recall between 16.40% and 33.32%, demonstrating that there are many false negative cases. In other words, it fails to detect many malicious apps, which would be inappropriate for a security application. Second, although AppScanner and ScanVPN use a similar feature (i.e., a sequence of packet lengths), the gap between the performance of ScanVPN (F1-score between 84.66% and 95.49%) and that of AppScanner (F1-score between 25.63% and 47.56%) is significantly large.

We also observe FlowPrint's critical limitation for VPN traffic. FlowPrint cannot identify apps on VPN traffic (F1-score between 0.56% and 3.41%). Note that FlowPrint identifies apps based on their destination IP addresses. In our experiment, FlowPrint only refers to the VPN servers' IP addresses; thus, it fails to identify apps with insufficient information. In other words, FlowPrint cannot distinguish two different apps that use the same VPN server as

[8] https://developer.android.com/studio/command-line/adb
[9] https://tshark.dev
[10] https://www.wireshark.org
[11] https://auto.gluon.ai

Table 6: Comparison results of classifiers for VPN traffic.

Dataset		ScanVPN				FlowPrint				AppScanner				ET-BERT			
		Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
SuperVPN	CV1	0.8383	**0.8233**	**0.8166**	**0.8233**	0.0236	0.0467	0.0202	0.0467	**0.8394**	0.1657	0.2600	0.1657	0.0000	0.0067	0.0001	0.0067
	CV2	0.9088	**0.9013**	**0.8992**	**0.9013**	0.0204	0.0480	0.0203	0.0480	**0.9810**	0.2793	0.4127	0.2793	0.0000	0.0067	0.0001	0.0067
	CV3	0.9342	**0.9273**	**0.9252**	**0.9273**	0.0172	0.0473	0.0193	0.0473	**0.9736**	0.3332	0.4756	0.3332	0.0002	0.0127	0.0004	0.0127
TurboVPN	CV1	**0.8978**	**0.8887**	**0.8873**	**0.8887**	0.0335	0.0693	0.0340	0.0693	0.8651	0.1514	0.2427	0.1514	0.0000	0.0067	0.0001	0.0067
	CV2	0.9507	**0.9453**	**0.9450**	**0.9453**	0.0339	0.0747	0.0344	0.0747	**0.9537**	0.2394	0.3655	0.2394	0.0000	0.0067	0.0001	0.0067
	CV3	0.9590	**0.9553**	**0.9549**	**0.9553**	0.0322	0.0707	0.0341	0.0707	**0.9680**	0.2787	0.4152	0.2787	0.0000	0.0067	0.0001	0.0067
NordVPN	CV1	0.8518	**0.8367**	**0.8324**	**0.8367**	0.0024	0.0280	0.0042	0.0280	**0.9421**	0.1504	0.2457	0.1504	0.0000	0.0067	0.0001	0.0067
	CV2	0.9009	**0.8887**	**0.8853**	**0.8887**	0.0029	0.0320	0.0050	0.0320	**0.9668**	0.2154	0.3371	0.2154	0.0000	0.0067	0.0001	0.0067
	CV3	0.9397	**0.9333**	**0.9331**	**0.9333**	0.0032	0.0333	0.0056	0.0333	**0.9655**	0.2640	0.3998	0.2640	0.0000	0.0067	0.0001	0.0067
Surfshark	CV1	**0.6852**	**0.6593**	**0.6510**	**0.6593**	0.0139	0.0500	0.0151	0.0500	0.5878	0.1063	0.1680	0.1063	0.0000	0.0067	0.0001	0.0067
	CV2	**0.7997**	**0.7760**	**0.7682**	**0.7760**	0.0136	0.0507	0.0150	0.0507	0.7768	0.1399	0.2221	0.1399	0.0000	0.0067	0.0001	0.0067
	CV3	**0.8650**	**0.8507**	**0.8466**	**0.8507**	0.0140	0.0500	0.0141	0.0500	0.8077	0.1640	0.2563	0.1640	0.0000	0.0067	0.0001	0.0067

Table 7: Comparison results of overall performance on normal and VPN dataset.

Dataset		AppSniffer				FlowPrint				AppScanner				ET-BERT			
		Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
Normal	CV1	0.8231	**0.8078**	**0.8044**	**0.8078**	0.2110	0.2339	0.2098	0.2339	**0.9837**	0.4517	0.5951	0.4517	0.4071	0.3928	0.3968	0.3928
&	CV2	0.8861	**0.8739**	**0.8721**	**0.8739**	0.2117	0.2379	0.2118	0.2379	**0.9846**	0.5331	0.6740	0.5331	0.4639	0.4529	0.4562	0.4529
All VPNs	CV3	0.9155	**0.9068**	**0.9063**	**0.9068**	0.2098	0.2353	0.2100	0.2353	**0.9887**	0.5648	0.7036	0.5648	0.4911	0.4639	0.4713	0.4639

their fingerprints are identical. In the VPN scenario, FlowPrint can recognize whether the app traffic is sent from VPN or not, but it is not proper for the app classifier for VPN traffic.

There are two findings on ET-BERT. First, as the results in Table 6 show, we observe that ET-BERT is ineffective for app identification for VPN traffic. The F1-scores for ET-BERT on all VPN datasets are less than 0.04%. As discussed, ET-BERT highly relies on plaintext fields, including TCP option fields, TLS record headers, and plaintext metadata in TLS, which requires the transport layer to be TCP. However, many VPN protocols use UDP as their transport layer; thus, the F1-score of ET-BERT decreases compared to that of TCP traffic. Second, due to the limited size (a maximum sequence length of 512 tokens by default [10]) of the input sequence for the BERT model, it is difficult for ET-BERT to use all the raw packets to build its model. We find that in the ET-BERT implementation, a feature vector is generated by concatenating raw packets until the number of tokens is enough for model training. The default value of the number of packets to be parsed is five, which means that the first five packets of a given flow are used to construct a feature vector. Because of it, in some flows, there are cases where only packets of TCP or TLS handshakes are captured for a feature vector, making ET-BERT depend on the plaintext metadata (e.g., a server name in TLS Client Hello) in the normal scenario. However, ET-BERT cannot learn these features in the VPN dataset. Therefore, ET-BERT is inappropriate for the app recognition for VPN traffic because of lack of information. For example, as we can see from the results on SuperVPN and NordVPN, ET-BERT completely failed to identify the app traffic (F1-score of 0.04% and 0.01% on SuperVPN and NordVPN). When it comes to TurboVPN and Surfshark, the dominant features are SPI in IPsec ESP, TSval, and TSecr in the TCP option field. We find that without them (i.e., applying to mask), the performance of ET-BERT significantly decreases. Moreover, when the dataset size is insufficient to train the patterns, the performance of ET-BERT decreases close to the F1-score of 0%. That is, a large training dataset size is required for fine-tuning compared to other systems.

Overall performance of AppSniffer. In this experiment, we compare the overall performance of AppSniffer with those of the existing app fingerprinting systems against the mixed traffic with VPN traffic and normal traffic. The objective of this experiment is to see whether the systems are practical as they may face mixed traffic in general. Our result reports that AppSniffer shows the best performance (90.63% of F1-score) among the evaluated systems (see Table 7).

In addition, we also experimented to see the performance of the systems when they were trained on normal traffic and tested on VPN traffic to show the limitation of the existing app fingerprinting systems. The result shows that all the existing systems cannot properly recognize the VPN traffic. The F1-scores of FlowPrint and AppScanner are 0%, and that of ET-BERT is less than 0.52% (see Table 8), demonstrating that it is necessary to consider the VPN traffic for training.

Table 8: Performance of ET-BERT on VPN traffic dataset when ET-BERT is trained only with normal traffic.

Dataset	**ET-BERT** (trained only with normal traffic)			
	Precision	Recall	F1-score	Accuracy
SuperVPN	0.0081	0.0080	0.0052	0.0080
TurboVPN	0.0039	0.0093	0.0035	0.0093
NordVPN	0.0033	0.0080	0.0037	0.0080
Surfshark	0.0062	0.0087	0.0051	0.0087

Optimal feature size for the app classifier. For ScanVPN, we select the feature size (i.e., the number of packets in a sequence) as the average number of packets in each flow based on a training dataset. For example, based on a training dataset in CV3, 3,410, 4,213, 2,670, and 1,643 are selected as the feature sizes on SuperVPN, TurboVPN, NordVPN, and Surfshark, respectively. To optimize the classifier, we also conduct an experiment to see the performance of ScanVPN varying the feature size used in a sequence in the performance of ScanVPN, ranging from 500 to 10,000 (see Figure 3). The experiment is done on CV3, and the metric used is F1-score. Overall, ScanVPN's performance improves as the feature size increases, even though there are some fluctuations. Based on these results, we suggest that the feature size should be larger than 1,250.

6 DISCUSSION AND LIMITATIONS

Adaptive attacks. A limitation of our framework is that it would be vulnerable to adaptive attacks. For example, an attacker can generate the app traffic with a proxy (e.g., Obfs4 proxy [3]) that can change a sequence of packet lengths in a flow. That is, using a proxy, the attacker can append padding bytes with zero to all the packets to make them equal to the maximum transmission unit. Such an

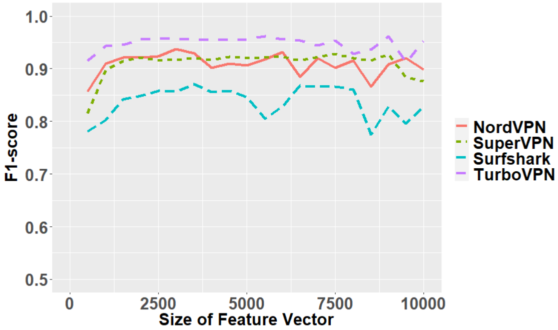

Figure 3: Performance of ScanVPN with the feature size.

active obfuscation scheme may make difficult for AppSniffer to find significant patterns from traffic. Furthermore, we do not design AppSniffer to be robust against adversarial example attacks [5, 38]. As a countermeasure, we can introduce an adversarial example detection system [15, 36] in the first stage of AppSniffer and block the attack traffic. As another countermeasure, we can generate and train adversarial traffic for adversarial learning [34].

Generating valid datasets. Based on the feature importance analysis, we find that some raw packet-based systems highly rely on the features in a specific scenario, which makes it difficult to deploy the system in the real world. For instance, we find that the dominant features of one system are TCP option fields which are not generally used. Further, some TCP fields are timestamps when the dataset is collected, and the values are similar in both the training set and the testing set (only the last byte of the timestamps are different), which results in high accuracy. However, in the real world, we cannot expect that there exist such similar values in the fields. Therefore, we suggest masking some field values (e.g., TSval and TSecr in the TCP option fields) in the datasets that cannot be reflected in real-world situations.

Limitation against unknown VPNs Our current AppSniffer implementation cannot identify apps over VPNs unknown to AppSniffer. In this case, AppSniffer can only label the given traffic as *unknown VPN traffic*. For future research, we plan to explore additional features and develop more advanced classifiers for identifying apps when an unseen VPN protocol is used.

Limitation against multiple active apps AppSniffer is designed as a single-label classifier, in line with previous works such as [32, 46, 47], under the same assumptions. However, in practical settings, multiple apps may be running simultaneously, making it difficult to differentiate packets related to a specific app since they are merged in a single VPN flow. To address this issue, we have enhanced AppSniffer by dividing the VPN flow into smaller segments based on packet inter-arrival times and classifying each segment individually. We conducted a pilot study to evaluate the feasibility of this approach. We randomly selected two apps from a pool of 10 popular Android apps and ran them simultaneously over TurboVPN 50 times. Traffic data for both apps was collected for 30 seconds. Our preliminary results demonstrate that AppSniffer was able to identify both apps with an accuracy of 10%, and either of the apps with a 58% accuracy. In the future, we plan to develop a more general multi-label model to handle scenarios where multiple apps are running concurrently.

7 RELATED WORK

Mobile App Fingerprinting Mobile traffic has increased dramatically in recent years as mobile devices and apps have grown tremendously. In the meantime, mobile app fingerprinting is getting much attention. AppPrint [37] makes fingerprints of mobile apps based on statistics of tokens such as HTTP header strings or query characters contained in HTTP packets. However, this approach is difficult to use in practice when packets are encrypted; AppPrint cannot read content of the encrypted packets. To address such a challenge, Taylor *et al.* [46] proposed AppScanner that classifies HTTPS traffic with networking patterns instead of content in HTTP headers. AppScanner uses 54 statistical features from the collected packets and builds models based on a support vector machine and Random Forest models. Some frameworks [41, 47] have been proposed to use characteristics of mobile traffic according to user behavior. Using a semi-supervised learning approach, FlowPrint [47] uses the destination IP addresses/port numbers, timestamps, packet sizes, and TLS certificate to identify mobile apps. MAppGraph [41] constructs a communication graph containing a node with the destination IP address and port as a tuple and an edge-weighted communication correlation. This approach uses deep graph convolution neural networks. Lin *et al.* [32] proposed a system dubbed ET-BERT that takes a pre-trained BERT model based on raw traffic and fine-tunes it on encrypted traffic. However, as discussed, all of the above approaches are ineffective in identifying apps running over VPNs.

VPN Detection There have been several previous approaches to detect VPN traffic. Gao *et al.* [17] searches the best VPN traffic classifiers among Naive Bayesian, Logistic Regression, Support Vector Machine, XGBoost, and Random Forest algorithms based on deep packet inspection, sample entropy fingerprint, and a sequence of payload lengths, respectively. Similarly, Bagui *et al.* [4] also uses machine learning algorithms based on time-related features to distinguish VPN traffic from normal traffic. Other approaches learn features from raw traffic or detected VPN traffic using LSTM models [16, 48]. Xue *et al.* [50] propose an approach for OpenVPN fingerprinting based on byte patterns, packet lengths, and responses from the VPN server. These approaches, however, only focus on detecting VPN traffic but do not aim to identify mobile apps over VPNs.

8 CONCLUSION

In this paper, we discuss the limitation of the existing mobile app fingerprint systems against an attacker that uses VPN protocols to hide its malicious activities. We propose AppSniffer, a framework able to recognize app traffic over both VPN traffic and normal traffic. Our numerical results show that AppSniffer is effective with a high F1-score of 90.63%.

There are three directions for future work. First, we plan to extend AppSniffer to prevent adaptive or adversarial attacks. For instance, we may consider obfuscation. Second, we will strengthen AppSniffer to identify apps over unknown VPNs. Third, we will extend AppSniffer to identify simultaneously active apps.

ACKNOWLEDGMENTS

The authors would thank anonymous reviewers. Hyoungshick Kim and Hyunwoo Lee are the corresponding authors. This work was supported by the KENTECH Research Grant (202200048A), Korea Internet & Security Agency (KISA) grant (No.1781000003), and the Institute of Information & communications Technology Planning & Evaluation (IITP) grants (No.2022-0-00688, No.2019-0-01343, and No.2022-0-00495) funded by the Korean government.

REFERENCES

[1] 2022. Percentage of Mobile Device Website Traffic Worldwide from 1st Quarter 2015 to 2nd Quarter 2022. https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices. (Accessed on 9/23/2022).

[2] André Altmann, Laura Toloşi, Oliver Sander, and Thomas Lengauer. 2010. Permutation Importance: A Corrected Feature Importance Measure. *Bioinformatics* (2010).

[3] Yawning Angel and Philipp Winter. 2014. Obfs4 (The Obfourscator). https://gitweb.torproject.org/pluggable-transports/obfs4.git/tree/doc. (Accessed on 10/06/2022).

[4] Sikha Bagui, Xingang Fang, Ezhil Kalaimannan, Subhash C. Bagui, and Joseph Sheehan. 2017. Comparison of Machine-learning Algorithms for Classification of VPN Network Traffic Flow Using Time-related Features. *Journal of Cyber Security Technology* (2017).

[5] Nicholas Carlini and David Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[6] Yi-Chao Chen, Yong Liao, Mario Baldi, Sung-Ju Lee, and Lili Qiu. 2014. OS Fingerprinting and Tethering Detection in Mobile Networks. In *Proc. of the ACM Internet Measurement Conference (IMC)*.

[7] Irfaan Coonjah, Pierre Clarel Catherine, and K. M. S Soyjaudah. 2015. Experimental Performance Comparison Between TCP vs UDP Tunnel Using OpenVPN. In *Proc. of the IEEE International Conference on Computing, Communication and Security (ICCCS)*.

[8] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. 2013. NetworkProfiler: Towards Automatic Fingerprinting of Android Apps. In *Proc of the IEEE International Conference on Computer Communications (INFOCOM)*.

[9] Statista Research Department. 2022. Number of smartphone subscriptions worldwide from 2016 to 2021, with forecasts from 2022 to 2027. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. (Accessed on 10/14/2022).

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[11] Jason A Donenfeld. 2017. WireGuard: Next Generation Kernel Network Tunnel. In *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*.

[12] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A. Ghorbani. 2016. Characterization of Encrypted and VPN Traffic Using Time-related Features. In *Proc. of the International Conference on Information Systems Security and Privacy (ICISSP)*.

[13] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. Autogluon-Tabular: Robust and Accurate AutoML for Structured Data. *arXiv preprint arXiv:2003.06505* (2020).

[14] Dino Farinacci, Tony Li, Stan Hanks, David Meyer, and Paul Traina. 2000. Generic Routing Encapsulation (GRE). https://www.rfc-editor.org/rfc/rfc2784. (Accessed on 10/10/2022).

[15] Reuben Feinman, Ryan R. Curtin, Saurabh Shintre, and Andrew B. Gardner. 2017. Detecting Adversarial Samples from Artifacts. *arXiv preprint arXiv:1703.00410* (2017).

[16] Peipei Fu, Chang Liu, Qingya Yang, Zhenzhen Li, Gaopeng Gou, Gang Xiong, and Zhen Li. 2020. NSA-Net: A NetFlow Sequence Attention Network for Virtual Private Network Traffic Detection. In *Proc. of the International Conference on Web Information Systems Engineering (WISE)*.

[17] Ping Gao, Guangsong Li, Yanan Shi, and Yang Wang. 2020. VPN Traffic Classification Based on Payload Length Sequence. In *Proc. of the International Conference on Networking and Network Applications (NaNA)*.

[18] Cyril Goutte and Eric Gaussier. 2005. A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation. In *Proc. of the European Conference on Information Retrieval (ECIR)*.

[19] Kory Hamzeh, Grueep Pall, William Verthein, Jeff Taarud, W Little, and Glen Zorn. 1999. Point-to-Point Tunneling Protocol (PPTP). https://www.rfc-editor.org/rfc/rfc2637. (Accessed on 10/10/2022).

[20] Jeremy Howard and Sylvain Gugger. 2020. FastAI: A Layered API for Deep Learning. *Information* (2020).

[21] Van Jacobson, Robert Braden, and David Borman. 1992. TCP Extensions for High Performance. https://www.rfc-editor.org/rfc/rfc1323. (Accessed on 10/06/2022).

[22] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. 2020. Programmable In-Network Security for Context-aware BYOD Policies. In *Proc. of the USENIX Security Symposium (USENIX Security)*.

[23] Charlie Kaufman, Paul Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. 2014. Internet Key Exchange Protocol Version 2 (IKEv2). https://www.rfc-editor.org/rfc/rfc7296. (Accessed on 09/23/2022).

[24] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. *Advances in Neural Information Processing Systems* (2017).

[25] Stephen Kent. 2005. IP Encapsulating Security Payload (ESP). https://www.rfc-editor.org/rfc/rfc4303. (Accessed on 10/10/2022).

[26] S. Kent and K. Seo. 2005. Security Architecture for the Internet Protocol. https://www.rfc-editor.org/rfc/rfc4301. (Accessed on 09/23/2022).

[27] Hyunchul Kim, Kimberly C Claffy, Marina Fomenkov, Dhiman Barman, Michalis Faloutsos, and KiYoung Lee. 2008. Internet Traffic Classification Demystified: Myths, Caveats, and the Best Practices. In *Proc. of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*.

[28] Arash Habibi Lashkari, Gerard Draper Gil, Mohammad Saiful Islam Mamun, and Ali A. Ghorbani. 2017. Characterization of Tor Traffic Using Time Based Features. In *Proc. of the International Conference on Information Systems Security and Privacy (ICISSP)*.

[29] Hyunwoo Lee, Imtiaz Karim, Ninghui Li, and Elisa Bertino. 2022. VWAnalyzer: A Systematic Security Analysis Framework for the Voice over WiFi Protocol. In *Proc. of the ACM on Asia Conference on Computer and Communications Security (ASIACCS)*.

[30] Hyunwoo Lee, Doowon Kim, and Yonghwi Kwon. 2021. TLS 1.3 in Practice: How TLS 1.3 Contributes to the Internet. In *Proc. of the ACM Web Conference (WWW)*.

[31] Kunda Lin, Xiaolong Xu, and Honghao Gao. 2021. TSCRNN: A Novel Classification Scheme of Encrypted Traffic Based on Flow Spatiotemporal Features for Efficient Management of IIoT. *Computer Networks* (2021).

[32] Xinjie Lin, Gang Xiong, Gaopeng Gou, Zhen Li, Junzheng Shi, and Jing Yu. 2022. ET-BERT: A Contextualized Datagram Representation with Pre-training Transformers for Encrypted Traffic Classification. In *Proc. of the ACM Web Conference (WWW)*.

[33] 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* (2020).

[34] Daniel Lowd and Christopher Meek. 2005. Adversarial Learning. In *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*.

[35] Petr Matoušek, Ivana Burgetová, Ondřej Ryšavý, and Malombe Victor. 2020. On Reliability of JA3 Hashes for Fingerprinting Mobile Applications. In *Proc. of the EAI International Conference on Digital Forensics and Cyber Crime (ICDF2C)*.

[36] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. 2017. On Detecting Adversarial Perturbations. *arXiv preprint arXiv:1702.04267* (2017).

[37] Stanislav Miskovic, Gene Moo Lee, Yong Liao, and Mario Baldi. 2015. AppPrint: Automatic Fingerprinting of Mobile Applications in Network Traffic. In *Proc. of the International Conference on Passive and Active Network Measurement (PAM)*.

[38] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. 2021. Defeating DNN-Based Traffic Analysis Systems in Real-Time With Blind Adversarial Perturbations. In *Proc. of the USENIX Security Symposium (USENIX Security)*.

[39] G. Pall. 1997. Microsoft Point-To-Point Compression (MPPC) Protocol. https://www.rfc-editor.org/rfc/rfc3078. (Accessed on 10/10/2022).

[40] Ioannis Papapanagiotou, Erich Nahum, and Vasileios Pappas. 2012. Configuring DHCP Leases in the Smartphone Era. In *Proc. of the ACM Internet Measurement Conference (IMC)*.

[41] 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 *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.

[42] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. https://www.rfc-editor.org/rfc/rfc8446.

[43] Subhabrata Sen, Oliver Spatscheck, and Dongmei Wang. 2004. Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures. In *Proc. of the ACM Web Conference (WWW)*.

[44] William Simpson. 1993. The Point-to-Point Protocol (PPP). https://www.rfc-editor.org/rfc/rfc1548. (Accessed on 10/10/2022).

[45] Zixian Tang, Qiang Wang, Wenhao Li, Huaifeng Bao, Feng Liu, and Wen Wang. 2021. HSLF: HTTP Header Sequence Based LSH Fingerprints for Application Traffic Classification. In *Proc. of the International Conference on Computational Science (ICCS)*.

[46] Vincent F. Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. 2016. AppScanner: Automatic Fingerprinting of Smartphone Apps from Encrypted Network Traffic. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*.

[47] 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. of the ISOC Network and Distributed System Security Symposium (NDSS)*.

[48] Wei Wang, Ming Zhu, Jinlin Wang, Xuewen Zeng, and Zhongzhen Yang. 2017. End-to-end Encrypted Traffic Classification with One-dimensional Convolution Neural Networks. In *Proc. of the IEEE international conference on intelligence and security informatics (ISI)*.

[49] Wei Wang, Ming Zhu, Xuewen Zeng, Xiaozhou Ye, and Yiqiang Sheng. 2017. Malware Traffic Classification Using Convolutional Neural Network for Representation Learning. In *Proc. of the IEEE International conference on information networking (ICOIN)*.

[50] Diwen Xue, Reethika Ramesh, Arham Jain, Michalis Kallitsis, J. Alex Halderman, Jedidiah R. Crandall, and Roya Ensafi. 2022. OpenVPN is Open to VPN Fingerprinting. In *Proc. of the USENIX Security Symposium (USENIX Security)*.

[51] Liuqun Zhai, Zhuang Qiao, Zhongfang Wang, and Dong We. 2021. Identify What You are Doing: Smartphone Apps Fingerprinting on Cellular Network Traffic. In *Proc. of the IEEE Symposium on Computers and Communications (ISCC)*.

A FEATURE IMPORTANCE SCORES OF ET-BERT

We used a permutation-based feature importance measure [2] to analyze the relative importance of individual features. Table 9 provides the importance scores computed for 10 most important features of ET-BERT on the Cross-Platform datasets [47]. In both Cross-Platform (iOS) and Cross-Platform (Android) datasets, Timestamp Value (TSval) and Timestamp Echo Reply (TSecr) [21] in the TCP option fields are commonly important.

B LIST OF MOBILE APPS

We provide a list of mobile apps selected in our experiments (see Table 10).

Table 9: 10 most important features of ET-BERT on the Cross-Platform datasets [47].

Cross-Platform (iOS)			Cross-Platform (Android)		
Feature	**Score**	**p-value**	**Feature**	**Score**	**p-value**
Timestamp value (TCP Option) in the first packet	0.0046	0.0156	Timestamp value (TCP Option) in the first packet	0.0027	0.0158
Timestamp echo reply (TCP Option) in the first packet	0.0011	0.1267	Timestamp echo reply (TCP Option) in the second packet	0.0011	0.0037
Timestamp echo reply (TCP Option) in the fourth packet	0.0008	0.0516	Timestamp value (TCP Option) in the third packet	0.0011	0.0997
TCP payload in the third packet	0.0007	0.0947	Timestamp echo reply (TCP Option) in the second packet	0.0009	0.0533
Timestamp value (TCP Option) in the third packet	0.0007	0.0726	Timestamp value (TCP Option) in the second packet	0.0008	0.2000
Timestamp value (TCP Option) in the first packet	0.0006	0.3336	Timestamp value (TCP Option) in the fourth packet	0.0008	0.0801
Timestamp value (TCP Option) in the second packet	0.0006	0.3131	Timestamp value (TCP Option) in the fifth packet	0.0008	0.0947
TCP payload in the third packet	0.0005	0.2113	Timestamp value (TCP Option) in the first packet	0.0008	0.0959
Public Key (TLS Client Key Exchange) in the fourth packet	0.0005	0.0995	Timestamp value (TCP Option) in the fourth packet	0.0006	0.1080
Server name (TLS Client Hello) in the first packet	0.0005	0.0995	Timestamp value (TCP Option) in the fourth packet	0.0005	0.2859

Table 10: List of Mobile Apps

No.	Package Name	Number of Installs	No.	Package Name	Number of Installs
1	com.facebook.lite	> 10 billion	76	com.appswing.qr.barcodescanner.barcodereader	> 10 million
2	com.google.android.apps.translate	> 10 billion	77	com.bestbuy.android	> 10 million
3	com.google.android.play.games	> 10 billion	78	com.bravo.booster	> 10 million
4	com.instagram.android	> 10 billion	79	com.brighthouse.mybhn	> 10 million
5	com.snapchat.android	> 10 billion	80	com.cbs.app	> 10 million
6	com.spotify.music	> 10 billion	81	com.coinbase.android	> 10 million
7	com.zhiliaoapp.musically	> 10 billion	82	com.creditkarma.mobile	> 10 million
8	com.booking	> 500 million	83	com.discovery.discoveryplus.mobile	> 10 million
9	com.contextlogic.wish	> 500 million	84	com.dominospizza	> 10 million
10	com.facebook.mlite	> 500 million	85	com.doordash.driverapp	> 10 million
11	com.google.android.apps.nbu.paisa.user	> 500 million	86	com.etsy.android	> 10 million
12	com.picsart.studio	> 500 million	87	com.fetchrewards.fetchrewards.hop	> 10 million
13	com.pinterest	> 500 million	88	com.fortunescope	> 10 million
14	com.shazam.android	> 500 million	89	com.google.android.apps.youtube.unplugged	> 10 million
15	us.zoom.videomeetings	> 500 million	90	com.grubhub.android	> 10 million
16	com.agminstruments.drumpadmachine	> 100 million	91	com.hopper.mountainview.play	> 10 million
17	com.airbnb.android	> 100 million	92	com.instacart.client	> 10 million
18	com.amazon.avod.thirdpartyclient	> 100 million	93	com.kohls.mcommerce.opal	> 10 million
19	com.audible.application	> 100 million	94	com.konylabs.capitalone	> 10 million
20	com.callapp.contacts	> 100 million	95	com.mcdonalds.app	> 10 million
21	com.canva.editor	> 100 million	96	com.mercariapp.mercari	> 10 million
22	com.cleanteam.oneboost	> 100 million	97	com.microsoft.bing	> 10 million
23	com.digidust.elokence.akinator.freemium	> 100 million	98	com.mistplay.mistplay	> 10 million
24	com.discord	> 100 million	99	com.myhomescreen.sms	> 10 million
25	com.disney.disneyplus	> 100 million	100	com.myklarnamobile	> 10 million
26	com.duolingo	> 100 million	101	com.nbcuni.nbc	> 10 million
27	com.ebay.mobile	> 100 million	102	com.nextdoor	> 10 million
28	com.gamma.scan	> 100 million	103	com.nike.omega	> 10 million
29	com.google.android.apps.adm	> 100 million	104	com.onedebit.chime	> 10 million
30	com.google.android.apps.authenticator2	> 100 million	105	com.peacocktv.peacockandroid	> 10 million
31	com.google.android.apps.chromecast.app	> 100 million	106	com.pinger.textfree	> 10 million
32	com.google.android.apps.youtube.kids	> 100 million	107	com.rfi.sams.android	> 10 million
33	com.google.earth	> 100 million	108	com.ringapp	> 10 million
34	com.gotv.nflgamecenter.us.lite	> 100 million	109	com.shopify.arrive	> 10 million
35	com.hbo.hbonow	> 100 million	110	com.sirius	> 10 million
36	com.indeed.android.jobsearch	> 100 million	111	com.starbucks.mobilecard	> 10 million
37	com.lemon.lvoverseas	> 100 million	112	com.tacobell.ordering	> 10 million
38	com.microsoft.teams	> 100 million	113	com.ticketmaster.mobile.android.na	> 10 million
39	com.opera.app.news	> 100 million	114	com.tool.fast.smart.cleaner	> 10 million
40	com.pandora.android	> 100 million	115	com.united.mobile.android	> 10 million
41	com.paypal.android.p2pmobile	> 100 million	116	com.wave.livewallpaper	> 10 million
42	com.psafe.msuite	> 100 million	117	com.zillow.android.zillowmap	> 10 million
43	com.reddit.frontpage	> 100 million	118	me.lyft.android	> 10 million
44	com.soundcloud.android	> 100 million	119	co.vulcanlabs.rokuremote	> 5 million
45	com.tinder	> 100 million	120	com.affirm.central	> 5 million
46	com.tubitv	> 100 million	121	com.afterpaymobile.us	> 5 million
47	com.ubercab.eats	> 100 million	122	com.airgoat.goat	> 5 million
48	com.waze	> 100 million	123	com.americasbestpics	> 5 million
49	com.weather.Weather	> 100 million	124	com.fandango.regal	> 5 million
50	com.zzkko	> 100 million	125	com.filemanager.files.explorer.boost.clean	> 5 million
51	jp.ne.ibis.ibispaintx.app	> 100 million	126	com.immediasemi.android.blink	> 5 million
52	net.zedge.android	> 100 million	127	com.justplay.app	> 5 million
53	sg.bigo.live	> 100 million	128	com.macys.android	> 5 million
54	tv.twitch.android.app	> 100 million	129	com.meetalbert	> 5 million
55	com.amazon.clouddrive.photos	> 50 million	130	com.nvidia.geforcenow	> 5 million
56	com.amazon.dee.app	> 50 million	131	com.oculus.twilight	> 5 million
57	com.apple.android.music	> 50 million	132	com.optimizer.pro.beeztel	> 5 million
58	com.azure.authenticator	> 50 million	133	com.thunderclap.fakecallfromsantavideocallsammy	> 5 million
59	com.bumble.app	> 50 million	134	com.vrbo.android	> 5 million
60	com.cleanteam.onesecurity	> 50 million	135	us.current.android	> 5 million
61	com.dd.doordash	> 50 million	136	com.blockfolio.blockfolio	> 1 million
62	com.enflick.android.TextNow	> 50 million	137	com.cbs.tve	> 1 million
63	com.espn.score_center	> 50 million	138	com.engro.cleanerforsns	> 1 million
64	com.hp.printercontrol	> 50 million	139	com.home.bible.verse.prayer	> 1 million
65	com.hulu.plus	> 50 million	140	com.investvoyager	> 1 million
66	com.microsoft.xboxone.smartglass	> 50 million	141	com.love.biremoji	> 1 million
67	com.offerup	> 50 million	142	com.pointone.buddyglobal	> 1 million
68	com.particlenews.newsbreak	> 50 million	143	com.vod.vodcy	> 1 million
69	com.roku.remote	> 50 million	144	com.weatherport.android	> 1 million
70	com.yelp.android	> 50 million	145	org.gamatech.androidclient.app	> 1 million
71	clean.phone.cleaner.boost.security.applock	> 10 million	146	org.toshi	> 1 million
72	com.aa.android	> 10 million	147	com.mwm.beat_looper_pro	> 0.5 million
73	com.adpmobile.android	> 10 million	148	com.finnci.finnci	> 0.1 million
74	com.ai.face.play	> 10 million	149	com.ralphlauren.us.app	> 0.1 million
75	com.amazon.storm.lightning.client.aosp	> 10 million	150	com.wodol.dol	> 0.1 million