Packet-Level Open-World App Fingerprinting on Wireless Traffic

Jianfeng Li[1], Shuohan Wu[1], Hao Zhou[1], Xiapu Luo[1]*, Ting Wang[2], Yangyang Liu[1] and Xiaobo Ma[3]
[1]The Hong Kong Polytechnic University, {csljianfeng, csswu, cshaoz, csxluo, csyyliu}@comp.polyu.edu.hk
[2]Pennsylvania State University, ting@psu.edu
[3]Xi'an Jiaotong University, xma.cs@xjtu.edu.cn

Abstract—**Mobile apps have profoundly reshaped modern lifestyles in different aspects. Several concerns are naturally raised about the privacy risk of mobile apps. Despite the prevalence of encrypted communication, app fingerprinting (AF) attacks still pose a serious threat to users' online privacy. However, existing AF attacks are usually hampered by four challenging issues, namely i) hidden destination, ii) invisible boundary, iii) app multiplexing, and iv) open-world recognition, when they are applied to wireless traffic. None of existing AF attacks can address all these challenges. In this paper, we advance a novel AF attack, dubbed PACKETPRINT, to recognize user activities associated with the app of interest from encrypted wireless traffic and tackle the above challenges by proposing two novel models, i.e., sequential XGBoost and hierarchical bag-of-words model. We conduct extensive experiments to evaluate the proposed attack in a series of challenging scenarios, including i) open-world setting, ii) packet loss and network congestion, iii) simultaneous use of different apps, and iv) cross-dataset recognition. The experimental results show that PACKETPRINT can accurately recognize user activities associated with the apps of interest. It achieves the average F1-score** 0.884 **for open-world app recognition and the average F1-score** 0.959 **for in-app user action recognition.**

I. INTRODUCTION

Mobile devices, such as smartphones and tablets, are ubiquitous in modern life. A myriad of mobile apps empower them with the capabilities that profoundly reshape people's lifestyle, ranging from information retrieval to instant messaging, and from shopping to entertainment. The recent prevalence of online to offline apps, e.g., food delivery, further signalizes this trend by bridging the gap between the online information and physical businesses [1], [2].

Every coin has a flip side. Mobile apps offer users high-quality services, accompanied by the collection, transmission, storage, and even sharing of user data, raising serious privacy concerns. For example, compromising apps' cloud servers may cause disastrous privacy leakage [3]–[5]. Such attacks are basically due to software flaws. While threatening, these flaws are scarce and, once found, will be immediately fixed to avoid severe consequences. Data transmission channel can be another vulnerable point. Despite the widespread adoption of encrypted communication, mobile apps are still susceptible to app fingerprinting (AF) attacks [6]–[14]. Adversaries recognize user activities associated with the app of interest to infer user privacy. Such a threat has been increasingly aggravated. First, the rapid proliferation of appified IoT systems, such as Smart home [15], [16], potentially extends cyber-space attacks to the physical world. Therefore, leaking sensitive information, such as user dynamics and installed IoT devices with security flaws, may endanger not only the privacy but also the safety of users [17]. Second, the COVID-19 pandemic has unprecedentedly influenced personal living habits [18], [19]. People experienced the imposition of lockdown and had to get used to doing everything online, from virtual meeting to online education, potentially expanding the attack surface for AF attacks.

Existing AF attacks [6]–[14] are generally carried out via capturing TCP/IP traffic in the wireless access point (AP). Unfortunately, their practicality is limited because the adversary needs to comprise the AP or conspire with the network administrator for the permission to capture TCP/IP traffic in AP. Contrarily, a WiFi sniffer enables the adversary to passively capture 802.11 wireless frames between mobile devices and the wireless AP over the air without being noticed. By leveraging WiFi sniffer, the adversary needs neither controlling the wireless AP nor the permission granted by the network administrator, thereby lowering the barrier for AF attacks. For example, the adversary can launch an AF attack outside the door and recognize user activities associated with various apps, e.g., Smart home apps, to infer indoor human dynamics for reconnaissance purposes. However, existing AF attacks will be hampered by four-fold challenges when they are applied to wireless traffic, and none of them can fully solve these challenges.

- **Hidden Destination.** Raw destination information, such as IP address and domain name of the remote server, is hidden because Internet-layer, transport-layer, and application-layer data are encrypted and encapsulated in the frame body of 802.11 wireless frames. Consequently, AF attacks based on IP address [6], [11] or traffic analysis techniques based on DNS [20] are inapplicable to 802.11 wireless frames.
- **Invisible Boundary.** TCP/UDP network flows of mobile traffic cannot be identified and extracted because transport-layer endpoints are invisible in 802.11 wireless frames. Therefore, AF attacks that need to extract network flows as traffic samples [7]–[9], [11], [14] are unable to handle 802.11 wireless frames. An alternative solution widely used

*The corresponding author.

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24210
www.ndss-symposium.org

in website fingerprinting (WF) attacks [21]–[28] is extracting traffic samples as segments of encrypted traffic separated by obvious time gaps between packets. Unfortunately, such an idea can hardly be borrowed by AF attacks because app traffic is much more complex as it generally contains lots of packet bursts mixed with background traffic during a long time. As a result, it is very difficult to find a proper threshold of time gap to separate traffic generated by different apps.

- **App Multiplexing.** Packets generated by different apps may be strongly mixed together because mobile users may use different apps simultaneously in the wild. For example, a user may read news with Flipboard app while listening to music with Apple music app. If the adversary wants to analyze user activities associated with Flipboard, packets generated by Apple music naturally become interleaved noises that potentially undermine recognition accuracy. The situation becomes even worse since Android 7 because two apps can be in the foreground at the same time by splitting the screen [29]. Since 802.11 wireless frames conceal Internet-layer and transport-layer endpoints in the encrypted frame body, packets generated by different apps cannot be grouped into different network flows for further recognition. None of existing AF attacks can handle app multiplexing.
- **Open-World Recognition.** Most existing AF attacks [10]–[12], [14] work under the closed-world assumption. That is, apps presented in the recognition stage must also be present during model training. Otherwise, when facing an app that is unseen during model training, it will be erroneously classified as a known app. In practice, it is impossible to enumerate all apps and collect their traffic for model training since both Android and iOS host more than 3.7 million apps [30], [31]. A possible improvement is conducting open-world recognition and training a one-vs.-rest binary classifier for each app of interest. Open-world recognition is able to handle apps that are unseen during model training because they will be classified as the negative class for each classifier rather than a known app of interest. Nonetheless, open-world recognition is a non-trivial task. On one hand, the adversary needs to involve as many apps as possible to collect their traffic as negative samples for model training. On the other hand, more apps involved as the negative class will result in a more severe sample imbalance, which is commonly regarded as a negative factor for machine learning models.

TABLE I: Comparing PacketPrint with existing AF attacks.

AF Attack	Hidden Destination	Invisible Boundary	App Multiplexing	Open-World Recognition
Conti et al. [11], [13]	×	×	×	×
AppScanner [7], [8]	✓	×	×	✓
NetScope [9]	✓	×	×	×
MIMETIC [14]	✓	×	×	×
Liu et al. [10]	✓	✓	×	×
ActiveTracker [12]	✓	✓	×	×
FlowPrint [6]	×	✓	×	✓
PacketPrint (Our Method)	✓	✓	✓	✓

In this paper, we advance a novel AF attack, named by PacketPrint, to recognize user activities associated with the app of interest by tackling the above challenges. *First*, PacketPrint capitalizes on features extracted from packet size and packet direction instead of destination information, thereby immune to hidden destination. *Second*, we make use of short-term sequential pattern of packet size to achieve traffic segmentation. Without relying on visible boundaries to segment wireless traffic generated by different apps, PacketPrint is capable of automatically locating all possible time periods within which packets are probably generated by the app of interest. *Third*, PacketPrint recognizes app traffic by leveraging structural patterns of packet arrivals. We conduct label-aware feature mapping to extract structural patterns relevant to the app of interest while adaptively ignoring those irrelevant to it. Such as, PacketPrint is noise-tolerant and overcomes app multiplexing. *Lastly*, PacketPrint carries out open-world recognition by leveraging a distinguishable characterization of packet arrivals, which synthesizes structural patterns at different time scales and captures long-term contextual information. It is worth noting that PacketPrint is not designed for zero-shot traffic recognition. Therefore, apps of interest need to be presented during the training stage. Table I demonstrates the properties of PacketPrint and compares it to other AF attacks. We summarize our contributions as follows.

- To the best of our knowledge, PacketPrint is the first AF attack i) capable of dealing with unsegmented encrypted traffic in the open-world setting, and ii) resilient to app multiplexing. We release its source code and the datasets at `https://github.com/jflixjtu/PacketPrint`.
- We address several challenging issues in the design of PacketPrint, including hidden destination, invisible boundary, app multiplexing, and open-world recognition. To solve these technique challenges, we propose two novel models, i.e., sequential XGBoost (S-XGBoost) and hierarchical bag-of-words (H-BoW) model.
- We implement a prototype of PacketPrint and evaluate it through extensive experiments. The experimental results show that PacketPrint consistently outperforms baseline methods. It achieves the average F1-score 0.884 for open-world app recognition and the average F1-score 0.959 for in-app user action recognition. Moreover, PacketPrint is resilient to app multiplexing and transferable to carry out cross-dataset recognition.

II. Overview

A. Threat Model

The adversary considered in this paper captures wireless encrypted traffic using a WiFi sniffer. His *goal* is to recognize user activities associated with the app of interest from wireless encrypted traffic. Based on the training dataset, the adversary can infer either coarse-grained information (i.e., which apps are being used) [7], [8], [14] or fine-grained information (i.e., how a user is interacting with her apps) [9], [11]. For example, if the adversary has the dataset of WhatsApp's traffic, he can decide whether a user is using WhatsApp or not according to the features extracted from WhatsApp's traffic. If the adversary has the dataset of action-specific traffic of WhatsApp, he can identify specific in-app user actions (e.g., making video call on WhatsApp). We conduct experiments to demonstrate them in § VI-C and § VI-G, respectively.

We assume that the adversary can trace back all 802.11 wireless frames that he captures to different mobile devices according to source/destination MAC addresses extracted from the frame header. However, he *cannot* obtain i) source/destination IP addresses from the IP header, ii) source/destination ports from the transport-layer packet header, and iii) the plaintext payload from the application-layer packet, because such information is encrypted and encapsulated in the frame body of 802.11 wireless frames.

Additionally, unlike most WF attacks [21]–[28], we don't assume that app traffic generated by different apps can be reasonably segmented as samples based on inter-packet time gaps, because packets generated by different apps and background services may be strongly mixed together. We also don't assume that apps are executed one at a time as existing works [6], [12], because mobile users may simultaneously use different apps in the wild. Finally, we assume the AF attack in question is conducted in the open-world setting. That is, it needs to handle apps that are unseen during model training. More formally, we define open-world setting along with closed-world setting as follows. Let $\mathcal{S}_T = \{\text{app}_T^i\}_{i=1}^{m_T}$ (resp. $\mathcal{S}_R = \{\text{app}_R^i\}_{i=1}^{m_R}$) be the set comprised of all apps presented during model training (resp. in the recognition stage). In the closed-world setting, we have $\mathcal{S}_R \subseteq \mathcal{S}_T$. In the open-world setting, $\mathcal{S}_R$ might not be a subset of $\mathcal{S}_T$ but it could be. Similar open-world setting has also been considered in existing works [6], [22], [24].

B. Workflow of PacketPrint

Without loss of generality, we assume that $\mathbb{A}$ is an app of interest. PacketPrint recognizes user activities associated with $\mathbb{A}$ from encrypted wireless traffic by tackling four major challenges, including hidden destination (**C1**), invisible boundary (**C2**), app multiplexing (**C3**), and open-world recognition (**C4**). Fig. 1 illustrates how PacketPrint achieves this goal in a pipelined workflow.

• **Traffic Preprocessing** (§ III)**:** The adversary captures encrypted wireless traffic using a WiFi sniffer. PacketPrint sidesteps **C1** because it only extracts features from packet size and packet direction, thereby no need to acquire destination information. To facilitate the downstream analysis, PacketPrint first preprocesses wireless traffic through protocol filtering, packet size normalization, and packet size filtering. Protocol filtering aims to filter out Management-Type and Control-Type frames and only leave Data-Type frames for further analysis, because application-layer data are only encapsulated in Data-Type frames. Packet size normalization aims to transform 802.11 wireless frame size to TCP/IP packet size by removing the frame header and encryption overhead, depending on the encryption protocol employed by the wireless AP. The first two steps are consistent across all apps, while packet size filtering is app-specific.

To conduct packet size filtering, we jointly consider both packet size and packet direction, and construct an app-specific list of directional packet sizes. We filter out the packet without further analysis during the recognition stage if its directional packet size is not in this list. Such a list is derived by solving a combinational optimization problem over the training data of $\mathbb{A}$. Specifically, we specify a lower bound of packet arrival rate associated with $\mathbb{A}$ and meanwhile minimize packet arrival rate associated with other apps. By doing so, PacketPrint can considerably reduce computational overhead for model training and traffic recognition since the packets that need to analyze are significantly reduced. Packet size filtering also brings additional benefits. It mitigates interleaved noises caused by **C3** because many packets generated by other apps have been filtered out.

• **Traffic Segmentation** (§ IV)**:** To solve **C2**, PacketPrint relies on neither transport-layer endpoints nor inter-packet time gaps because such information is unavailable for encrypted wireless traffic. Instead, PacketPrint segments samples for user activity recognition by adaptively locating all possible time periods within which packets are probably generated by $\mathbb{A}$. We refer to these time periods as *target segments*. To this end, we propose a new metric, dubbed *sequential similarity* (S-similarity), to characterize how likely a packet, say p, is generated by $\mathbb{A}$ given its contextual packet size sequence.

We advance a novel model, named by sequential XGBoost (S-XGBoost), to compute p's S-similarity with $\mathbb{A}$. As shown in Fig. 1, S-XGBoost recursively involves p's neighbor packets within an increasing range to extract features that are informative in characterizing sequential patterns of packets generated by $\mathbb{A}$. Such a task is achieved by cascading addictive functions f_j^k for $1 \leq j \leq J$ and $0 \leq k \leq N$, each of which is approximated by a decision tree and recursively trained to gradually minimize the loss function over the training data. We refer to packets that have high S-similarity with $\mathbb{A}$ as *anchor packets*. As shown in Fig. 1, anchor packets are marked with solid heads. Intuitively, anchor packets are expected to be considerably denser within target segments, compared with other time periods. Therefore, we locate target segments via clustering analysis of anchor packets.

• **Traffic Recognition** (§ V)**:** Target segments are identified based on S-similarity, which only captures short-term patterns. However, short-term patterns are insufficient to fully characterize apps' network behaviors because they focus on each individual packet and its short-term sequential context rather than characterize apps' traffic integrally. Therefore, recognition only based on S-similarity inclines to generate more false positives, especially for the open-world setting, when faced with apps that are unseen during model training. In other words, some target segments obtained in the previous step may be irrelevant to $\mathbb{A}$. Therefore, PacketPrint recognizes user activities associated with $\mathbb{A}$ by identifying all target segments that are exactly relevant to $\mathbb{A}$. To this end, we propose another novel metric, dubbed *componential similarity* (C-similarity), to quantify how likely packets within a target segment are generated by $\mathbb{A}$ from the perspective of structural characteristic. We refer to a target segment as a relevant segment of $\mathbb{A}$ if its C-similarity with $\mathbb{A}$ is larger than the preset similarity threshold. We identify user activities associated with $\mathbb{A}$ exactly occur in relevant segments of $\mathbb{A}$.

To compute C-similarity, we devise a hierarchical bag-of-words (H-BoW) model to extract structural features from packet arrival sequences at different time scales. H-BoW employs feature mappers $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ to hierarchically map

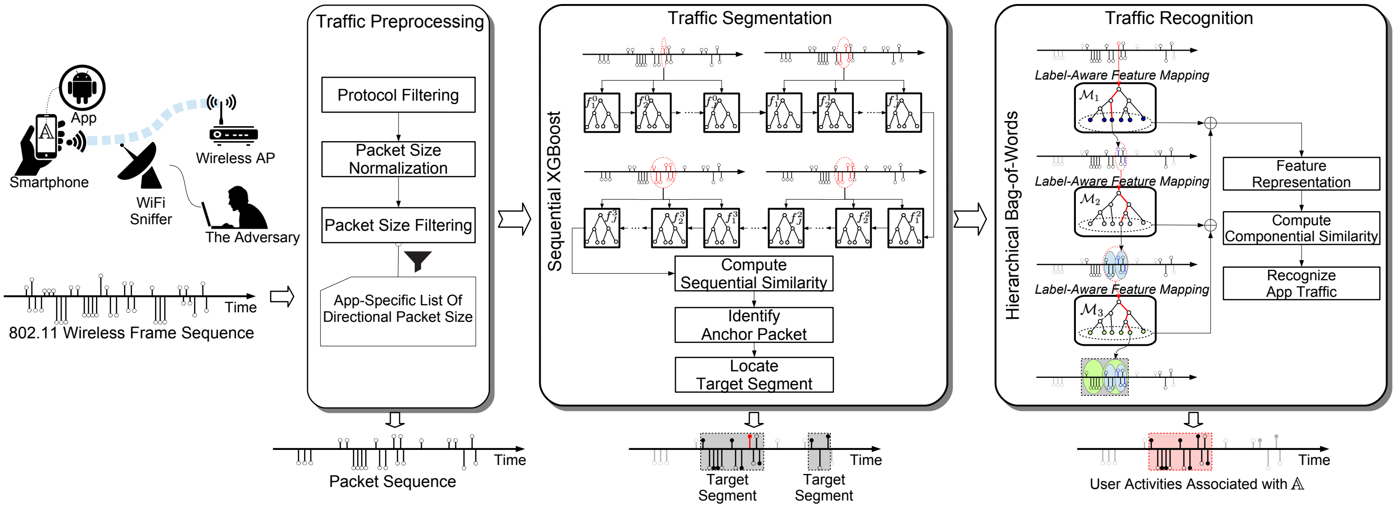


Fig. 1: The architecture of PacketPrint. Each bar on the packet sequence stands for a packet and the height of the bar represents its packet size. Bars with positive (resp. negative) height stands for packets sent from mobile devices (resp. the wireless AP) to the wireless AP (resp. mobile devices).

structural patterns at smaller time scales to their compressed representation, i.e., "words", at a larger time scale. After obtaining "words" through hierarchical feature mapping, we take advantage of combinational optimization to efficiently represent semantic features from these "words" and then train a classifier with a probabilistic output to compute target segments' C-similarity with $\mathbb{A}$. H-BoW's advantages are two-fold. First, feature mappers are constructed in a supervised fashion, thereby automatically ignoring structural patterns irrelevant to $\mathbb{A}$. Such as, H-BoW is noise-tolerant and thus overcomes **C3**. Second, H-BoW generates a comprehensive and distinguishable characterization of $\mathbb{A}$ due to its capability of capturing structural features of packet arrivals at various time scales during a long time period. Benefiting from such a characterization, C-similarity is effective to reduce false positives and address **C4**.

Besides app recognition, PacketPrint can also identify in-app user actions in favor of a more fine-grained inference of user behavior. To recognize in-app user actions of a certain app, PacketPrint needs to collect wireless traffic generated by these user actions and then train an S-XGBoost model and an H-BoW model for each of them. In the recognition stage, PacketPrint first identifies and locates encrypted wireless traffic segments corresponding to this app. From these traffic segments, PacketPrint further recognizes each in-app user action independently.

III. Traffic Preprocessing

PacketPrint first preprocesses wireless traffic through i) protocol filtering, ii) packet size normalization, and iii) packet size filtering.

A. Protocol Filtering

We first trace back 802.11 wireless frames to different end-devices according to source/destination MAC address pair. Next, we filter out Management-Type and Control-Type frames and only leave Data-Type frames for further analysis, since application-layer data are encapsulated in Data-Type frames.

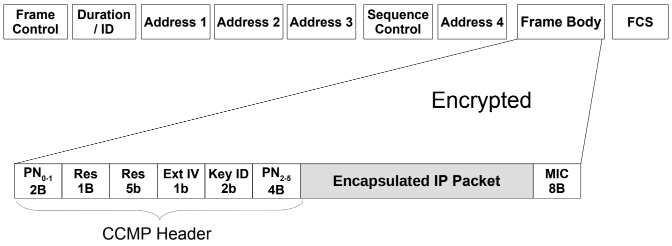


Fig. 2: An example of 802.11 wireless frame with WPA2.

B. Packet Size Normalization

PacketPrint recognizes app traffic by leveraging features extracted from packet sizes. Therefore, the prerequisite for accurate traffic recognition is correctly acquiring packet sizes. In this paper, we define the packet size as the length of IP packet encapsulated in an 802.11 wireless frame instead of the length of the 802.11 wireless frame itself, because the former is independent of security standards, e.g., WEP, WPA, and WPA2, for wireless Internet connections. We extract the packet size by subtracting the encryption overhead from the length of frame body. For example, WPA2 employs CCM mode Protocol (CCMP) as its encryption protocol. CCM is a derivation of CTR mode [32], which is a stream cipher based on AES. Encrypting the plaintext (i.e., encapsulated IP packet) yields the ciphertext with the same length. As shown in Fig. 2, CCMP also generates another 16-byte encryption overhead, including an 8-byte CCMP header and an 8-byte message integrity code (MIC) [33]. Consequently, we extract the packet size from the 802.11 wireless frame with WPA2 by subtracting 16 bytes from the length of frame body. For the 802.11 wireless frame with WEP, the encryption overhead is 8 bytes. For the 802.11 wireless frame with WPA, the encryption overhead is 20 bytes.

C. Packet Size Filtering

The reason why we conduct packet size filtering is two-fold. First, nowadays apps heavily rely on data-intensive network services, such as live streaming and online backup, to facilitate user experience, yielding a huge number of network packets. Processing all packets is often computation-intensive. Packet size filtering is a conceptually simple but effective solution to reduce computational overhead in model training and traffic recognition. Second, to recognize user activities associated with $\mathbb{A}$, network traffic generated by background services and other apps naturally becomes "noise" that potentially undermines recognition accuracy. The situation becomes even worse in face of app multiplexing. Therefore, filtering out packets that are less frequently generated by $\mathbb{A}$ than other apps will mitigate interleaved noises.

We refer to packets sent (resp. received) by a mobile device as outbound (resp. inbound) packets. When conducting packet size filtering, not only packet size but also packet direction are considered in favor of a more effective filtering. Specifically, we denote by $\mathcal{S}_p = \{s_i\}_{i=1}^{M}$ a set comprised of all possible directional packet sizes, where s_i is the ith directional packet size. $|s_i|$ represents the packet size, while $\operatorname{sgn} s_i$ represents the direction, i.e., $\operatorname{sgn} s_i = 1$ (resp. $\operatorname{sgn} s_i = -1$) indicates it is an outbound (resp. inbound) packet. The core task for packet size filtering is constructing an app-specific list of directional packet size $\mathcal{S}_p^{\mathbb{A}} \subset \mathcal{S}_p$, based on which we filter out the packet without further analysis if its directional packet size is not in $\mathcal{S}_p^{\mathbb{A}}$. Intuitively, $\mathcal{S}_p^{\mathbb{A}}$ should exclude packet sizes that are more frequently observed in other apps' traffic as many as possible while reserving sufficient packet sizes of $\mathbb{A}$ for downstream analysis. For packets generated by $\mathbb{A}$ (resp. apps other than $\mathbb{A}$), we denote by $r_+(s_i)$ (resp. $r_-(s_i)$) the average packet arrival rate for the directional packet size s_i. Both $r_+(s_i)$ and $r_-(s_i)$ can be readily estimated from the training data of $\mathbb{A}$. We formulate the construction of $\mathcal{S}_p^{\mathbb{A}}$ as a combinational optimization problem:

$$\begin{aligned} &\min_{z_1,z_2,\dots,z_{|\mathcal{S}_p|}} \sum_{s_i\in\mathcal{S}_p} z_i\cdot r_-(s_i), \\ &\quad \text{st.} \sum_{s_i\in\mathcal{S}_p} z_i\cdot r_+(s_i) \geq \min\left\{\sum_{s_i\in\mathcal{S}_p} r_+(s_i), R_{\min}\right\}, \qquad (1) \\ &\qquad \sum_{i=1}^{|\mathcal{S}_p|} z_i \geq M_{\min}^{s},\ z_i\in\{0,1\}, \end{aligned}$$

where z_i is a binary variable indicating whether the directional packet size s_i will be involved in $\mathcal{S}_p^{\mathbb{A}}$, $R_{\min}$ is the expected minimum packet arrival rate of $\mathbb{A}$, and $M_{\min}^{s}$ is the minimum number of different directional packet sizes in $\mathcal{S}_p^{\mathbb{A}}$. In this paper, we set $R_{\min} = 2$ pkt/s and $M_{\min}^{s} = 200$ by default.

The combinational optimization in Eqn. (1) can be solved by integer programming, such as branch and price [34]. To speed up the construction of $\mathcal{S}_p^{\mathbb{A}}$, we solve Eqn. (1) in a greedy fashion. Specifically, $\mathcal{S}_p^{\mathbb{A}}$ is initialized to contain all directional packet sizes in $\mathcal{S}_p$. We iteratively remove directional packet sizes in $\mathcal{S}_p^{\mathbb{A}}$ with the maximum ratio of $r_-(s_i)$ to $r_+(s_i)$ without violating constraints in Eqn. (1). The above process is summarized as Algorithm 1 in § A-B.

IV. Traffic Segmentation

Traffic segmentation aims to locate target segments, namely all possible time periods within which packets are probably generated by $\mathbb{A}$. We propose a new metric, sequential similarity (S-similarity), to identify anchor packets that are further used to locate target segments. To compute S-similarity, we advance a novel model, dubbed sequential XGBoost (S-XGBoost). It incrementally computes S-similarity of a packet by recursively involving its neighbor packets within an increasing range to extract features that are informative in characterizing sequential patterns of network traffic generated by $\mathbb{A}$.

A. Sequential Similarity

Before a formal definition of S-similarity, we first define N-gram sequential context. Let $\mathbf{p} = (p_1, p_2, \dots, p_m)$ be a sequence of network packets obtained through traffic preprocessing (§ III). We denote by x_i the directional packet size of p_i. Besides, we denote by y_i a variable to indicate whether or not p_i is generated by $\mathbb{A}$. If p_i is generated by $\mathbb{A}$ we have $y_i = 1$ and otherwise $y_i = 0$. We refer to $\mathbf{x}_i^N = (x_{i-N}, \dots, x_i, \dots, x_{i+N})$ as the N-gram sequential context of p_i, which captures short-term but ordered contextual information for p_i.

Definition 1. *For a packet p_i, its N-gram sequential similarity with $\mathbb{A}$ is denoted by $\Phi_N^{\mathbb{A}}(p_i)$ and defined as the probability that it is generated by $\mathbb{A}$ given p_i's N-gram sequential context. Formally, we have $\Phi_N^{\mathbb{A}}(p_i) = Pr\left(y_i = 1|\mathbf{x}_i^N\right)$.*

At first glance, $\Phi_N^{\mathbb{A}}(p_i)$ can be computed by training a classifier to output the probability that p_i is generated by $\mathbb{A}$ given the feature vector $\mathbf{x}_i^N$. However, such a straightforward solution has two major drawbacks.

D1: It is susceptible to noises caused by app multiplexing and background traffic, because $\mathbf{x}_i^N$ is order-sensitive and thus noises interleaved in the sequential context will change the order of features in $\mathbf{x}_i^N$.

D2: It is easy to overfit training data, because the feature space of N-gram sequential context can be very sparse for real-world packet arrivals.

B. Sequential XGBoost

In this paper, we propose sequential XGBoost (S-XGBoost) to overcome drawbacks **D1** and **D2**. The vanilla version of XGBoost [35] is a tree boosting system where additive functions approximated by binary decision trees are recursively added to minimize the loss function. S-XGBoost generalizes this idea to further minimize the loss function by taking advantage of gradually extended sequential context.

• **Computing S-Similarity.** S-XGBoost computes S-similarity by recursively involving $\mathbf{x}_i^0, \mathbf{x}_i^1, \dots, \mathbf{x}_i^N$. Specifically, $\Phi_N^{\mathbb{A}}(p_i)$ is computed by $\Phi_N^{\mathbb{A}}(p_i) = \frac{e^{q(p_i)}}{1+e^{q(p_i)}}$, where $q(p_i)$ is the cumulative confidence for $y_i = 1$. We compute $q(p_i) = \sum_{k=0}^{N} F_k(\mathbf{x}_i^k)$, where $F_k(\mathbf{x}_i^k)$ denotes the confidence contributed by $\mathbf{x}_i^k$. In here, $F_k(\mathbf{x}_i^k) = \sum_{j=1}^{J} f_j^k(\mathbf{x}_i^k)$ is constructed by synthesizing multiple additive functions, each of which is approximated by a decision tree. $f_j^k(\cdot)$ is the jth decision tree, which contains n_j^k leaves. Let $\mathcal{C}_t$ be the tth leaf on this decision tree and v_t be the value associated with $\mathcal{C}_t$. We obtain $f_j^k(\mathbf{x}_i^k) = v_t$ if $\mathbf{x}_i^k$ falls into $\mathcal{C}_t$.

It is worth noting that $F_k(\mathbf{x}_i^k)$ is recursively added to minimize the residual loss from $F_0(\mathbf{x}_i^0), F_1(\mathbf{x}_i^1), \dots, F_{k-1}(\mathbf{x}_i^{k-1})$, enabling an incremental improvement of estimation accuracy.

As such, S-XGBoost is inclined to exploit contextual information closer to p_i, which tends to be more resilient to noises than that far from p_i. For example, if there exist noise packets in $\mathbf{x}_i^k$, sequential context at a larger range, i.e., $\mathbf{x}_i^{k+1}, \mathbf{x}_i^{k+2}, \ldots, \mathbf{x}_i^N$, will definitely be influenced but not vice versa. By doing so, S-XGBoost mitigates the impact of noises (**D1**).

- **Recursive Model Training.** S-XGBoost is trained in a recursive fashion. When training the decision tree $f_j^k(\cdot)$, we minimize the loss function

$$\mathcal{L}_j^k = \sum_{i=1}^{m} L\left(y_i, \sum_{r=0}^{k-1} F_r(\mathbf{x}_i^r) + \sum_{s=0}^{j-1} f_s^k(\mathbf{x}_i^k) + f_j^k(\mathbf{x}_i^k)\right) + \Theta(f_j^k), \quad (2)$$

where $L(\cdot)$ is the logistic loss and

$$\Theta(f_j^k) = \gamma n_j^k + \frac{1}{2}\lambda \sum_{t=1}^{n_j^k} \|v_t\|^2 \quad (3)$$

is a regularization term that penalizes the complexity of f_j^k. Introducing the regularization term effectively reduces the risk of overfitting (**D2**) because i) the first term in Eqn. (3) allows the decision tree to further split only if the splitting results in substantial loss reduction and ii) the second term in Eqn. (3) guarantees the smoothness of $F_k(\cdot)$. To speed up the training process, we minimize Eqn. (2) in a greedy fashion (see § A-A for details).

- **Identifying Anchor Packet.** For every packet in $\mathbf{p} = (p_1, p_2, \ldots, p_m)$, S-XGBoost computes its S-similarity with $\mathbb{A}$. Given an S-similarity threshold $\phi_{\min}$, we refer to p_i is an anchor packet if $\Phi_N^{\mathbb{A}}(p_i) \geq \phi_{\min}$. Anchor packets play an important role in locating target segments.

C. Target Segment Location

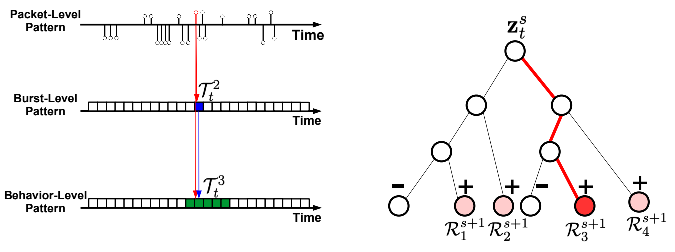
Densely distributed anchor packets are strong indicators for the presence of user activities associated with $\mathbb{A}$. Therefore, we locate target segments by clustering anchor packets. Specifically, we use hierarchical agglomerative clustering because it obviates the need to preset the cluster number, which is unknown a prior. We choose single-linkage distance metric (i.e., the minimum distance) to guide cluster merging. Given a clustering threshold ϵ, clusters recursively merge in a bottom-up manner until the distance between all clusters is larger than ϵ. We denote by $\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_m$ the resulting clusters. For every cluster, the time period corresponding to it is identified as a target segment for further analysis.

V. Traffic Recognition

To recognize user activities associated with $\mathbb{A}$, PacketPrint identifies all target segments that are exactly relevant to $\mathbb{A}$ and excludes those irrelevant to $\mathbb{A}$. To this end, we propose a metric, dubbed *componential similarity* (C-similarity), to quantify how likely packets within a target segment are generated by $\mathbb{A}$. To compute C-similarity, we devise a hierarchical bag-of-words (H-BoW) model that extracts structural features from packet arrival sequences at different time scales. By leveraging C-similarity, PacketPrint carries out app traffic recognition.

A. Componential Similarity

Definition 2. *For a target segment $\mathbf{s}_t$, its componential similarity with $\mathbb{A}$ is denoted by $\Psi^{\mathbb{A}}(\mathbf{s}_t)$ and defined as the probability that there exist packets in $\mathbf{s}_t$ generated by $\mathbb{A}$. Formally, we have $\Psi^{\mathbb{A}}(\mathbf{s}_t) = Pr\left(\{p_j \in \mathbf{s}_t : y_j = 1\} \neq \varnothing | \mathbf{s}_t\right)$.*



(a) Hierarchical sliding windows. (b) Label-aware feature mapping.

Fig. 3: Examples of hierarchical sliding windows and label-aware feature mapping.

The core challenge to compute $\Psi^{\mathbb{A}}(\mathbf{s}_t)$ is how to effectively extract features from $\mathbf{s}_t$ in response to the following requirements:

R1: *Distinguishable characterization*. To reduce false positive risk in an open-world recognition, features extracted from $\mathbf{s}_t$ should be informative in distinguishing $\mathbb{A}$ from other apps (even unseen apps during model training).

R2: *Noise-tolerance*. Features extracted from $\mathbf{s}_t$ should be robust against interleaved noises caused by app multiplexing and background traffic.

R3: *Overfitting-resilience*. Features need to be represented in a compact form to avoid the downstream model overfitting training data.

B. Hierarchical Bag-of-Words Model

We propose a hierarchical bag-of-words (H-BoW) model in response to the above requirements. It is comprised of two key components, i.e., label-aware feature mapping and optimization-based feature representation.

- **Label-Aware Feature Mapping.** H-BoW extracts features from structural patterns of packet arrivals at gradually expanding time scales. Specifically, H-BoW considers i) packet-level patterns, ii) burst-level patterns, and iii) behavior-level patterns. The packet-level pattern describes directional packet sizes of a packet together with the closest packets before and after it, while the burst-level pattern concerns the combination of various packet-level patterns within burst transmission. The behavior-level patterns are related to user operations and characterized by synthesizing packet-level patterns and burst-level patterns within user operations.

As shown in Fig. 1, H-BoW recursively maps structural patterns at a smaller time scale to a categorical variable, i.e., a "word", at a larger time scale. For simplicity, we refer to packet-level patterns, burst-level patterns, and behavior-level patterns as structural patterns at the first, second, and third time scales respectively. We denote by $\mathcal{M}_s$ the feature mapper for patterns at the sth time scale. Fig. 3 illustrates how this process is conducted. As shown in Fig. 3(a), the pattern at the first time scale is derived from each packet while we capture patterns at higher time scales using sliding windows. Specifically, we set sliding windows $\mathcal{T}_t^2$ for the second time scale (i.e., burst pattern) and sliding windows $\mathcal{T}_t^3$ for the third time scale (i.e., behavior pattern). We empirically specify the window size of $\mathcal{T}_t^2$ (resp. $\mathcal{T}_t^3$) to be 1 second (resp. 5 seconds). For both $\mathcal{T}_t^2$ and $\mathcal{T}_t^3$, sliding step is set to be 1 second. By leveraging hierarchical sliding windows, H-BoW is capable of extracting structural patterns of packet arrivals at various time scales during a long time period, enabling a distinguishable characterization of $\mathbb{A}$'s traffic (**R1**).

To achieve **R2**, we propose label-aware feature mapping (LFM), which maps structural patterns of $\mathbb{A}$'s packet arrivals at a smaller time scale to categorical variables, i.e., "words" at a larger time scale, while adaptively filtering out structural patterns associated with other apps. As shown in Fig. 3(b), LFM is based on a decision tree. Next, we will describe how LFM is trained and why it makes H-BoW noise-tolerant.

• **Recursive LFM Training.** LFMs are recursively trained from small time scales to the large. We initiate an empty vocabulary $\mathcal{V}$ to record words generated afterward. We denote by $\mathbf{z}_t^s$ a feature vector that describes the structural pattern of p_t (resp. $\mathcal{T}_t^s$) for $s = 1$ (resp. $s > 1$). Recall that packet-level pattern describes directional packet sizes of a packet together with the closest packets before and after it. Formally, for the packet p_t, we characterize its structural pattern using p_t's 1-gram sequential context (see § IV-A) and obtain $\mathbf{z}_t^1 = \mathbf{x}_t^1$. As for a sliding window, say $\mathcal{T}_t^s$, we characterize its structural pattern using tf-idf representation of words in $\mathcal{T}_t^s$. To this end, we construct a set $\mathscr{T}_{train}^s$ comprised of sliding windows at the sth time scale extracted from packet arrival sequences in $\mathbb{A}$'s training dataset. Formally, we represent the structural pattern of $\mathcal{T}_t^s$ using a $|\mathcal{V}|$-dimensional feature vector $\mathbf{z}_t^s$. $\mathbf{z}_t^s(k)$ is computed by

$$\mathbf{z}_t^s(k) = \mathrm{tf}_b(w_k, \mathcal{T}_t^s) \cdot \mathrm{idf}(w_k, \mathscr{T}_{train}^s), \tag{4}$$

where

$$\mathrm{tf}_b(w_k, \mathcal{T}_t^s) = \mathbf{1}_{W(\mathcal{T}_t^s)}(k), \tag{5}$$

$$\mathrm{idf}(w_k, \mathscr{T}_{train}^s) = \log \frac{|\mathscr{T}_{train}^s|}{|\{\mathcal{T}_i^s \in \mathscr{T}_{train}^s : k \in W(\mathcal{T}_t^s)\}|}, \tag{6}$$

w_k is the kth word in the current vocabulary $\mathcal{V}$, $W(\mathcal{T}_t^s)$ is a function that returns a set comprised of indices of all words within $\mathcal{T}_t^s$, and $\mathbf{1}_{W(\mathcal{T}_t^s)}(\cdot)$ is an indicator function. We employ Boolean "frequencies" when computing term frequency (tf) in favor of a more robust representation of structural patterns. After obtaining the feature vectors $\mathbf{z}_t^s$ for $1 \le t \le m_s$, we construct a set $\mathscr{D}_{train}^s = \{(\mathbf{z}_t^s, y_t^s)\}_{t=1}^{m_s}$, where y_t^s is the label for $\mathbf{z}_t^s$. If there is any packet in $\mathcal{T}_t^s$ generated by $\mathbb{A}$, we have $y_t^s = 1$ (positive sample) and otherwise $y_t^s = 0$ (negative sample). We train a decision tree over $\mathscr{D}_{train}^s$. This decision tree acts as $\mathcal{M}_s(\cdot)$, i.e., the feature mapper at the sth time scale. Fig. 3(b) illustrates the basic idea of feature mapper. $\mathbf{z}_t^s$ is mapped to a unique leaf, i.e., $\mathcal{R}_3^{s+1}$ on the decision tree. A leaf dominated by positive samples is referred to as a *positive leaf* and otherwise a *negative leaf.* All positive leaves constitute a set $\mathcal{R}_+^{s+1}$ and they will be added to the vocabulary as new words, whereas negative leaves will be skipped. We summarize the process of recursive LFM training as Algorithm 2 in § A-C.

Label-aware feature mapping facilitates an effective extraction of structural features. First, it focuses on positive leaves and ignores negative leaves, thereby automatically filtering out most structural patterns that frequently appear for apps other than $\mathbb{A}$. As such, H-BoW is noise-tolerant (**R2**). Second, when mapping a feature vector $\mathbf{z}_t^s$ to a categorical variable, the feature space of $\mathbf{z}_t^s$ is split into different regions with adaptive resolution to maximize the distinguishability between $\mathbb{A}$ and other apps. It again enhances H-BoW's capabilities for **R1**.

• **Feature Representation.** The vocabulary $\mathcal{V}$ may contain hundreds of words. If words in $\mathcal{V}$ are directly used as features, the downstream model is prone to overfitting training data due to the sparsity of samples in such a high dimensional feature space. To reduce the overfitting risk, we derive a more compact and efficient feature representation based on combinational optimization. We denote by $\mathbf{z}_t$ a n_f-dimensional feature vector of the target segment $\mathbf{s}_t$. The feature vector $\mathbf{z}_t$ characterizes the presence/absence of words within $\mathbf{s}_t$. Recall that words in $\mathcal{V}$ are indicators of $\mathbb{A}$. Intuitively, we expect they appear in positive samples (i.e., the presence of user activities associated with $\mathbb{A}$) as frequent as possible while they appear in negative samples (i.e., the absence of user activities associated with $\mathbb{A}$) as rare as possible. Therefore, we expect positive samples' feature vectors should contain much more non-zero elements than those of negative samples. To optimize the feature representation, we collect training samples from historical packet arrival sequences generated by $\mathbb{A}$ as well as other apps. Let $\tilde{\mathbf{s}}_t$ be a packet arrival sequence and $\tilde{y}_t$ be the label of $\tilde{\mathbf{s}}_t$. If $\tilde{\mathbf{s}}_t$ is generated by $\mathbb{A}$, we have $\tilde{y}_t = 1$ (positive sample) and otherwise $\tilde{y}_t = 0$ (negative sample). We denote by $\mathscr{D}_{train}^+$ (resp. $\mathscr{D}_{train}^-$) the set comprised of all positive samples (resp. negative samples). Combining $\mathscr{D}_{train}^+$ and $\mathscr{D}_{train}^-$ yields $\mathscr{D}_{train} = \mathscr{D}_{train}^+ \cup \mathscr{D}_{train}^-$.

We formulate the feature representation as the following optimization problem.

$$\begin{aligned} \min_{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_{|\mathcal{V}|}} \; & -\alpha \bar{Q}_+ + \bar{Q}_-, \\ \text{st.} \quad \bar{Q}_+ &= \frac{\sum_{\tilde{\mathbf{s}}_t \in \mathscr{D}_{train}^+} \left\| \sum_{k \in W(\tilde{\mathbf{s}}_t)} \mathbf{c}_k \right\|_0}{|\mathscr{D}_{train}^+|}, \\ \bar{Q}_- &= \frac{\sum_{\tilde{\mathbf{s}}_t \in \mathscr{D}_{train}^-} \left\| \sum_{k \in W(\tilde{\mathbf{s}}_t)} \mathbf{c}_k \right\|_0}{|\mathscr{D}_{train}^-|}, \\ & \mathbf{c}_j \in \{0,1\}^{n_f}, \; \|\mathbf{c}_j\|_1 \le 1, \; \forall j \in [1, |\mathcal{V}|], \end{aligned} \tag{7}$$

where $\mathbf{c}_j$ (for $1 \le j \le |\mathcal{V}|$) is a n_f-dimensional binary vector, α is a parameter that balances the importance between positive samples and negative samples, and $W(\tilde{\mathbf{s}}_t)$ is a function that returns a set comprised of indices of all words within $\tilde{\mathbf{s}}_t$. Essentially, Eqn. (7) aims at the optimal merging of words to maximize the difference between positive samples and negative samples. $\mathbf{c}_j$ describes how w_j will be merged with other words. If $\mathbf{c}_j(k) = 1$, w_j will become the kth new word after word merging. If all elements in $\mathbf{c}_j$ equal 0, w_j will be discarded.

The combinational optimization in Eqn. (7) can be solved by integer programming, such as branch and price [34]. To speed up the training process, we solve it in a greedy fashion. Specifically, we first initialize $\mathbf{c}_j$ for $1 \le j \le |\mathcal{V}|$ as a zero vector. Next, we greedily search for the vector $\mathbf{c}_{j^*}$ and k^* that result in maximum loss reduction when setting $\mathbf{c}_{j^*}(k^*) = 1$. Such a process repeats until all words in $\mathcal{V}$ are merged into at most n_f new words or we cannot find a $\mathbf{c}_j$ and k that can reduce the loss function. We elaborate this process as Algorithm 3 in § A-D. Given $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_{|\mathcal{V}|}$, one can readily derive a new vocabulary $\mathcal{V}' = \{w'_k\}_{k=1}^{n_f}$ after merging words in $\mathcal{V}$. We compute $\mathbf{s}_t$'s feature vector $\mathbf{z}_t$ by

$$\mathbf{z}_t(k) = \mathrm{tf}_f(w'_k, \mathbf{s}_t) \cdot \mathrm{idf}(w'_k, \mathscr{D}_{train}), \tag{8}$$

where

$$\mathrm{tf}_f(w'_k, \mathbf{s}_t) = \sum_{j \in W(\mathbf{s}_t)} \mathbf{c}_j(k), \tag{9}$$

$$\mathrm{idf}(w'_k, \mathscr{D}_{train}) = \log \frac{|\mathscr{D}_{train}|}{|\{\tilde{\mathbf{s}}_t \in \mathscr{D}_{train} : \mathrm{tf}_f(w'_k, \tilde{\mathbf{s}}_t) \geq 1\}|}. \tag{10}$$

Optimization-based word merging facilitates an efficient feature representation by substantially reducing feature dimension number, i.e., $n_f \ll |\mathcal{V}|$, and thus solve **R3**. Benefits are two-fold. First, such a compact feature representation effectively reduces the overfitting risk during model training. For this reason, we employ raw "frequency" in Eqn. (9) instead of Boolean "frequency" that may lose some information. Second, word merging also reduces computational overhead.

C. App Traffic Recognition

• **Computing C-Similarity.** For target segments located via traffic segmentation (see § IV-C), we can compute their C-similarity with $\mathbb{A}$ by training a classifier with a probabilistic output, e.g., random forest and Logistic regression, which is denoted by f_C. In the training stage, we extract the feature vector for each sample in $\mathscr{D}_{train}$ based on the proposed H-BoW model. We then train f_C over $\mathscr{D}_{train}$ to predict the probability $f_C(\tilde{\mathbf{z}}_t) = Pr(\tilde{y}_t = 1|\tilde{\mathbf{z}}_t)$, where $\tilde{\mathbf{z}}_t$ is the feature vector of the sample $\tilde{\mathbf{s}}_t$. In the recognition stage, we extract the feature vector $\mathbf{z}_t$ for the target segment $\mathbf{s}_t$ according to Eqn. (8) and compute its C-similarity with $\mathbb{A}$ by $\Psi^{\mathbb{A}}(\mathbf{s}_t) = f_C(\mathbf{z}_t)$.

Given a C-similarity threshold $\psi_{\min}$, we recognize that user activities associated with $\mathbb{A}$ occur during $\mathbf{s}_t$ if $\Psi^{\mathbb{A}}(\mathbf{s}_t) \geq \psi_{\min}$.

VI. Evaluation

We evaluate PacketPrint through extensive experiments. Our experiments aim at answering five research questions.

- **RQ1:** Can PacketPrint accurately recognize apps of interest from encrypted wireless traffic?
- **RQ2:** How will the recognition accuracy be affected by practical network conditions, such as sniffer packet loss and congestion-related packet loss?
- **RQ3:** To what extent is PacketPrint resilient to noise packets due to app multiplexing?
- **RQ4:** Can PacketPrint trained on app traffic triggered by automatic test tools recognize human-driven app traffic?
- **RQ5:** Can PacketPrint recognize more fine-grained in-app user actions?

A. Data Collection

To answer the above research questions, we collect wireless app traffic in our testbed.

• **802.11 Traffic Sniffing.** As illustrated in Fig. 1, we set up the wireless AP using a TP-LINK WiFi router, which employs WPA2 for traffic encryption. All mobile devices access Internet via this wireless AP. We deploy the WiFi sniffer in a Windows PC equipped with Ralink 802.11ac wireless LAN cards, which support over-the-air sniffing. The sniffer captures and saves 802.11 traffic with the aid of Omnipeek [36]. To carry out protocol filtering, it further filters out Management-Type and Control-Type frames with a C# library PacketDotNet.Ieee80211. A wireless AP may work on different WiFi channels, while a single wireless LAN card cannot simultaneously sniff on different WiFi channels. To facilitate the data collection, we specify the wireless AP to work on a fixed WiFi channel 149 and sniff on this channel. Omnipeek also supports sniffing on multiple channels by aggregating results from multiple wireless LAN cards, each of which sniffs on a single channel.

• **Dataset Construction.** To answer RQ1 to RQ3, we construct the Monkey500 dataset; to answer RQ4, we construct the Human100 dataset. As for RQ5, we construct the IUA dataset.

Monkey500 dataset: We download 500 apps from Google Play to generate app traffic. The selected apps are popular according to the rank in Google play store. Besides, these apps fall into 29 different categories to guarantee a reasonable representativeness. When selecting apps, we skip various mobile browsers, because network traffic generated by them is essentially from mobile websites. To avoid potential legal risks, we also skip apps that need to sign up with a credit card. We generate app traffic instances by leveraging automatic test tools Monkey [37] and RERAN [38]. Monkey can imitate various users' behaviors such as clicking the screen and inputting texts, while RERAN can record and replay user behaviors on Android smartphones. When starting an app, initialization, such as login to account and preference settings, is often a necessary step. Consequently, directly running monkey testing may generate lots of useless user events if the test is stuck in the initialization step. To avoid this situation, we manually record scripts that complete login and preference settings with the aid of RERAN and replay these scripts before monkey tests. For every app, we generate 50 traffic instances. To generate each traffic instance, we run the app on Google Pixel 2 smartphones with Monkey, which generates pseudo-random streams of user events to simulate user operations. The maximum number of user events is set to be 5000. To analyze how network congestion influences the performance of PacketPrint, we generate another 10 traffic instances for each app in a setting where we deliberately induce network congestion by limiting the inbound/outbound network speed on the wireless AP.

To evaluate PacketPrint, we collect 802.11 traffic sniffed over the air. Additionally, we also capture and store inbound and outbound TCP/IP traffic of smartphones with Tcpdump [39]. Meanwhile, log files (in Android, we call them logcat, the same as below) are saved. TCP/IP traffic and logcat are merely used for establishing an accurate ground truth for 802.11 traffic, i.e., which app generates it. Specifically, we first correlate 802.11 wireless frames with TCP/IP packets by comparing packet sizes (see § III-B) and timestamps. Next, we correlate TCP/IP packets with apps by comparing Internet-layer and transport-layer endpoints of a packet with socket operations recorded in logcat. By doing so, we can distinguish 802.11 wireless frames exactly generated by the app of interest from those generated by background services. We elaborate the above process in § B-A.

Human100 dataset: To evaluate if PacketPrint trained based on Monkey-generated app traffic can also recognize human-generated app traffic, we construct a dataset containing app traffic generated by human users. Specifically, we randomly choose 100 out of 500 apps and recruit 5 volunteers to manually operate these apps to generate 5×100 traffic instances, each of which lasts for about 5 minutes.

IUA dataset: Besides user activities associated with the app of interest, we also evaluate whether PacketPrint is competent in recognizing more fine-grained in-app user actions. To this end, we construct an in-app user action (IUA) dataset, which contains in-app user actions of 13 popular mobile apps, such as Youtube, Netflix, Wechat, and WhatsApp. We recruit 5 volunteers to manually trigger user actions listed in Table IV and generate 5 traffic instances for each of them. We collect 802.11 traffic generated by different user actions. Meanwhile, volunteers were required to record the start and end time of their operations to establish the ground truth.

Note that we obtained IRB approval for the experiments involving volunteers.

B. Baseline

To the best of our knowledge, PacketPrint is the first AF attack i) capable of dealing with unsegmented encrypted traffic in the open-world setting, and ii) resilient to app multiplexing. Existing AF attacks [6]–[10], [40]–[51] have different threat models from our work. For example, some AF attacks [7]–[9], [14] assume network flows can be extracted. [6] considers destination features and [52], [53] only handle unencrypted data. To handle unsegmented encrypted traffic, a naive solution is dividing traffic into sliding windows with the size of τ, extracting features for each window, and then training a classifier to recognize user activities within each window. Although it is difficult to make a direct comparison between PacketPrint and existing AF attacks due to different threat models, features used in the existing AF attacks can also be used in the above sliding window approach. Specifically, we reuse features of two state-of-the-art AF attacks, i.e., AppScanner [7] and NetScope [9], to implement baseline methods. As for the classifier, we compare various classifiers that achieve the best accuracy in existing AF attacks and WF attacks, including random forest (RF), support vector machine (SVM), k-nearest neighbors (KNN) classifier, and find RF classifier performs the best for almost all experiments. Therefore, we choose the RF classifier with a probabilistic output to implement baseline methods and specify a recognition threshold for traffic recognition. Additionally, we also set varied sliding window sizes, i.e., $\tau = 1, 10, 100$ seconds, for baseline methods.

C. Closed-World Setting vs. Open-World Setting

In this experiment, we evaluate PacketPrint in both closed-world and open-world settings.

• **Experimental Setup.** This experiment is based on *Monkey500* dataset. Since PacketPrint trains an S-XGBoost and a H-BoW model for every individual app, we construct a training dataset and testing dataset for each app. Specifically, we randomly distribute traffic instances of an app into two subsets: training subset (40 instances) and testing subset (10 instances). To construct the training dataset of an app, say $\mathbb{A}$, besides training subset of itself, we randomly choose 40 apps other than $\mathbb{A}$ and involve traffic instances in their training subset as negative samples. There are two testing datasets for closed-world and open-world settings, respectively. For the former, apps involved in the testing are the same as those in the training. For the latter, we construct the testing dataset by involving $\mathbb{A}$ and another 40 apps, which are absent in the training dataset. In other words, apps considered for negative samples in the testing are *completely* different from those for model training. To mimic the random use of apps by humans in the wild, we shuffle traffic instances in the testing dataset and arrange them in a random order. To synthesize unsegmented traffic for testing, we concatenate all traffic instances (410 traffic instances) and meanwhile record the time slots when user activities associated with different apps start and stop. We compare PacketPrint with baseline methods. Recall that all these methods recognize time periods/windows where user activities associated with $\mathbb{A}$ fall into. We refer to the median arrival time of packets in these time periods/windows as their reference time. If the reference time falls into a time slot of $\mathbb{A}$ (resp. other apps), we report a true positive (resp. false positive) case. If no reference time falls into a time slot of $\mathbb{A}$ (resp. other apps), we report a false negative (resp. true negative) case. We elaborate on the process of hyperparameter tuning in § B-B.

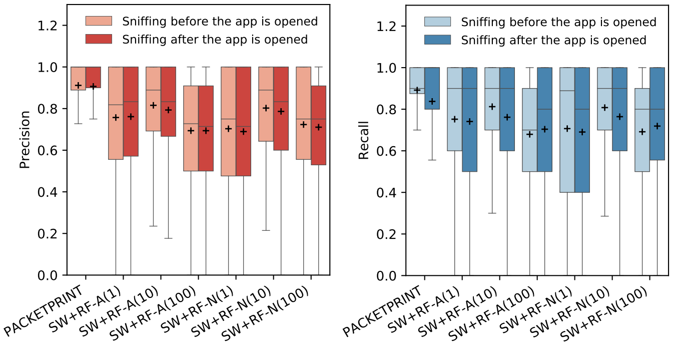
• **Result.** Table II reports the experimental results. For both closed-world and open-world settings, PacketPrint consistently outperforms all baseline methods in terms of precision, recall, and F1-score. Compared with the baseline method that achieves the highest F1-score, PacketPrint improves the average F1-score from 0.822 to 0.935 in the closed-world setting, while it improves the average F1-score from 0.778 to 0.884 in the open-world setting. Another observation is that all methods perform better in the closed-world setting due to a higher precision. The root cause is false positives are more likely generated when facing apps that are unseen during model training in the open-world setting.

• **Initially Opened App.** In a practical AF attack, the adversary may start to sniff wireless traffic after the app of interest is opened and thus cannot obtain the entire app traffic. It is particularly common for apps with long-running behaviors. We further explore whether PacketPrint is still effective in this scenario, i.e., recognizing the app of interest when the user is already using the app. Specifically, we assume that PacketPrint starts to recognize the app of interest after this app has been opened for 5 seconds. Here we only consider the open-world setting because it is more challenging than the closed-world setting. Fig. 4 reports the experimental results. PacketPrint achieves similar average precision no matter whether the app of interest is initially opened or not. The same situation happens for baseline methods. Unlike the precision, we observe a slight recall decline for PacketPrint, SW+RF-A(10), and SW+RF-N(10), if the app is initially opened. For example, the average recall for PacketPrint drops from 0.892 to 0.838. It reveals that app traffic during initialization phase may exhibit app-specific patterns that are informative to recognize apps.

Answer to RQ1: PacketPrint achieves reasonable accuracy in recognizing apps of interest from encrypted wireless traffic. The average F1-score is 0.935 in the closed-world and 0.884 in the open-world setting.

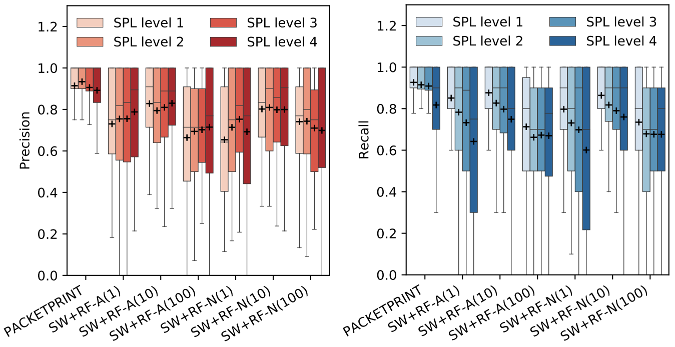
TABLE II: Evaluating PacketPrint in closed-world setting and open-world setting (mean±standard deviation).

Scenario Setting	Metric	SW+RF (AppScanner features)			SW+RF (NetScope features)			PACKETPRINT
		$\tau = 1$	$\tau = 10$	$\tau = 100$	$\tau = 1$	$\tau = 10$	$\tau = 100$	(Our method)
Closed world	Precision	0.860 ± 0.170	0.893 ± 0.144	0.816 ± 0.191	0.775 ± 0.230	0.846 ± 0.178	0.819 ± 0.192	$\mathbf{0.985 \pm 0.053}$
	Recall	0.748 ± 0.303	0.811 ± 0.239	0.678 ± 0.279	0.708 ± 0.319	0.812 ± 0.232	0.682 ± 0.291	$\mathbf{0.909 \pm 0.160}$
	F1-score	0.742 ± 0.244	0.822 ± 0.187	0.698 ± 0.234	0.667 ± 0.249	0.794 ± 0.181	0.690 ± 0.247	$\mathbf{0.935 \pm 0.130}$
Open world	Precision	0.757 ± 0.245	0.816 ± 0.205	0.694 ± 0.253	0.703 ± 0.267	0.802 ± 0.215	0.723 ± 0.241	$\mathbf{0.911 \pm 0.150}$
	Recall	0.752 ± 0.299	0.812 ± 0.236	0.679 ± 0.282	0.707 ± 0.321	0.808 ± 0.237	0.692 ± 0.282	$\mathbf{0.892 \pm 0.163}$
	F1-score	0.681 ± 0.242	0.778 ± 0.190	0.629 ± 0.232	0.613 ± 0.244	0.764 ± 0.195	0.644 ± 0.235	$\mathbf{0.884 \pm 0.151}$



(a) Precision. (b) Recall.

Fig. 4: Recognizing initially opened apps. The mean is marked with "+". For simplicity, SW+RF-A(x) (resp. SW+RF-N(x)) stands for the baseline method using AppScanner (resp. NetScope) features ($\tau = x$ seconds).



(a) Precision. (b) Recall.

Fig. 5: The impact of sniffer packet loss. SPL levels 1 to 4 represent an increasing degree of sniffer packet loss (SPL). SPL level k contains apps with a SPL rate between $25(k-1)$th percentile and $25k$th percentile.

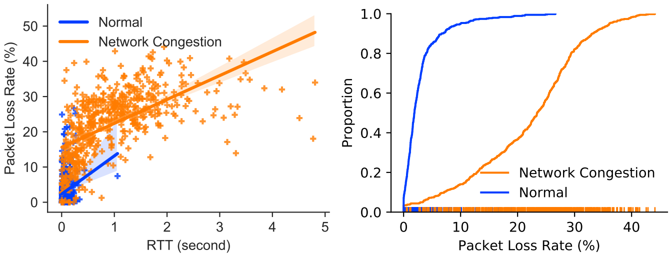
D. Impact of Packet Loss

We next analyze how packet loss influences the performance of PacketPrint. We focus on the following two major factors that cause packet loss in a practical wireless AF attack.

- **Sniffer Packet Loss.** Since the WiFi sniffer captures wireless traffic over the air, it may miss some 802.11 wireless frames between mobile devices and wireless AP due to signal attenuation, signal refraction, and multipathing [54]. As such, the WiFi sniffer only obtains incomplete wireless traffic. We refer to packet loss in this situation as *sniffer packet loss* (SPL). For a traffic instance, we estimate its SPL rate as $(m_p - m_p^c)/m_p$, where m_p is the number of all TCP/IP packets captured on the device and m_p^c is the number of TCP/IP packets that we can correlate 802.11 wireless frames with them. Apps that generate a larger amount of traffic may suffer more severe SPL. We compute the SPL rate of an app as an average SPL rate for its traffic instances. The average SPL rate of all apps in Monkey500 dataset is 11.6%. To analyze the impact of SPL rate, we group apps into four SPL levels according to their SPL rate. SPL level k contains apps with a SPL rate between $25(k-1)$th percentile and $25k$th percentile. Average SPL rates for SPL level 1, SPL level 2, SPL level 3, and SPL level 4 are 3.8%, 6.6%, 11.0%, and 22.4% respectively. Fig. 5(a) shows the experimental results, where we also consider the open-world setting. SPL undermines the recall of PacketPrint but does not significantly reduce its precision. Packet loss decreases the recall of PacketPrint because it decomposes structural patterns of packet arrivals. Without sufficient information extracted from structural patterns, PacketPrint generates more false negatives. For baseline methods, their precision is relatively stable but their recall also decreases as SPL rate increases. An interesting observation is a larger value of τ results in a slower drop in recall. We conjecture the reason is features extracted from larger sliding windows tend to be more statistically stable and more robust against SPL.

- **Congestion-Related Packet Loss.** Another important factor that causes packet loss is network congestion. It happens if network nodes or links are carrying more packets than they can handle, and the typical consequences include queueing delay and packet loss. We refer to packet loss in this situation as *congestion-related packet loss* (CPL). To simulate network congestion, we modify wireless AP's configuration to set the maximum inbound/outbound network speed of mobile devices to be 10KB/s. Fig. 6(a) presents the joint distribution of app traffic's RTT and CPL rate with/without network congestion. These two metrics are strongly positively correlated with each other. We elaborate how we estimate them in § B-C. Fig. 6(b) shows an ECDF plot of CRL rate. The average CRL rate dramatically increases from 3.0% to 21.8% when network congestion occurs. Fig. 7 reports the experimental results. Network congestion results in a negligible precision decline for PacketPrint but a more substantial precision decline for baseline methods. The impact of network congestion on the recall is more significant than that on the precision. The average recall of PacketPrint drops from 0.891 to 0.631. Nevertheless, PacketPrint still achieves a higher recall than all baseline methods and its advantage even appears to be larger. For example, compared with the baseline method that achieves the highest F1-score in the open-world setting, PacketPrint improves the recall with ~ 0.08 without network congestion, while it improves the recall with ~ 0.33 when network congestion occurs.

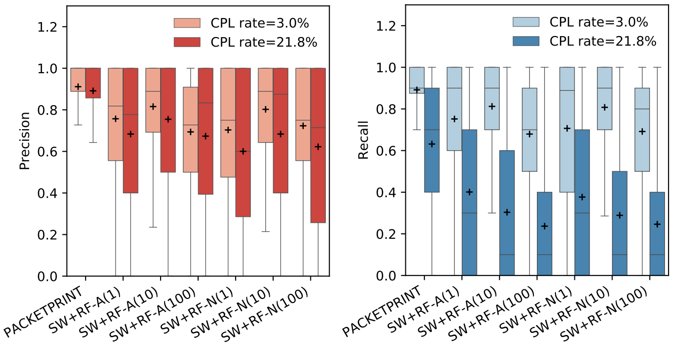
Comparing Fig. 7(b) to Fig. 5(b), we can see that



(a) Joint distribution of app traffic's RTT and CPL rate. Each scatter represents an app.

(b) ECDF of CPL rate.

Fig. 6: Comparing app traffic with/without network congestion.



(a) Precision.

(b) Recall.

Fig. 7: The impact of congestion-related packet loss (CPL).

PacketPrint seems to be more sensitive to CPL than SPL since PacketPrint achieves the recall 0.818 when the average SPL rate is 22.4% (SPL level 4) but achieves the recall 0.631 when the average CPL rate is 21.8%. The underlying reasons are two-fold. First, given a packet loss rate, CPL results in more lost packets than SPL because packet retransmission inflates the total packet number. For example, assume that the original number of packets is m if no packet is lost. 50% SPL rate corresponds to $m/2$ lost packets, while 50% CPL corresponds to m lost packets. Second, in the CPL experiment, PacketPrint is trained over app traffic with 3.0% CPL rate but is tested over app traffic with 21.8% CPL rate. Huge difference in CPL rate leads to a considerable concept drift between training data and testing data, thereby degrading PacketPrint's performance.

> **Answer to RQ2:** Sniffer packet loss (SPL) and congestion-related packet loss (CPL) will reduce PacketPrint's recall but have little impact on its precision. PacketPrint is more sensitive to CPL than SPL. Nonetheless, it still outperforms all baseline methods for both SPL and CPL, and even shows a more significant advantage in face of CPL.

E. *Impact of Noise Packets due to App Multiplexing*

In practice, mobile users may simultaneously use different apps. We refer to this scenario as app multiplexing. It will introduce a large number of noise packets that potentially undermine recognition accuracy. We next evaluate how PacketPrint is resilient to noise packets due to app multiplexing.

- **Experimental Setup.** Here we still consider the open-world setting. To simulate simultaneous use of different apps, we first synthesize unsegmented traffic $\mathscr{T}_a$ just like that in § VI-C. Next, we choose apps not involved in both the training and testing dataset to synthesize another part of unsegmented traffic $\mathscr{T}_b$ with their traffic instances. Finally, we superimpose the latter traffic into the former one to generate composite traffic $\mathscr{T}_{ab}$ in which packets generated by different apps are strongly mixed together. To evaluate PacketPrint's noise tolerance for different noise packet ratios, we set different average time intervals between traffic instances in $\mathscr{T}_b$. Specifically, we simulate the use of different apps as the Poisson process, which is commonly used to model human behaviors [55]–[57]. The time interval between traffic instances is exponentially distributed with a varying mean μ_b. A smaller value of μ_b results in more noise packets. We set $\mu_b = 1, 10, 50, 100, 500, 1000$ seconds to induce different noise packet ratios.

- **Result.** Fig. 8 reports the experimental results. PacketPrint exhibits strong robustness against noise packets. For example, it still achieves an average F1-score 0.826 when the noise packet ratio is 0.823. As opposed to PacketPrint, all baseline methods suffer from substantial performance decline with the increase of noise packet ratio, indicating they are not resilient to app multiplexing. A closer look reveals the recall of baseline methods decreases drastically as τ increases, which implies baseline methods with a larger sliding window size tend to be more susceptible to noise packets caused by app multiplexing. For example, the F1-score of SW+RF-A(100) features drops from 0.626 to 0.330 (~ 0.30 decline). As for PacketPrint, we only observe a F1-score decline from 0.884 to 0.795 (~ 0.09 decline) for the largest noise packet ratio. We believe it is attributed to the noise-tolerant property of label-aware feature mapping.

> **Answer to RQ3:** PacketPrint works well when faced with app multiplexing and exhibits significantly stronger noise tolerance than baseline methods. It achieves an average F1-score 0.826 when the noise packet ratio is 0.823.

F. *Cross-Dataset Recognition*

In this experiment, we analyze PacketPrint's transferability across different datasets by evaluating to what extent PacketPrint trained on app traffic triggered by automatic test tools, e.g., Monkey, can recognize human-generated app traffic.

- **Experimental Setup.** To evaluate PacketPrint's transferability, we consider the apps of interest as those belonging to both *Monkey500* dataset and *Human100* dataset. Without loss of generality, assume that $\mathbb{A}$ is an app of interest. We construct $\mathbb{A}$'s training dataset using traffic instances from *Monkey500* dataset. Similar to the setting in § VI-C, we randomly choose another 40 apps from *Monkey500* dataset and involve traffic instances in their training subset as negative samples of $\mathbb{A}$'s training dataset. Two testing datasets are constructed for different transfer settings. In the "M → M" setting, PacketPrint is trained over Monkey-generated dataset and tested over the same dataset. That is, the testing dataset of $\mathbb{A}$ is comprised of traffic instances of $\mathbb{A}$ and other randomly chosen apps from *Monkey500* dataset. In the "M → H" setting, PacketPrint is trained over Monkey-generated dataset and tested over human-generated dataset. Specifically, we construct the testing dataset

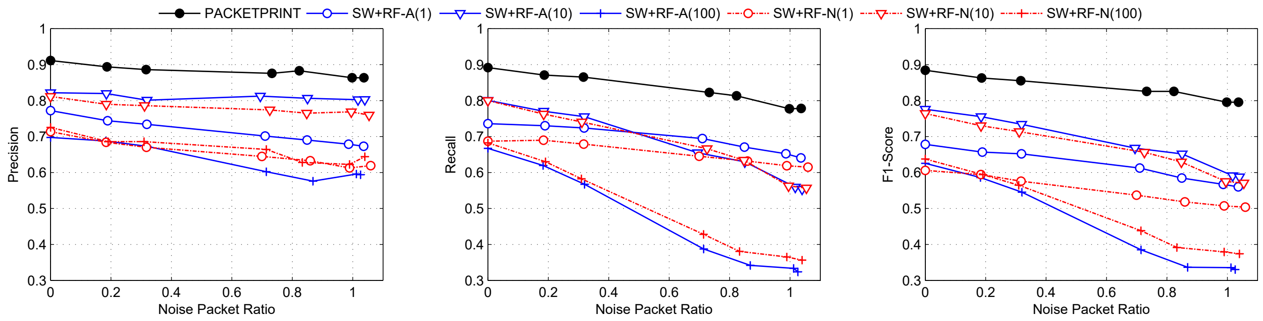


Fig. 8: The impact of noise packets due to app multiplexing.

TABLE III: Evaluating PacketPrint in cross-dataset app traffic recognition (mean±standard deviation).

Transfer Setting	Metric	SW+RF (AppScanner features)			SW+RF (NetScope features)			PACKETPRINT (Our method)
		$\tau = 1$	$\tau = 10$	$\tau = 100$	$\tau = 1$	$\tau = 10$	$\tau = 100$	
M → M	Precision	0.778 ± 0.243	0.840 ± 0.195	0.672 ± 0.267	0.718 ± 0.248	0.854 ± 0.191	0.706 ± 0.257	$\mathbf{0.901 \pm 0.134}$
	Recall	0.749 ± 0.312	0.810 ± 0.244	0.647 ± 0.282	0.733 ± 0.295	0.806 ± 0.256	0.679 ± 0.273	$\mathbf{0.915 \pm 0.086}$
	F1-score	0.687 ± 0.249	0.798 ± 0.197	0.613 ± 0.240	0.649 ± 0.209	0.794 ± 0.207	0.631 ± 0.233	$\mathbf{0.899 \pm 0.084}$
M → H	Precision	0.696 ± 0.317	0.789 ± 0.292	0.625 ± 0.368	0.585 ± 0.340	0.726 ± 0.349	0.608 ± 0.391	$\mathbf{0.910 \pm 0.168}$
	Recall	0.668 ± 0.395	0.546 ± 0.377	0.432 ± 0.372	0.614 ± 0.402	0.424 ± 0.380	0.370 ± 0.364	$\mathbf{0.850 \pm 0.225}$
	F1-score	0.553 ± 0.341	0.540 ± 0.334	0.386 ± 0.322	0.496 ± 0.331	0.425 ± 0.348	0.332 ± 0.310	$\mathbf{0.854 \pm 0.185}$

of $\mathbb{A}$ by involving all 5 traffic instances of $\mathbb{A}$ and another 40 randomly chosen apps from *Human100* dataset. In addition, we still consider the open-world setting in this experiment.

- **Result.** Table III reports the experimental results. PacketPrint shows strong transferability across datasets generated in different ways. More specifically, in the M → H setting, PacketPrint trained with Monkey-generated traffic accurately recognizes true human-generated traffic with the average F1-score 0.854, which is slightly lower than that in the M → M setting. Contrarily, baseline methods suffer from substantial performance degradation when conducting cross-dataset recognition. For example, SW+RF with AppScanner features ($\tau = 10$) has F1-score decline from 0.798 to 0.540, while SW+RF with NetScope features ($\tau = 10$) has F1-score decline from 0.794 to 0.425 Their performance degradation is mainly caused by the drop of recall. In other words, baseline methods generate more false negatives in the M → H setting.

The underlying reasons are two-fold. First, Monkey test can hardly cover all UI operations that human may trigger in the wild and thus true human-generated traffic may contain new traffic patterns that are absent in the Monkey-generated traffic. These new traffic patterns may become noises and change traffic features if they are mixed with known traffic patterns in the same slide window. Second, UI operations triggered by humans are contextually correlated while those for Monkey tend to be contextually independent because the behavior of Monkey is highly random. Such a difference potentially leads to different operation orders. Baseline methods are susceptible to the above two factors because features used by them are noise-sensitive and order-sensitive. As opposed to these methods, PacketPrint is resilient to the above two factors. First, it is noise-tolerant because of label-aware feature mapping, thereby immune to new traffic patterns. Second, it takes advantage of hierarchical bag-of-words model to extract order-independent features.

Since manually generating app traffic for the construction of training dataset is substantially labor-intensive, good transferability significantly facilitates PacketPrint's practicality due to a low cost for training data collection.

Answer to RQ4: PacketPrint trained on app traffic driven by automatic test tools can accurately recognize human-generated app traffic with the average F1-score 0.854.

G. In-App User Action Recognition

The above experiments focus on recognizing user activities associated with the app of interest without distinguishing specific in-app user actions. Here we evaluate whether PacketPrint can recognize fine-grained in-app user actions.

- **Experimental Setup.** Our evaluation is based on *IUA* dataset. Recall that PacketPrint recognizes in-app user actions associated with $\mathbb{A}$ from its relevant segments, i.e., segments of encrypted traffic that have been recognized to be generated by $\mathbb{A}$ (see § II-B). It implies that in-app user action recognition is essentially conducted in the closed-world setting. Therefore, we treat in-app user action recognition for different apps as independent tasks. For each in-app user action, we construct its training dataset and testing dataset by involving traffic instances of all in-app user actions belonging to the same app. Specifically, 4 out of 5 traffic instances are randomly chosen for model training, while the remaining one traffic instance is used for testing. To meet statistical soundness, for each in-app user action, we run training dataset construction, model training, and testing 100 times and find the mean to report the final performance of the attack.

- **Result.** Table IV reports the results for all in-app user actions considered in this experiment. PacketPrint recognizes in-app actions with an average F1-score 0.959. It can differentiate not only user actions that exhibit considerable difference in terms of traffic patterns, such as browsing video and watching video using Netflix, but also more subtle user actions. For

TABLE IV: Identifying in-app user actions.

App	In-app user action	Precision	Recall	F1-score
Youtube	Browse video	1.000	1.000	1.000
	Watch video	0.930	1.000	0.953
Netflix	Browse video	0.995	0.960	0.957
	Watch video	0.975	1.000	0.987
Bilibili	Browse video	1.000	0.990	0.990
	Watch video	0.960	0.980	0.953
Wechat	Share location	1.000	1.000	1.000
	Voice call	0.990	0.980	0.973
	Video call	1.000	1.000	1.000
	Browse moment	1.000	1.000	1.000
WhatsApp	Voice call	0.935	1.000	0.957
	Video call	0.980	1.000	0.987
	Send file	1.000	0.850	0.850
Line	Voice call	0.980	1.000	0.987
	Video call	1.000	1.000	1.000
	Send file	1.000	1.000	1.000
	Browse news	1.000	1.000	1.000
Spotify	Browse music	1.000	1.000	1.000
	Play music	0.920	0.900	0.847
Youtube Music	Browse music	0.955	0.980	0.950
	Play music	0.850	1.000	0.900
Musi	Browse music	0.855	0.930	0.853
	Play music	0.880	0.830	0.777
Taobao	Search items	1.000	1.000	1.000
	Purchase products	1.000	1.000	1.000
Google Play	Browse app	1.000	1.000	1.000
	Download app	0.970	1.000	0.980
Tik Tok	Browse video	1.000	0.960	0.960
	Watch video	1.000	1.000	1.000
Instagram	Browse posts	1.000	0.960	0.960
	Post	1.000	0.920	0.920
Average	**	0.973	0.975	0.959

example, PacketPrint is able to accurately distinguish between searching items and purchasing products using Taobao, a famous Chinese e-commerce app. Compared with the previous experiments, PacketPrint achieves higher accuracy for in-app user action recognition. The underlying reasons are two-fold. First, in-app user action recognition is conducted in the closed-world setting and less challenging than other tasks in the open-world setting. Second, in-app user actions listed in Table IV have relatively stable and distinguishable traffic patterns. PacketPrint is capable of capturing traffic pattern differences and recognizing different in-app user actions.

• **Long-Running Behaviors.** We further evaluate whether PacketPrint is able to distinguish long-running behaviors of the same type. Specifically, we consider four representative long-running user actions, including watching video with different video streaming apps, making a voice/video call with different IM apps, and playing music with different music apps as listed in Table V. Compared to different in-app user actions with the same app, it is a more challenging to distinguish user actions of the same type for different apps because they typically exhibit very similar traffic patterns. In this experiment, we consider both "action only" and "app+action" settings. In the "action only" setting, PacketPrint directly recognizes and distinguishes user actions of the same type without first recognizing apps. In the "app+action" setting, PacketPrint first recognizes the app to locate all its traffic segments and then recognizes in-app user actions from these traffic segments. For each user action, we regard its traffic instances as positive samples and traffic instances generated by other user actions of the same type as negative samples. Table V reports the experimental results. We observe a precision decline for PacketPrint to distinguish user actions of the same type in the "action only" setting. For example, the precision of recognizing watching video with Netflix drops from 0.975 to 0.517. Fortunately, PacketPrint in the "app+action" setting effectively improves the precision because other in-app actions of the app will be jointly considered to reduce false positive cases. For example, PacketPrint improves the precision of recognizing watching video with Netflix from 0.517 to 0.900. We also observe that "app+action" setting improves precision at the cost of slight recall decline. The root cause is that "app+action" setting is essentially two-stage app traffic recognition, and thus the false negative cases of app recognition also cause false negatives for in-app user action recognition.

TABLE V: Distinguishing user actions of the same type.

Behavior	Setting	App	Precision	Recall	F1-score
Watch video	AO	Youtube	0.980	1.000	0.986
		Netflix	0.517	1.000	0.668
		Bilibili	0.782	1.000	0.853
	AA	Youtube	1.000	0.950	0.950
		Netflix	0.900	0.990	0.918
		Bilibili	0.995	0.920	0.917
Voice call	AO	Wechat	1.000	1.000	1.000
		WhatsApp	0.900	1.000	0.933
		Line	1.000	1.000	1.000
	AA	Wechat	1.000	0.980	0.980
		WhatsApp	0.985	1.000	0.990
		Line	0.995	0.930	0.927
Video call	AO	Wechat	0.985	1.000	0.990
		WhatsApp	0.930	1.000	0.953
		Line	0.980	1.000	0.987
	AA	Wechat	0.985	0.990	0.980
		WhatsApp	1.000	1.000	1.000
		Line	1.000	0.970	0.970
Play music	AO	Spotify	0.712	1.000	0.798
		Youtube Music	0.745	1.000	0.827
		Musi	0.748	1.000	0.832
	AA	Spotify	0.980	0.890	0.877
		Youtube Music	0.995	0.990	0.987
		Musi	1.000	0.970	0.970

AO is short for "action only" and AA is short for "app+action".

Answer to RQ5: PacketPrint can accurately recognize in-app user actions. The average F1-score is up to 0.959.

VII. Discussion

Experimental results in § VI show that PacketPrint poses a serious threat to mobile users' online privacy. In this section, we discuss possible mitigation solutions. AF attacks aim to infer user activities associated with apps by leveraging features extracted from i) destination information, ii) packet timing, iii) packet direction, and iv) packet size, without accessing packet payload plaintext. To protect apps from AF attacks, the defender may obfuscate app traffic through feature perturbations.

• **Hiding Real Destination.** To hide app servers' destination information (e.g., IP address and domain name), mobile users can access Internet through VPN or encrypted proxies, e.g., ShadowSocks [58]. However, such a defense is only effective for AF attacks that use destination information [6], [11] and will not work for PacketPrint.

• **MAC Address Randomization.** Starting in iOS 8 and Android 8.0, mobile devices enable MAC address randomization as an optional security mechanism to enhance privacy protection. However, the frequency of MAC address randomization is generally much lower than app dynamics. For example, iPhones choose a new randomized address for each wireless network and randomize it every 24 hours. Therefore, MAC address randomization cannot prevent the adversary from recognizing user activities associated with the app of interest from wireless encrypted traffic but could cause some difficulties in correlating short-term user activities across multiple days to infer long-term user habits. Besides, some methods have been proposed to circumvent this security mechanism. For example, Matte et. al propose to use the fingerprinting of the probe request from a certain device as persistent identifier [59] and Martin et. al propose to recover the true MAC addresses with a fine-grained inferred address mapping [60].

• **Perturbing Timings.** It changes packet-timing-based features by introducing inter-packet delay [61], [62]. To avoid interfering with normal interactive logic in the communication between the app client and server, timing perturbation will not change packet order, resulting in little impact on sequential patterns and structural patterns of packet arrivals. Therefore, we expect PacketPrint is robust against timing perturbation.

• **Dummy Packet Injection.** The strategy of injecting dummy packets has been proven to be effective against WF attacks [62], [63]. Essentially, dummy packets can be viewed as noises that perturb packet-direction-based features. As demonstrated in § VI-E, PacketPrint is resilient to a moderate number of noise packets.

• **Packet Padding.** It can perturb packet-size-based features. Since PacketPrint takes advantage of features extracted from packet sizes, it is naturally susceptible to packet padding. In essence, the strategy of packet padding undermines PacketPrint's recognition accuracy by reducing the resolution of packet-size-based features. In an extreme case where all packets are padded to MTU, PacketPrint can only observe two kinds of packets in terms of packet size, i.e., inbound packets and outbound packets. Packet padding can be implemented at application layer. For example, HTTPOS implemented application-layer padding by altering the size of outgoing packets on the HTTP layer [64]. Despite the effectiveness, application-layer padding often needs extra efforts by app developers, thereby limiting its scalability. In fact, we didn't observe application-layer padding had been widely applied in popular apps in our dataset. Additionally, if only a few apps choose to apply packet padding, consistent packet sizes may become a strong indicator for the presence of these apps, instead of hiding them. Packet padding can also be implemented based on VPN or encrypted proxy. For example, IMProxy implemented packet padding by leveraging a local proxy and a remote proxy for traffic relay [61]. Although both VPN-based padding and proxy-based padding are globally effective for all apps on the mobile device, they will introduce considerable network delay and degrade user experience. Moreover, untrustworthy third-party proxies may invade user privacy. Another side effect of packet padding is all the above implementations will introduce extra network overhead and reduce communication efficiency.

VIII. Related Work

Traffic fingerprinting (TF) constitutes an important branch of traffic analysis techniques. Network administrators employ TF for Internet demographics, security monitoring, and censorship [65]–[69], whereas adversaries take advantage of TF to eavesdrop sensitive information [9], [17], [23], [24].

A. App Fingerprinting

Our work falls into this category. By analyzing app traffic, some existing works [6]–[8], [14] aim to identify apps from encrypted app traffic. For example, Van Ede et al. proposed a semi-supervised approach to fingerprint mobile apps based on destination-related features [6]. Taylor et al. proposed a modular framework, dubbed AppScanner, for automatic fingerprinting and real-time identification of Android apps from their encrypted network traffic [7], [8]. Aceto et. al proposed a multimodal deep learning framework, named by MIMETIC, to classify mobile encrypted traffic and identify apps [14].

Another line of works [9]–[13], [70] intend to identify fine-grained user behaviors associated with apps. For example, Conti et al. proposed a framework to infer which particular actions the user executed on some app by using information available in TCP/IP packets (like IP addresses and ports), together with other information like the size, direction, and timing [11]. Saltaformaggio et al. presented NetScope, a systematic method to perform robust inference of users' semantic activities [9]; Liu et al. developed an iterative analyzer for classifying encrypted mobile traffic in a real-time way. They segmented app traffic based on a recursive time continuity constrained KMeans clustering (rCKC) algorithm and classified the segmented traffic to identify in-app user activity [10].

Existing AF attacks cannot tackle the challenges listed in § I. AF attacks that rely on destination information, e.g., [6], [11], are unable to handle encrypted wireless traffic due to hidden destination. Second, the vast majority of existing AF attacks, e.g., [9]–[12], [14], [51], work under the closed-world setting, where apps in the testing stage should also be present in the training stage. In other words, they are unable to handle open-world app fingerprinting by design. Third, there is no AF attack can handle app multiplexing for encrypted wireless traffic. For example, existing works, e.g., [6], [10], [12], either explicitly or implicitly, assume that apps are executed one at a time. At the first glance, AF attacks that extract network flows as traffic samples can solve app multiplexing because each network flow is exclusively generated by one app. Unfortunately, network flows cannot be identified and extracted since transport-layer endpoints are invisible in 802.11 wireless frames. Consequently, AF attacks, e.g., [7]–[9], [14], which need to extract network flows, fail to solve our problem.

B. Website Fingerprinting

Website fingerprinting (WF) attacks aim to infer the websites being visited by users via encrypted proxies or anonymity networks [71]. Existing WF attacks, e.g., [21]–[28], [72], [73], can be roughly grouped into feature-engineering-based WF and deep-learning-based WF. WF attacks in the former category [21], [23], [24], [26], [27] extract features based on domain knowledge and train classifiers to identify website. WF attacks in the latter category [25], [28], [74] use deep learning models to achieve automatic feature extraction. Since deep-learning-based WF attacks often need enormous training data, some improvements leverage metric learning [22] or adversarial domain adaption [75] to achieve few-shot learning. Despite higher accuracy, deep-learning-based WF attacks have been proven to be susceptible to adversarial perturbations [62]. Existing WF attacks usually assumed that network traffic corresponding to different websites can be reasonably separated by obvious time gaps between packets [63]. Unfortunately, packets generated by different apps are often strongly mixed together. Therefore, models of WF attacks cannot be directly applied to our problem.

IX. Conclusion

We proposed PacketPrint, a novel AF attack to recognize user activities associated with apps of interest in the open-world setting by addressing four challenges, including hidden destination, invisible boundary, app multiplexing, and open-world recognition. The extensive experimental results show that PacketPrint achieves the average F1-score 0.884 for open-world app recognition and the average F1-score 0.959 for in-app user action recognition. When faced with noise packets caused by the simultaneous use of different apps, PacketPrint still achieves a reasonably high F1-score 0.826 when the noise packet ratio is 0.823. Moreover, PacketPrint trained over app traffic driven by automatic test tools can accurately recognize human-generated app traffic with the average F1-score 0.854.

Acknowledgment

We sincerely thank the anonymous reviewers for their constructive comments. This work is partly supported by Hong Kong ITF Project (No. GHP/052/19SZ), National Science Foundation (No. 1951729, 1953813, and 1953893), National Natural Science Foundation of China (No. 61972313), and Cyrus Tang Foundation as an XJTU Tang Scholar.

References

[1] M. Yea-Ji and K. Hyun-Bin, "O2O apps become part of daily life," https://www.korea.net/NewsFocus/Society/view?articleId=162968, 2021.

[2] C.-D. Chen, C.-K. Huang, M.-J. Chen, and E. Ku, "User's adoption of mobile o2o applications: Perspectives of the uses and gratifications paradigm and service dominant logic," 2015.

[3] Arthur and Charles, "Naked celebrity hack: security experts focus on icloud backup theory," http://www.theguardian.com/technology/2014/sep/01/naked-celebrity-hack-icloud-backup-jennifer-lawrence, 2020.

[4] T. McCoy, "4chan: The 'shock post' site that hosted the private jennifer lawrence photos," http://shorturl.at/biHLN, 2020.

[5] Zetter and Kim, "Sony got hacked hard: What we know and don't know so far," http://www.wired.com/2014/12/sony-hack-what-we-know/, 2020.

[6] T. van Ede, R. Bortolameotti, A. Continella, J. Ren, D. J. Dubois, M. Lindorfer, D. Choffnes, M. van Steen, and A. Peter, "Flowprint: Semi-supervised mobile-app fingerprinting on encrypted network traffic," in *Network and Distributed System Security Symposium, NDSS 2020*. Internet Society, 2020.

[7] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 439–454.

[8] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Robust smartphone app identification via encrypted network traffic analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 1, pp. 63–78, 2017.

[9] B. Saltaformaggio, H. Choi, K. Johnson, Y. Kwon, Q. Zhang, X. Zhang, D. Xu, and J. Qian, "Eavesdropping on fine-grained user activities within smartphone apps over encrypted network traffic," in *10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16)*, 2016.

[10] J. Liu, Y. Fu, J. Ming, Y. Ren, L. Sun, and H. Xiong, "Effective and real-time in-app activity analysis in encrypted internet traffic streams," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 335–344.

[11] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, "Can't you hear me knocking: Identification of user actions on android apps via traffic analysis," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015, pp. 297–304.

[12] D. Li, W. Li, X. Wang, C.-T. Nguyen, and S. Lu, "Activetracker: Uncovering the trajectory of app activities over encrypted internet traffic streams," in *2019 16th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE, 2019, pp. 1–9.

[13] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, "Analyzing android encrypted network traffic to identify user actions," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 1, pp. 114–125, 2015.

[14] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapè, "Mimetic: Mobile encrypted traffic classification using multimodal deep learning," *Computer Networks*, vol. 165, p. 106944, 2019.

[15] Samsung, "Smartthings," https://www.smartthings.com, 2021.

[16] Apple, "Homekit," https://www.apple.com/ios/home/, 2021.

[17] A. Acar, H. Fereidooni, T. Abera, A. K. Sikder, M. Miettinen, H. Aksu, M. Conti, A.-R. Sadeghi, and S. Uluagac, "Peek-a-boo: I see your smart home activities, even encrypted!" in *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020, pp. 207–218.

[18] W. E. Forum, "The COVID-19 pandemic has changed education forever," https://www.weforum.org/agenda/2020/04/coronavirus-education-global-covid19-online-digital-learning/, 2021.

[19] D. I. Jason Abbruzzese and S. Click, "The coronavirus pandemic drove life online," https://www.nbcnews.com/tech/internet/coronavirus-pandemic-drove-life-online-it-may-never-return-n1169956, 2021.

[20] J. Li, X. Ma, L. Guodong, X. Luo, J. Zhang, W. Li, and X. Guan, "Can we learn what people are doing from raw dns queries?" in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 2240–2248.

[21] A. Panchenko, F. Lanze, J. Pennekamp, T. Engel, A. Zinnen, M. Henze, and K. Wehrle, "Website fingerprinting at internet scale." in *NDSS*, 2016.

[22] P. Sirinam, N. Mathews, M. S. Rahman, and M. Wright, "Triplet fingerprinting: More practical and portable website fingerprinting with n-shot learning," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1131–1148.

[23] T. Wang and I. Goldberg, "On realistically attacking tor with website fingerprinting," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 4, pp. 21–36, 2016.

[24] T. Wang, "High precision open-world website fingerprinting," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 152–167.

[25] V. Rimmer, D. Preuveneers, M. Juarez, T. Van Goethem, and W. Joosen, "Automated website fingerprinting through deep learning," *arXiv preprint arXiv:1708.06376*, 2017.

[26] J. Hayes and G. Danezis, "k-fingerprinting: A robust scalable website fingerprinting technique," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 1187–1203.

[27] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 639–656.

[28] P. Sirinam, M. Imani, M. Juarez, and M. Wright, "Deep fingerprinting: Undermining website fingerprinting defenses with deep learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1928–1943.

[29] Google, "Multi-window support," https://developer.android.com/guide/topics/ui/multi-window, 2021.

[30] Statista, "Number of available apps in apple app store," https://www.statista.com/statistics/779768/number-of-available-apps-in-the-apple-app-store-quarter/, 2020.

[31] Statista, "Number of available apps in googleplay store," https://www.statista.com/statistics/289418/number-of-available-apps-in-the-google-play-store-quarter/, 2020.

[32] J. Jonsson, "On the security of ctr+ cbc-mac," in *International Workshop on Selected Areas in Cryptography*. Springer, 2002, pp. 76–93.

[33] A. D. Potorac and D. Balan, "The impact of security overheads on 802.11 wlan throughput," *Journal of Computer Science and Control Systems*, vol. 2, no. 1, pp. 47–52, 2009.

[34] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. Savelsbergh, and P. H. Vance, "Branch-and-price: Column generation for solving huge integer programs," *Operations research*, vol. 46, no. 3, pp. 316–329, 1998.

[35] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proc. of ACM SIGKDD*, 2016.

[36] "Omnipeek," https://www.liveaction.com/products/omnipeek-network-protocol-analyzer/, 2022.

[37] "Monkey," https://developer.android.com/studio/test/monkey, 2020.

[38] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing- and touch-sensitive record and replay for android," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 72–81.

[39] "Tcpdump," https://www.tcpdump.org, 2021.

[40] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman, "Identifying diverse usage behaviors of smartphone apps," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, 2011, pp. 329–344.

[41] Q. Wang, A. Yahyavi, B. Kemme, and W. He, "I know what you did on your smartphone: Inferring app usage over encrypted data traffic," in *2015 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2015, pp. 433–441.

[42] E. Grolman, A. Finkelshtein, R. Puzis, A. Shabtai, G. Celniker, Z. Katzir, and L. Rosenfeld, "Transfer learning for user action identication in mobile apps via encrypted traffic analysis," *IEEE Intelligent Systems*, vol. 33, no. 2, pp. 40–53, 2018.

[43] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "Traffic classification of mobile apps through multi-classification," in *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE, 2017, pp. 1–6.

[44] M. Tian, P. Chang, Y. Sang, Y. Zhang, and S. Li, "Mobile application identification over https traffic based on multi-view features," in *2019 26th International Conference on Telecommunications (ICT)*. IEEE, 2019, pp. 73–79.

[45] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "Mobile encrypted traffic classification using deep learning," in *2018 Network Traffic Measurement and Analysis Conference (TMA)*. IEEE, 2018, pp. 1–8.

[46] H. D. Trinh, A. F. Gambin, L. Giupponi, and P. Dini, "Classification of mobile services and apps through physical channel fingerprinting: a deep learning approach," *arXiv preprint arXiv:1910.11617*, 2019.

[47] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "Multi-classification approaches for classifying mobile app traffic," *Journal of Network and Computer Applications*, vol. 103, pp. 131–145, 2018.

[48] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, "Networkprofiler: Towards automatic fingerprinting of android apps," in *2013 Proceedings IEEE INFOCOM*. IEEE, 2013, pp. 809–817.

[49] Y. Chen, W. You, Y. Lee, K. Chen, X. Wang, and W. Zou, "Mass discovery of android traffic imprints through instantiated partial execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 815–828.

[50] C. Hou, J. Shi, C. Kang, Z. Cao, and X. Gang, "Classifying user activities in the encrypted wechat traffic," in *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2018, pp. 1–8.

[51] Y. Fu, H. Xiong, X. Lu, J. Yang, and C. Chen, "Service usage classification with encrypted internet traffic in mobile messaging apps," *IEEE Transactions on Mobile Computing*, vol. 15, no. 11, pp. 2851–2864, 2016.

[52] R. Bortolameotti, T. van Ede, M. Caselli, M. H. Everts, P. Hartel, R. Hofstede, W. Jonker, and A. Peter, "Decanter: Detection of anomalous outbound http traffic by passive application fingerprinting," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, 2017, pp. 373–386.

[53] R. Bortolameotti, T. Van Ede, A. Continella, T. Hupperich, M. H. Everts, R. Rafati, W. Jonker, P. Hartel, and A. Peter, "Headprint: detecting anomalous communications through header-based application fingerprinting," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1696–1705.

[54] C. A. G. Da Silva and C. M. Pedroso, "Mac-layer packet loss models for wi-fi networks: A survey," *IEEE Access*, vol. 7, pp. 180 512–180 531, 2019.

[55] J. Cho and H. Garcia-Molina, "Estimating frequency of change," *ACM Transactions on Internet Technology (TOIT)*, vol. 3, no. 3, pp. 256–290, 2003.

[56] C. C. Zou, D. Towsley, and W. Gong, "Email worm modeling and defense," in *Proceedings. 13th International Conference on Computer Communications and Networks (IEEE Cat. No. 04EX969)*. IEEE, 2004, pp. 409–414.

[57] K. C. Sia, J. Cho, and H.-K. Cho, "Efficient monitoring algorithm for fast news alerts," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 7, pp. 950–961, 2007.

[58] "ShadowSocks," https://shadowsocks.org/en/index.html, 2021.

[59] C. Matte, M. Cunche, F. Rousseau, and M. Vanhoef, "Defeating mac address randomization through timing attacks," in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2016, pp. 15–20.

[60] J. Martin, E. Rye, and R. Beverly, "Decomposition of mac address structure for granular device inference," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 78–88.

[61] A. Bahramali, A. Houmansadr, R. Soltani, D. Goeckel, and D. Towsley, "Practical traffic analysis attacks on secure messaging applications," in *NDSS*, 01 2020.

[62] M. Nasr, A. Bahramali, and A. Houmansadr, "Defeating dnn-based traffic analysis systems in real-time with blind adversarial perturbations," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[63] J. Gong and T. Wang, "Zero-delay lightweight defenses against website fingerprinting," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 717–734.

[64] X. Luo, P. Zhou, E. W. Chan, W. Lee, R. K. Chang, R. Perdisci *et al.*, "Httpos: Sealing information leaks with browser-side obfuscation of encrypted flows." in *NDSS*, vol. 11, 2011.

[65] X. Ma, J. Qu, J. Li, J. C. Lui, Z. Li, and X. Guan, "Pinpointing hidden iot devices via spatial-temporal traffic fingerprinting," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 894–903.

[66] T. Stöber, M. Frank, J. Schmitt, and I. Martinovic, "Who do you sync you are? smartphone fingerprinting via application behaviour," in *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, 2013, pp. 7–12.

[67] Y. Wan, K. Xu, G. Xue, and F. Wang, "Iotargos: A multi-layer security monitoring system for internet-of-things in smart homes," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 874–883.

[68] M. Wei, "Domain shadowing: Leveraging content delivery networks for robust blocking-resistant communications," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[69] X. Ma, J. Qu, J. Li, J. C. S. Lui, Z. Li, W. Liu, and X. Guan, "Inferring hidden iot devices and user interactions via spatial-temporal traffic fingerprinting," *IEEE/ACM Transactions on Networking*, pp. 1–15, 2021.

[70] J. Li, H. Zhou, S. Wu, X. Luo, T. Wang, X. Zhan, and X. Ma, "FOAP: Fine-Grained Open-World android app fingerprinting," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston,

MA: USENIX Association, Aug. 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/li-jianfeng

[71] "Tor," https://www.torproject.org/, 2021.

[72] X. Ma, M. Shi, B. An, J. Li, D. X. Luo, J. Zhang, and X. Guan, "Context-aware website fingerprinting over encrypted proxies," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[73] X. Xiao, W. Xiao, R. Li, X. Luo, H. Zheng, and S. Xia, "Ebsnn: Extended byte segment neuralnetwork for network traffic classification," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[74] S. Bhat, D. Lu, A. Kwon, and S. Devadas, "Var-cnn and dynaflow: Improved attacks and defenses for website fingerprinting," *CoRR*, vol. abs/1802.10215, 2018. [Online]. Available: http://arxiv.org/abs/1802.10215

[75] C. Wang, J. Dani, X. Li, X. Jia, and B. Wang, "Adaptive fingerprinting: Website fingerprinting over few encrypted traffic," in *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, 2021, pp. 149–160.

APPENDICES

APPENDIX A
DETAILS OF MODELS AND ALGORITHMS

A. Recursive Model Training

To speed up the training process, Eqn. (2) is approximated by its second-order Taylor expansion

$$\hat{\mathcal{L}}_j^k = \sum_{i=1}^{m} g_i(0) + g_i^{(1)} f_j^k(\mathbf{x}_i^k) + \frac{1}{2} g_i^{(2)} f_j^k(\mathbf{x}_i^k)^2 + \Theta(f_i^k), \tag{11}$$

where $g_i(z) = L\left(y_i, \sum_{r=0}^{k-1} F_r(\mathbf{x}_i^r) + \sum_{s=0}^{j-1} f_s^k(\mathbf{x}_i^k) + z\right)$ is the loss function of p_i,

$$g_i^{(1)} = -y_i + \frac{\exp\left(\sum_{r=0}^{k-1} F_r(\mathbf{x}_i^r) + \sum_{s=0}^{j-1} f_s^k(\mathbf{x}_i^k)\right)}{\exp(\sum_{r=0}^{k-1} F_r(\mathbf{x}_i^r) + \sum_{s=0}^{j-1} f_s^k(\mathbf{x}_i^k)) + 1} \tag{12}$$

is the first-order derivative of $g(z)$ at $z = 0$, and

$$g_i^{(2)} = \frac{\exp\left(\sum_{r=0}^{k-1} F_r(\mathbf{x}_i^r) + \sum_{s=0}^{j-1} f_s^k(\mathbf{x}_i^k)\right)}{\left[\exp\left(\sum_{r=0}^{k-1} F_r(\mathbf{x}_i^r) + \sum_{s=0}^{j-1} f_s^k(\mathbf{x}_i^k)\right) + 1\right]^2} \tag{13}$$

is the second-order derivative of $g(z)$ at $z = 0$. Recalling that f_j^k is approximated by a decision tree, we rewrite Eqn. (11) as

$$\hat{\mathcal{L}}_j^k = \sum_{t=1}^{n_j^k} \sum_{\mathbf{x}_i^k \in \mathcal{C}_j} g_i^{(1)} v_t + \frac{1}{2}\left(\lambda + \sum_{i \in \mathcal{C}_t} g^{(2)}\right) w_t^2 + g_i(0) + \gamma n_j^k. \tag{14}$$

To minimize $\hat{\mathcal{L}}_t^s$, the optimal value associated with the leaf $\mathcal{C}_t$ can be independently computed by

$$v_t^* = -\frac{\sum_{\mathbf{x}_i^k \in \mathcal{C}_j} g_i^{(1)}}{\lambda + \sum_{\mathbf{x}_i^k \in \mathcal{C}_t} g_i^{(2)}}. \tag{15}$$

Given v_j^*, the total residual loss corresponding to packets falling into $\mathcal{C}_j$ is computed by

$$H(\mathcal{C}_j) = \sum_{p_i \in \mathcal{C}_j} g_i(0) - \frac{(\sum_{i \in \mathcal{C}_j} g_i^{(1)})^2}{2\left(\lambda + \sum_{\mathbf{x}_i^k \in \mathcal{C}_t} g_i^{(2)}\right)} + \gamma. \tag{16}$$

To construct the f_j^k that minimizes the loss in Eqn. (14), we expand the decision tree by recursively splitting leaf nodes on it. Specifically, given a leaf $\mathcal{C}$ and a splitting point u, $\mathcal{C}$ can be split into two children $\mathcal{C}_u^L$ and $\mathcal{C}_u^R$, leading to a loss drop $\Delta H(u, \mathcal{C}) = H(\mathcal{C}) - [H(\mathcal{C}_u^L) + H(\mathcal{C}_u^R)]$. The optimal splitting is expressed by $(\mathcal{C}^*, u^*) = \arg\max_{\mathcal{C},u} \Delta H(u, \mathcal{C})$ subject to $\Delta H(u, \mathcal{C}) > 0$. We expand the decision tree by recursively searching for $(\mathcal{C}^*, u^*)$ and splitting $\mathcal{C}^*$ at u^*.

B. Packet Size List Construction

We transform the problem of constructing packet size list to a combinational optimization problem in Eqn. (1) and propose Algorithm 1 to solve it.

Algorithm 1: Packet size list construction

```
Input: $\mathcal{S}_p$, $R_{\min}$, $M_{\min}^s$, $r_+(s_i)$ and $r_-(s_i)$ for $\forall s_i \in \mathcal{S}_p$;
Output: $\mathcal{S}_p^{\mathbb{A}}$;
$\mathcal{S}_p^{\mathbb{A}} \leftarrow \mathcal{S}_p$, $\mathcal{S}_p' \leftarrow \mathcal{S}_p$;
$r_{\min} \leftarrow \min\left\{\sum_{s_i \in \mathcal{S}_p} r_+(s_i), R_{\min}\right\}$;
while $|\mathcal{S}_p^{\mathbb{A}}| > M_{\min}^s \wedge |\mathcal{S}_p'| > 0$ do
    $s_{i*} \leftarrow \arg\max_{s_i \in \mathcal{S}_p'} \frac{r_-(s_i)}{r_+(s_i)}$;
    if $\sum_{s_i \in \mathcal{S}_p^{\mathbb{A}} \setminus \{s_{i*}\}} r_+(s_i) \geq r_{\min}$ then
        $\mathcal{S}_p^{\mathbb{A}} \leftarrow \mathcal{S}_p^{\mathbb{A}} \setminus \{s_{i*}\}$;
    end
    $\mathcal{S}_p' \leftarrow \mathcal{S}_p' \setminus \{s_{i*}\}$;
end
```

C. Recursive LFM Training

H-BoW model recursively maps structural patterns at a smaller time scale to a categorical variable at a larger time scale. Mapping functions are learned in a supervised manner as shown in Algorithm 2.

D. Greedy Feature Representation

The feature representation of H-BoW model is formulated as the optimization problem in Eqn. (7). We propose a greedy algorithm to solve it as shown in Algorithm 3.

APPENDIX B
SUPPLEMENTARY MATERIAL OF EVALUATION

A. Training Data Labeling

- **Correlating 802.11 wireless frames with TCP/IP packets.** We make use of TCP/IP packets as a bridge to label 802.11 wireless frames because TCP/IP packets contain Internet-layer and transport-layer endpoints, which are informative to indicate which app generates them. Specifically, we correlate each Data-Type (QoS data) 802.11 wireless frame captured by the WiFi sniffer with a TCP/IP packet captured on the device. To this end, we first extract the timestamp of each Data-Type 802.11 wireless frame and then derive the packet size of the TCP/IP packet encapsulated in it by subtracting the length of frame body by a protocol-dependent encryption overhead (see § III-B). By comparing these two kinds of information with TCP/IP packets captured on the device, we are able to correlate 802.11 wireless frames with TCP/IP packets.

- **Correlating TCP/IP packets with sockets.** We correlate packets with sockets (described as 4-tuples) according to Internet-layer and transport-layer endpoints, i.e., source/destination IP address and source/destination port.

Algorithm 2: Recursive LFM training

```
Input: $\mathbf{p} = (p_1, p_2, \ldots, p_m)$, $\{\mathcal{T}_t^2\}_{t=1}^{m_2}$, $\{\mathcal{T}_t^3\}_{t=1}^{m_3}$;
Output: $\mathcal{V}$, $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$;
$\mathcal{V} \leftarrow \varnothing$;
for $s = 1, 2, 3$ do
    if $s = 1$ then
        for $t = 1, 2, \ldots, m$ do
            $\mathbf{z}_t^1 \leftarrow \mathbf{x}_t^1$;
        end
    else
        for $t = 1, 2, \ldots, m_s$ do
            Compute $\mathbf{z}_t^s$ according to Eqn. (4);
        end
    end
    Construct the set $\mathscr{D}_{train}^s = \{(\mathbf{z}_t^s, y_t^s)\}_{t=1}^{m_s}$;
    Train a decision tree $\mathcal{M}_s$ over $\mathscr{D}_{train}^s$;
    Add positive leaves on $\mathcal{M}_s$ to the vocabulary: $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{R}_+^{s+1}$;
end
```

Algorithm 3: Greedy Feature Representation

```
Input: $\mathscr{D}_{train}^+$, $\mathscr{D}_{train}^-$, $\alpha$;
Output: $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_{|\mathcal{V}|}$;
$\mathbf{c}_j \leftarrow \mathbf{0}_{1 \times n_f}$ for $1 \le j \le |\mathcal{V}|$;
for $i = 1, 2, \ldots, |\mathcal{V}|$ do
    for $j = 1, 2, \ldots, |\mathcal{V}|$ do
        if $\|\mathbf{c}_j\|_1 > 0$ then
            continue;
        end
        for $k = 1, 2, \ldots, n_f$ do
            $\Delta\mathcal{L}(j, k) \leftarrow 0$;
            foreach $\tilde{\mathbf{s}}_t \in \mathscr{D}_{train}^+$ do
                $\mathbf{h}_t \leftarrow \sum_{l \in W(\tilde{\mathbf{s}}_t)} \mathbf{c}_l$;
                if $\mathbf{h}_t(k) = 0$ then
                    $\Delta\mathcal{L}(j, k) \leftarrow \Delta\mathcal{L}(j, k) + \alpha / |\mathscr{D}_{train}^+|$;
                end
            end
            foreach $\tilde{\mathbf{s}}_t \in \mathscr{D}_{train}^-$ do
                $\mathbf{h}_t \leftarrow \sum_{l \in W(\tilde{\mathbf{s}}_t)} \mathbf{c}_l$;
                if $\mathbf{h}_t(k) = 0$ then
                    $\Delta\mathcal{L}(j, k) \leftarrow \Delta\mathcal{L}(j, k) - 1 / |\mathscr{D}_{train}^-|$;
                end
            end
        end
    end
    $j^*, k^* \leftarrow \arg\max_{j,k} \Delta\mathcal{L}(j, k)$;
    if $\Delta\mathcal{L}(j^*, k^*) \le 0$ then
        break;
    else
        $\mathbf{c}_{j^*}(k^*) \leftarrow 1$;
    end
end
```

• **Correlating sockets with apps.** To correlate the socket with app, we first instrument the socket-related system functions (e.g., `sendto`, `write` in `libc.so`), to retrieve sockets identified by a unique 4-tuple. Next, in these functions, we internally invoke `getpid` to obtain the PID values, with which we can find the app's package name by accessing the `/proc/PID/cmdline` file. By doing so, we establish the correlation between each socket and the app that creates it. We output the above information to `logcat` in real time.

Combining the above three kinds of information, we are able to label each Data-Type frame to indicate what app generates it.

B. Hyperparameter Tuning

Since PacketPrint is an AF attack working in the open-world setting by design, we fine-tune hyperparameters based on grid searching to maximize the average F1-score in the open-world setting. Table VI summarizes the hyperparameter tuning process. Similarly, we fine-tune the recognition threshold for each baseline method to maximize its F1-score in the open-world setting.

TABLE VI: Hyperparameter tuning.

Parameter	Search space	Selected value
$\phi_{\min}$	$\{0.5, 0.51, \ldots, 0.99\}$	0.95
ϵ	$\{10s, 50s, 100s, 200s, 300s, 500s\}$	$300s$
α	$\{0.05, 0.1, \ldots, 0.95\}$	0.1
n_f	$\{1, 2, 3, 5, 10, 20, 50\}$	3
$\psi_{\min}$	$\{0.05, 0.1, \ldots, 0.95\}$	0.1

C. Estimating CPL Rate And RTT

We estimate RTT as the average time delay between timestamps of SYN packet and ACK packet in the TCP handshake. As for CPL rate, we estimate it in a passive manner by analyzing retransmitted packets in TCP/IP traffic captured on devices. Therefore, we only consider CPL rate associated to outbound packets sent by mobile devices. For a traffic instance, we estimate its CPL rate as m_t^r / m_t, where m_t is the number of outbound TCP packets associated to the app that this traffic instance belongs to and m_t^r is the number of retransmitted outbound TCP packets. We focus on TCP packets because TCP packets are dominating in Android app traffic (more than 98% in our dataset). The CPL rate (resp. RTT) of an app is computed as an average CPL rate (resp. RTT) for its traffic instances.