Towards Fine-Grained Open-World Android App Fingerprinting

Jianfeng Li[1], Hao Zhou[1], Shuohan Wu[1], Xiapu Luo[1]*, Ting Wang[2], Xian Zhan[1], Xiaobo Ma[3]
[1]*Department of Computing, The Hong Kong Polytechnic University*
[2]*College of Information Sciences and Technology, Pennsylvania State University*
[3]*School of Computer Science and Technology, Xi'an Jiaotong University*

Abstract

Despite the widespread adoption of encrypted communication for mobile apps, adversaries can still identify apps or infer selected user activities of interest from encrypted mobile traffic via app fingerprinting (AF) attacks. However, most existing AF techniques only work under the closed-world assumption, thereby suffering potential precision decline when faced with apps unseen during model training. Moreover, serious privacy leakage often occurs when users conduct some sensitive operations, which are closely associated with specific UI components. Unfortunately, existing AF techniques are too coarse-grained to acquire such fine-grained sensitive information. In this paper, we take the first step to identify method-level fine-grained user action of Android apps in the open-world setting and present a systematic solution, dubbed FOAP, to address the above limitations. First, to effectively reduce false positive risks in the open-world setting, we propose a novel metric, named structural similarity, to adaptively filter out traffic segments irrelevant to the app of interest. Second, FOAP achieves fine-grained user action identification via synthesizing traffic and binary analysis. Specifically, FOAP identifies user actions on specific UI components through inferring entry point methods correlated with them. Extensive evaluations and case studies demonstrate that FOAP is not only reasonably accurate but also practical in fine-grained user activity inference and user privacy analysis.

1 Introduction

Mobile devices in markets have bloomed unprecedentedly in the last decade [38]. The ubiquitous mobile devices, such as smartphones and tablets, have become necessities in modern life. Their prosperity and success are largely attributed to diverse apps running on them [36, 37]. These apps substantially extend the capability of off-the-shelf mobile devices and increasingly improve modern life in different aspects.

*The corresponding author.

While offering convenience, mobile apps also raise privacy concerns. Massive amounts of private user data are transmitted to and stored in cloud servers. Once these servers are compromised, catastrophic privacy leakage will happen subsequently [6]. The adversary can also indirectly infer sensitive private information, such as diseases, religious preferences, and individual location, through side-channel attacks [26, 35].

Despite the widespread adoption of encrypted communication, mobile apps are still susceptible to app fingerprinting (AF) attacks [2, 3, 14, 40–42, 44], which are essentially side-channel attacks. In such attacks, adversaries recognize apps or selected user activities of interest using pre-trained machine learning models without inspecting the packet payload plaintext, thereby immune to traffic encryption. However, existing AF techniques face two major limitations.

Closed-World Assumption. The vast majority of existing AF techniques (e.g., [15, 22, 30]) formulate app identification as a multi-class classification problem. It implies that they cannot correctly identify apps that are unseen during model training, because these apps will be erroneously classified into known classes. A straightforward solution is training a one-vs.-rest binary classifier for each app. Nevertheless, such a plausible solution only works under the closed-world assumption, i.e., apps as the negative class in testing stage should be presented during model training. If not, false positive cases may tremendously increase. Unfortunately, involving all rest apps in the training dataset is impossible as Android and iOS both host more than 3.4 million third-party apps [36, 37]. Even worse, recent studies [44, 52] pointed out that apps extensively use third-party libraries, leading to similar network behaviors, which may increase the risks of false positives.

Identification Granularity. Serious privacy leakage often occurs when users conduct some sensitive operations on certain UI components. For instance, in some COVID-19 contact tracing apps, the click action on the button to report positive testing can be a strong indicator of COVID-19 infection. However, information obtained through existing AF techniques is too coarse-grained to infer the above user privacy because these techniques focus on either app identification [40, 44, 50]

or inferring selected user activities of interest [15, 22, 30], and none of them can identify user actions on specific UI components. Additionally, previous works need labor-intensive efforts to manually label selected user activities, limiting their scalability in the wild.

In this paper, we present FOAP, a novel Fine-grained Open-world Android App fingerPrinting technique, to address the above limitations. First, FOAP carries out *open-world* app recognition, obviating the need for the closed-world assumption. The core challenge is the risks of false positives in the open-world setting. We solve it by exploiting the fact that two different apps seldom exhibit completely identical network behaviors. Specifically, we propose a novel metric named *structural similarity* to characterize how network flows within a traffic segment behave similarly to those generated by the app of interest. By leveraging this metric, FOAP effectively reduces false positives by adaptively filtering out traffic segments that are irrelevant to the app of interest.

Second, FOAP aims at *method-level* fine-grained user action identification via synthesizing traffic and binary analysis. We focus on identifying user actions on specific UI components, e.g., clicking a certain button. To this end, we characterize user actions using the corresponding entry point (EP) methods, including callbacks of UI components and lifecycle methods of Android components, and transform user action identification to EP method identification. In the training stage, the major challenge is how to automatically label network flows with EP methods in the face of various ways to trigger network flows. We solve it through extensive app analysis and implement the automatic labeling based on Android framework instrumentation. Automatic labeling brings two salient advantages: i) it endows FOAP with the capability of exploring all possible user actions that may generate network flows, and thus FOAP can identify not only selected user activities but also other user actions that trigger network traffic; ii) it obviates the need of manual labeling, facilitating the scalability of FOAP. In the identification stage, the major challenge is the multi-label flow. A network flow is often correlated with multiple EP methods, leading to a multi-label classification problem. We relax this problem to a series of multi-class classification sub-problems by extracting in-flow bursts. To identify EP methods associated with a network flow, we model the spatial-temporal contextual dependency of in-flow bursts within it and infer the EP method associated with each of them based on the conditional random field.

In summary, this paper makes the following contributions:

- To the best of our knowledge, FOAP is the first approach for conducting method-level fine-grained app user action recognition in the open-world setting. We release its source code and the datasets at https://github.com/jflixjtu/FOAP.
- We address several challenging issues in the design of FOAP. First, we design structural similarity, a novel metric for effectively reducing the false positive risks in open-world app traffic recognition. Second, we propose a new approach to automatically label network flows with EP methods and build a spatial-temporal context model to accurately identify EP methods from network flows.
- We implement a prototype of FOAP and conduct extensive experiments to evaluate it. The results show that FOAP outperforms baseline approaches in open-world app recognition, improving the F1-score from 0.679 to 0.911. It is also reasonably accurate in EP method identification with an average F1-score of 0.885. We present three case studies to demonstrate FOAP's practicality in fine-grained user activity inference and user privacy analysis.

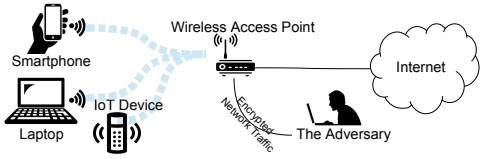


Figure 1: The threat model.

2 Overview

2.1 Threat Model

As shown in Figure 1, the adversary considered in this paper sniffs network traffic on a wireless access point. His goal is to recognize network flows generated by the app of interest and identify what EP methods of this app trigger them to infer fine-grained user actions. We define a network flow as a sequence of packets corresponding to a socket-to-socket communication identified by a unique combination of source and destination addresses and port numbers, together with transport protocols. In this paper, we focus on TCP flows, because i) TCP packets are dominating in Android app traffic (98.6% in our dataset), and ii) we mainly consider encrypted network traffic based on TCP protocol (e.g., HTTPS, etc.).

We assume that the adversary can trace back all network flows to different devices according to source addresses. We also assume the adversary *cannot* exploit i) packet plaintext payload and ii) destination feature of network flows. Our assumptions match the real-world scenario of AF attacks. First, since app network flows are commonly encrypted [44], the adversary cannot access the plaintext payload in most cases. Second, due to the wide use of encrypted proxy agents (e.g., ShadowSocks), the real destinations of network flows are often invisible. Even if the adversaries can obtain the real destinations, such features are not reliable since they may change across networks due to the widespread adoption of CDN. Moreover, the share of third-party libraries among apps leads to the share of destinations across apps.

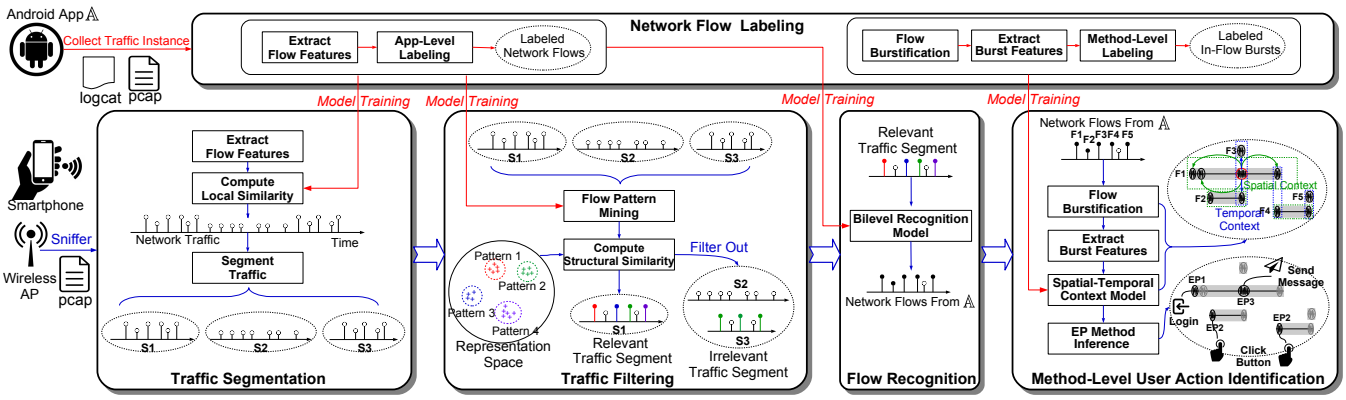


Figure 2: The workflow of FOAP. Training stage (resp. identification stage) is colored in red (resp. blue).

2.2 Workflow of FOAP

Without loss of generality, we assume $\mathbb{A}$ is one of the apps of interest. FOAP identifies method-level user actions of $\mathbb{A}$ from encrypted network traffic in the open-world setting. Figure 2 illustrates the workflow of FOAP.

Training Stage:

• ***Network Flow Labeling*** (§ 3) plays a core role in constructing training datasets. By either automatically or manually running $\mathbb{A}$, we collect $\mathbb{A}$'s traffic instances, each of which contains a log file (in Android, we call it logcat) and a traffic file in pcap format. We first label whether a network flow is from $\mathbb{A}$, i.e, app-level labeling. If yes, we next label when and what EP methods of $\mathbb{A}$ trigger this network flow, i.e., method-level labeling. We extract a feature vector for every network flow after app-level labeling to construct the training dataset for i) *Traffic Segmentation*, ii) *Traffic Filtering*, and iii) *Flow Recognition*. Since a network flow usually contains multiple in-flow bursts triggered by different EP methods, we divide a network flow into a series of in-flow bursts and extract a feature vector for each one to construct the training dataset for *Method-Level User Action Identification*.

Identification Stage:

• ***Traffic Segmentation*** (§ 5.1) divides network traffic into different time periods, such that there are some time periods during which $\mathbb{A}$ is active (if they exist). To this end, we extract a feature vector for each network flow and compute its *local similarity* with $\mathbb{A}$ to characterize how likely it is generated by $\mathbb{A}$, from the perspective of a single flow. Based on the local similarity, we segment network traffic to locate all possible time periods when $\mathbb{A}$ is probably active. In Figure 2, we represent each network flow with a bar. The time-axis location (resp. height) of a bar represents the start time (resp. local similarity) of the corresponding flow. Network traffic is divided into a series of traffic segments, i.e., $\mathbf{s}_1$, $\mathbf{s}_2$, and $\mathbf{s}_3$.

• ***Traffic Filtering*** (§ 5.2) is designed for mitigating false positives in the open-world setting. By leveraging flow pattern mining, we quantify how likely a network flow matches a certain pattern of $\mathbb{A}$'s network flows in the feature representation space and then use this pattern-related information to profile structural characteristics of a traffic segment. For each traffic segment, we compute its *structural similarity* with $\mathbb{A}$ by synthesizing pattern-related information from all network flows within it. Structural similarity is informative to reflect how likely $\mathbb{A}$ is active during the time period corresponding to a traffic segment. We take advantage of such a metric to identify traffic segments relevant to $\mathbb{A}$ (labeled by $\mathbb{A}$) and filter out other irrelevant traffic segments (labeled by non-$\mathbb{A}$) without further analysis. In Figure 2, $\mathbf{s}_1$ has a high structural similarity with $\mathbb{A}$ because network flows within it match various patterns of $\mathbb{A}$, whereas $\mathbf{s}_2$ has a very low structural similarity with $\mathbb{A}$ because network flows within it match no pattern of $\mathbb{A}$. Even though some network flows within $\mathbf{s}_3$ have high local similarities with $\mathbb{A}$, $\mathbf{s}_3$ has a relatively low structural similarity with $\mathbb{A}$ because these network flows only match one pattern of $\mathbb{A}$ (colored in green), implying $\mathbf{s}_3$ is structurally different from network traffic generated by $\mathbb{A}$. Finally, $\mathbf{s}_1$ is identified as a relevant traffic segment of $\mathbb{A}$, whereas $\mathbf{s}_2$ and $\mathbf{s}_3$ are identified as irrelevant traffic segments of $\mathbb{A}$. An irrelevant traffic segment of $\mathbb{A}$ may contain network traffic generated by more than one apps except $\mathbb{A}$, while relevant traffic segments of different apps may overlap.

• ***Flow Recognition*** (§ 5.3) constructs a bilevel recognition model to identify whether a network flow in the relevant traffic segment is generated by $\mathbb{A}$. This model considers not only the feature vector of a network flow but also contextual information from surrounding network flows in favor of better recognition accuracy.

• ***Method-Level User Action Identification*** (§ 6) infers which entry point (EP) methods trigger a network flow of $\mathbb{A}$ to characterize fine-grained user actions. Network flows corresponding to persistent connections often contain multiple packet bursts, i.e., in-flow bursts, each of which may be triggered by different EP methods. Network flows corresponding to short connections often contain only one in-flow burst that is triggered by an EP method. To identify EP methods, we first extract in-flow bursts and their feature vectors. Next, we construct a spatial-temporal context model to characterize the

correlation between in-flow bursts. Based on this model, we identify EP methods to infer fine-grained user actions, such as clicking on a button.

3 Network Flow Labeling

To construct the training dataset for open-world app traffic recognition and method-level user action identification, we capture network flows generated by the tested apps and label each network flow with the app and EP methods (i.e., the component lifecycle or the UI callback) that generate it. To this end, we instrument Android framework to i) extract socket information to correlate network flows, ii) retrieve process identifier (PID) to obtain the package name and collect stack trace to determine the EP method. The collected data is written to `logcat`.

3.1 Resolving Network Socket

Since Android apps commonly use network sockets to generate network flows, we resolve each network socket operation performed by the tested apps to correlate the captured network flows. According to the observation that framework APIs for performing socket operations (e.g., `connect` defined in `Socket`) rely on socket related native system functions (e.g., `connect` exported by `libc.so`) to complete tasks, we instrument socket related system functions to get a unique 4-tuple for each TCP socket. Moreover, since identifiers (i.e., PID and TID) of the process and thread, executing the socket related system functions, are essential for obtaining the package name and correlating the EP method, we make the instrumented socket related functions invoke `getpid` and `gettid` to retrieve the PID and TID values.

It is worth noting that, the state-of-the-art work [20] cannot correlate the socket operations implemented in apps' native code to their corresponding EP methods because it just analyzes socket related framework APIs. Since our approach analyzes the socket related system functions, which are internally called by socket related framework APIs, we can associate socket operations implemented in both Java code and native code with their EP methods.

3.2 Analyzing Network Socket Operation

In order to correlate each network flow to an app as well as EP methods of this app, we further analyze each network operation performed by the tested apps.

Correlating Network Flow to App: Relying on PIDs retrieved when the socket related system functions are invoked, we get the unique identifier of the app, performing the socket operations, by accessing the `/proc/PID/cmdline` file, which keeps app's package name.

Correlating Network Flow to EP Method: Since socket operations must be carried out in non-UI threads of apps [1], apps can create a new thread by internally calling the `clone` function in `libc.so` to perform network operations (case-1). Therefore, we instrument `clone` to get the stack trace and TID of each created thread, so that we can correlate the thread with EP method, which is presented in the stack trace and leads to the creation of the thread. If the TID equals the one retrieved when socket related functions are invoked, we correlate the EP method with generated network flows.

Instead of creating a new thread, apps can reuse a thread to perform different network operations. For example, apps can send messages to notify the handler running in a non-UI thread (case-2) or submit tasks to an idle thread managed by the thread pool (case-3) to conduct socket operations. It is non-trivial to handle these cases because we need to determine the EP method for each network operation conducted in the reused thread. In the following, we detail how we label network flows in these cases.

For case-2, we use the `Message` objects, which notify the message handler to perform socket operations, to distinguish every network operation conducted by the handler. Specifically, we instrument `enqueueMessage` defined in the `Handler` class to get the stack trace and the message (i.e., the `Message` object) sent to the handler so that we can correlate the message with the EP method that results in the sending of the message. Meanwhile, to correlate each message with the thread that runs the handler, we instrument `dispatchMessage` of the `Handler` class to get the received message and thread's TID. If the TID equals the one retrieved when socket related functions are invoked, we correlate the received message with generated network flows. Since the received and sent messages refer to the same `Message` object, we then correlate the EP method to the network flows.

For case-3, we use the `Runnable` objects, which are executed in the thread managed by the thread pool, to differentiate every network operation performed by the same thread in the thread pool. Specifically, we instrument `execute` defined in the `ThreadPoolExecutor` class to get the stack trace and the task (i.e., the `Runnable` object) submitted to the thread pool, so that we can correlate the task with the EP method that causes the submission of the task. Meanwhile, to correlate each task with the thread that runs the task, we instrument `runworker` of the `ThreadPoolExecutor` class to get the executed task and the thread's TID. If the TID equals the one retrieved when socket related functions are invoked, we correlate the executed task with the network flows. Since the submitted and the executed tasks refer to the same `Runnable` object, we then correlate the EP method to the network flows.

Note that, for those UI callbacks which are identified as EP methods, we associate them with their corresponding resource IDs. For example, we correlate each `onClick` callback with the resource ID of the relevant UI component that registered the click event listener.

4 Traffic Feature Extraction

Although some studies [5, 33, 34] take advantage of deep learning models to automatically extract traffic features, deep learning models require large amounts of training data to avoid model overfitting. In our problem, the number of network flows/in-flow bursts triggered by different EP methods is unbalanced. Some apps/EP methods only have few samples. Therefore, deep-learning-based feature extraction is not suitable for our problem. Either a network flow or an in-flow burst is essentially a packet sequence. We extract 123-dimensional features for both of them from five perspectives.

• **General characteristic** (8 features). We extract inbound-/outbound packet number, bidirectional packet number, inbound/outbound packet percentage, inbound/outbound bytes, and the duration of packet sequence.

• **Interactive pattern** (20 features). We characterize the interactive pattern between endpoints using a function $f(x) = (i-x)\sum_{j=1}^{i-1} d_j + (x-i+1)\sum_{j=1}^{i} d_j$ for $i-1 < x \leq i$, where $d_j = 1$ (resp. $d_j = -1$) if the jth packet is an inbound (resp. outbound) packet. Assume the packet sequence contains n packets. The interval $(0, n)$ is divided into 20 bins with the size of $n/20$. We pick $f(x)$ at the center of each bin as a feature and totally get 20 features to approximate $f(x)$ (or, equivalently, the interactive pattern).

• **Packet rate characteristic** (5 features). The duration of a packet sequence is divided into a series of time windows with the size of 1 second. Mean, maximum, minimum, median, and standard deviation of packet number in a time window (i.e., packet rate), are computed as features.

• **Temporal characteristic** (39 features). We consider bidirectional, inbound, and outbound packet arrivals respectively. For each one, mean, maximum, minimum, and standard deviation of packet interval time, and 9 percentiles (from 10% to 90%) of relative arrival time, i.e., the time lag relative to the first packet, are extracted as features.

• **Packet size characteristic** (51 features). We extract various packet size statistics for inbound, outbound, and bidirectional packets respectively, including mean, maximum, minimum, median absolute deviation, standard deviation, variance, skew, kurtosis, and 9 percentiles (from 10% to 90%).

5 Open-World App Traffic Recognition

Open-world app traffic recognition aims to recognize app-specific traffic. More specifically, given a network flow, FOAP should identify whether it is from the app of interest, say $\mathbb{A}$.

Open world vs. closed world. We define open-world setting along with closed-world setting in this paper. Let $\mathcal{S}_T = \{\text{app}_T^i\}_{i=1}^{m_T}$ (resp. $\mathcal{S}_I = \{\text{app}_I^i\}_{i=1}^{m_I}$) be the set comprised of all apps in the training stage (resp. the identification stage). In the closed-world setting, we have $\mathcal{S}_I \subseteq \mathcal{S}_T$. In the open-world setting, $\mathcal{S}_I$ might not be a subset of $\mathcal{S}_T$ but it could be.

In the open-world setting, a key challenge is how to effectively reduce false positives in face of apps that are unseen during model training. FOAP tackles it in two steps, i.e., traffic segmentation and traffic filtering. After that, FOAP constructs a bilevel recognition model to identify whether a network flow is generated by $\mathbb{A}$ by taking advantage of not only its feature vector but also its contextual information.

5.1 Traffic Segmentation

The goal of traffic segmentation is dividing network traffic into different time periods, such that there are some time periods during which $\mathbb{A}$ is active (if they exist). To this end, we first propose a metric, named by *local similarity*, to quantify how likely a network flow is generated by $\mathbb{A}$ from the perspective of a single flow.

Definition 1. *Local similarity is a metric to quantify how a network flow f is similar to network flows from $\mathbb{A}$ in terms of the intrinsic features from f itself. We compute f's local similarity with $\mathbb{A}$ by $S_l(f,\mathbb{A}) = Pr\{f \in \mathbb{A}|\mathbf{F}(f)\}$, where $\mathbf{F}(\cdot)$ is a function that returns the feature vector of f.*

We denote by $\mathbf{f} = (f_1, f_2, \ldots, f_M)$ the sniffed network traffic consisting of M network flows, where the ith flow is denoted by f_i and its start time is t_i. Let q_i be f_i's local similarity with $\mathbb{A}$, i.e., $S_l(f_i,\mathbb{A})$. To compute q_i, we first extract f_i's feature vector (see § 4) and feed it to a pre-trained random forest (RF) classifier. q_i is estimated as the percentage of decision trees that predict f_i is from $\mathbb{A}$. Next, we segment network traffic based on $q_1, q_2, \ldots, q_M$. Based on the measurement over our dataset, network flows generated by an app, say $\mathbb{A}$, have significantly higher local similarity with this app, compared with network flows generated by other apps, thereby resulting in obvious difference of local similarity between time periods dominated by $\mathbb{A}$ and time periods dominated by other apps. Such a difference is about 4.51 times the local similarity variance for different network flows from an app. Inspired by this observation, we expect network flows within the same traffic segment have similar local similarity with $\mathbb{A}$ (i.e., the small variance within a segment), whereas network flows belonging to adjacent traffic segments have significantly different local similarities with $\mathbb{A}$ (i.e., large variance between adjacent segments). By doing so, we are able to extract all possible traffic segments when $\mathbb{A}$ is active. Formally, we formulate traffic segmentation as a combinational optimization problem.

$$\begin{aligned} \min_{\mathbf{z}} \quad & \sum_{c=0}^{\max(\mathbf{z})} |Q_c| \cdot \text{Var}(Q_c) + \lambda \max(\mathbf{z}), \\ \text{s.t.} \quad & z_i \in \mathbb{N},\ z_1 = 0,\ 0 \leq z_{i+1} - z_i \leq 1 \text{ for } 1 \leq i < M, \\ & Q_c = \{q_i | z_i = c\},\ |Q_c| \geq n_{\min}, \\ & \max\{t_i | z_i = c\} - \min\{t_i | z_i = c\} \geq \tau_{\min}, \end{aligned} \tag{1}$$

where z_i is a variable indicating which segment the ith network flow belongs to, $\lambda \max(\mathbf{z})$ is a regularization term that

penalizes the complexity of results, $n_{\min}$ is the minimum number of network flows that a segment should contain, and $\tau_{\min}$ is the minimum time span of a segment. The above optimization problem can be solved based on integer programming [8]. To speed up the traffic segmentation, we propose a divisive-agglomerative tree method to solve it in a greedy fashion. Divisive tree recursively generates small segment candidates while agglomerative tree merges some of them to minimize the loss in (1). We elaborate this method in § A.

5.2 Traffic Filtering

If $\mathbb{A}$ is active during the time period of a traffic segment, we refer to it as a *relevant traffic segment* and otherwise an *irrelevant traffic segment*. Traffic filtering aims to filter out all irrelevant traffic segments.

5.2.1 Flow Pattern Mining

In this step, we quantify how likely a network flow matches a certain pattern of network flows generated by $\mathbb{A}$ in the feature representation space. This pattern-related information will be used for profiling structural characteristics of a traffic segment. To this end, we need to identify different patterns by clustering $\mathbb{A}$'s network flows over the training dataset. Unfortunately, the original feature space is irregular because i) both continuous and discrete features are considered and ii) the range and variance of different features are tremendously diverse. It is difficult to find a proper distance metric for reasonable clustering analysis in the original feature space. To tackle this problem, we learn a mapping function based on metric learning [46] to map original feature vectors to a low-dimensional representation space, where we expect the euclidean distance between network flows in the same pattern, i.e., the same cluster, will be minimized whereas that between network flows in different patterns, i.e., different clusters, will be maximized. We elaborate the learning of mapping function in § B.1.

Clustering Analysis. We map $\mathbb{A}$'s network flows in the training dataset to the representation space and conduct clustering analysis to identify flow patterns as different clusters. Hierarchical clustering is employed for this task as it obviates the need to specify the cluster number, which is unknown a prior. We specify the clustering threshold as δ and obtain m clusters $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_m$. We refer to these clusters as *patterns* of $\mathbb{A}$'s network flows. Next, we train a multi-class k-nearest neighbors classifier in the representation space to identify which pattern a new network flow belongs to. Let P_i be the identified pattern that f_i belongs to. We reuse f_i's local similarity q_i to quantify the *likelihood* of f_i matching pattern P_i.

5.2.2 Filtering Out Irrelevant Traffic Segment

To filter out irrelevant traffic segments, we compute traffic segments' *structural similarity* with $\mathbb{A}$.

Definition 2. *Structural similarity characterizes how network flows within a traffic segment $\mathbf{s}_k$ are similar to $\mathbb{A}$'s network flows in various patterns. It is computed by*

$$S_s(\mathbf{s}_k, \mathbb{A}) = \frac{\int_{q_{\min}}^{1} |\mathcal{H}(\mathbf{s}_k, q)| dq}{|\mathcal{H}(\mathbf{s}_k, 0)|(1 - q_{\min})}, \tag{2}$$

where $\mathcal{H}(\mathbf{s}_k, q) = \{P_i | q_i > q, f_i \in \mathbf{s}_k\}$ is a function that returns a set containing all patterns that network flows within $\mathbf{s}_k$ belong to with the likelihood greater than q.

We consider network flows match certain patterns only if their likelihood are greater than $q_{\min}$. In this paper, we set $q_{\min} = 0.5$ by default. The value of $S_s(\mathbf{s}_k, \mathbb{A})$ ranges from 0 to 1. A smaller value of $S_s(\mathbf{s}_k, \mathbb{A})$ indicates $\mathbf{s}_k$ is more likely an irrelevant traffic segment. Formally, $\mathbf{s}_k$ will be deemed to be an irrelevant traffic segment and filtered out if $S_s(\mathbf{s}_k, \mathbb{A}) < S^{\mathbb{A}}_{\min}$. In here, $S^{\mathbb{A}}_{\min} \in [S_{\min}, S_{\max}]$ is an app-specific similarity threshold bounded by $S_{\min}$ and $S_{\max}$. We specify it to be the value such that 95% network flows from $\mathbb{A}$ over the training dataset will not be erroneously filtered out. Despite a potential slight decline of recall for traffic recognition, such a similarity threshold can effectively reduce false positives by filtering out irrelevant traffic segments as many as possible.

5.3 Flow Recognition

Through traffic filtering, we obtain relevant traffic segments. However, not all network flows within them must be from $\mathbb{A}$ because they may contain network flows from background system services and other apps. To recognize whether a network flow is from $\mathbb{A}$, we construct a bilevel recognition model. The *low-level* model is comprised of two binary random forest classifiers. The first classifier is used for computing the local similarity and has been trained before. The second classifier predicts the probability that a network flow is a background flow, (i.e., network flows generated by background system services). For a network flow f_i, the first classifier outputs its local similarity with $\mathbb{A}$, denoted by q_i, while the second classifier outputs the probability that it is a background flow, denoted by r_i. Both q_i and r_i play important roles in the high-level model. The *high-level* model is a logistic regression classifier. We choose this model because of its excellent generalization capability due to a small number of parameters. For a network flow f_i, the high-level model takes advantage of features not only from itself but also from its contextual network flows. Specifically, we consider network flows within $\mathcal{N}_i$, a time window centered at the start time of f_i with the size of T, as f_i's contextual network flows. We construct f_i's feature vector for high-level model as $\mathbf{h}_i = (q_i, \bar{q}_{\mathcal{N}_i}, q_i - r_i)$, where $\bar{q}_{\mathcal{N}_i} = \sum_{j \in \mathcal{N}_i} q_j / |\mathcal{N}_i|$. For any network flow in relevant traffic segments, we recognize whether it is from $\mathbb{A}$ according to the output of the high-level model. To avoid mutual interference, we decouple the training of low-level and high-level models by leveraging bootstrap resampling method [12, 45].

6 Method-Level User Action Identification

After recognizing network flows from $\mathbb{A}$, we further infer what EP methods trigger them. Such information plays a critical role in characterizing fine-grained user actions.

Different from the binary classification in recognizing app-specific network flows, EP method identification can be formulated as a multi-label classification because a network flow may correspond to multiple EP methods. Network flows in app traffic can be roughly grouped into two categories. The first category corresponds to *persistent connection*. Network flows in this category often contain multiple packet bursts associated with sequentially invoked EP methods. The second category corresponds to *short connection*. Network flows in this category often contain only one packet burst associated with a single EP method. Instead of regarding a network flow as a whole to extract its feature vector and identify what EP methods trigger it, we identify EP methods associated with all packet bursts within it. By doing so, our method has three-fold advantages. First, we relax the multi-label classification to a series of multi-class classification subproblems. Second, we can make use of contextual dependency between bursts to improve identification accuracy. Third, we obtain a higher temporal resolution in characterizing user actions.

6.1 Flow Burstification

The first step is extracting in-flow bursts from network flows.

Definition 3. *Given a burst threshold* ε*, an in-flow burst is a subsequence of packets within a network flow. Time interval between two packets in an in-flow burst is less than* ε*, while that between two in-flow bursts is not less than* ε*.*

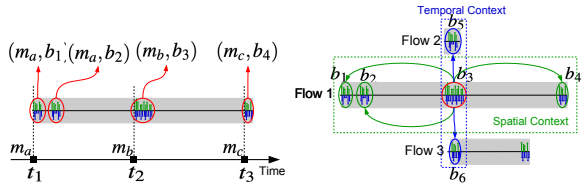
Figure 3(a) illustrates an example, where we extract four in-flow bursts from a network flow. For each in-flow burst, we extract a 123-dimensional feature vector (see § 4).

• **Burst Labeling.** To construct the training dataset for EP method identification, we need to label in-flow bursts. Assume that the network flow labeling module has obtained three EP methods, i.e., m_a, m_b, and m_c, and their invocation time t_1, t_2, and t_3 as shown in Figure 3(a). Before labeling in-flow burst, we first label each packet to indicate which method triggers it. Specifically, a packet is labeled by the first EP method invoked before it. We then label an in-flow burst by aggregating the label of packets belonging to this burst (majority voting). Finally, we obtain four in-flow bursts (m_a, b_1), (m_a, b_2), (m_b, b_3), and (m_c, b_4).

6.2 Spatial-Temporal Context Model

We model contextual dependency of in-flow bursts based on conditional random field [39] due to its flexibility to capture both spatial and temporal context.

• **Spatial Context:** It reflects the contextual relationship between bursts within the same network flow. In Figure 3(b), bursts b_1, b_2, b_3, and b_4 are within the same network flow and they are mutually spatially contextual bursts.



(a) Label within-flow bursts. (b) Spatial-temporal context.

Figure 3: Flow burstification and spatial-temporal context.

• **Temporal Context:** An EP method may trigger multiple network flows simultaneously, yielding concurrent in-flow bursts. To capture the temporal context, we specify an interval threshold τ_{burst}. Two in-flow bursts are temporally contextual if they belong to different network flows but the time interval between them is less than τ_{burst}. In Figure 3(b), bursts b_3, b_5, and b_6 are mutually temporally contextual bursts.

Formally, let $\mathcal{M}_{\mathbb{A}} = \{m_k\}_{k=1}^{K}$ be a set comprised of EP methods of $\mathbb{A}$ that may trigger network flows. We denote by $\mathbf{b} = (b_1, b_2, \ldots, b_N)$ a sequence of in-flow bursts extracted from $\mathbb{A}$'s network flows and $\mathbf{y} = (y_1, y_2, \ldots, y_N) \in \mathcal{M}_{\mathbb{A}}^N$ the underlying EP method invocations that trigger these in-flow bursts. The posterior distribution of $\mathbf{y}$ can be expressed by

$$p(\mathbf{y}|\mathbf{b}) = \frac{1}{Z(\mathbf{b})} \prod_{i=1}^{N} \left\{ \Omega_i(y_i, b_i) \prod_{j \in \mathcal{C}_i^S} \Psi_{ij}(y_i, y_j, \mathbf{b}) \prod_{j \in \mathcal{C}_i^T} \Phi_{ij}(y_i, y_j, \mathbf{b}) \right\}, \quad (3)$$

where $\Omega_i(y_i, b_i)$ is a unary potential characterizing the relationship between y_i and b_i, $\Psi_{ij}(y_i, y_j, \mathbf{b})$ (resp. $\Phi_{ij}(y_i, y_j, \mathbf{b})$) is a spatially (resp. temporally) pairwise potential characterizing the relationship between b_i and its spatially (resp. temporally) contextual burst b_j, $\mathcal{C}_i^S$ (resp. $\mathcal{C}_i^T$) is a set comprised of all spatially (resp. temporally) contextual bursts of b_i, and $Z(\mathbf{b})$ is a normalizing constant. We elaborate how to construct $\Omega_i(y_i, b_i)$, $\Psi_{ij}(y_i, y_j, \mathbf{b})$, and $\Phi_{ij}(y_i, y_j, \mathbf{b})$ in § C.1. These functions are parameterized by the model parameters of CRF. We elaborate model parameter learning in § C.2.

6.3 Iterative Inference of EP Method

Given the CRF-based spatial-temporal context model, the problem of EP method identification is transformed into inferring the hidden variables (i.e., EP method invocations) from the observable variables (i.e., in-flow bursts). For general graphs, like our spatial-temporal context model, the exact inference of CRFs is intractable because it requires exponential time in the worst case [39]. Therefore, we propose a greedy EP method inference algorithm to conduct approximate inference. Instead of estimating the posterior marginal distribution, this algorithm derives the most likely EP method sequence, i.e., $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} p(\mathbf{y}|\mathbf{b})$ in an iterative fashion. Let $\mathbf{y}^{(k)} = (y_1^{(k)}, y_2^{(k)}, \ldots, y_N^{(k)})$ be the estimation of $\mathbf{y}$ in the kth

round. We derive $y_i^{(k)}$ as the EP method that maximizes the probability $p(y_i|\mathbf{b}, \mathbf{y}^{(k-1)}/y_i^{(k-1)})$. We record $\mathbf{y}^{(k)}$ in the set $\mathcal{Y}$ and $p^{(k)} = Z(\mathbf{b})p(\mathbf{y}^{(k)}|\mathbf{b})$ will also be stored. The iteration process repeats until it exceeds the maximum iteration number or it meets the early stopping condition, i.e., $\mathbf{y}^{(k)} \in \mathcal{Y}$. The resulting inference is derived as $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}^{(k)} \in \mathcal{Y}} p^{(k)}$. We summarize this process as Algorithm 1 in § D. Given EP methods associated with all in-flow bursts, we infer what EP methods trigger a network flow by aggregating EP methods associated with in-flow bursts within it.

7 Evaluation

We first evaluate FOAP in open-world app traffic recognition and then evaluate it in EP method identification. We next analyze various factors that may influence the performance of FOAP, such as experimental data size and third-party libraries. To investigate how FOAP trained over the dataset generated by automatic test tools performs for network traffic generated by human users, we conduct cross-dataset evaluation. Finally, we analyze the running time of FOAP.

7.1 Dataset Construction

Because our work constitutes the first effort towards method-level open-world app fingerprinting, there is no available public dataset where method-level labels of network flows are available. Therefore, we download 1000 apps from Google Play Store and collect app traffic in our testbed. When selecting apps, we give consideration to both popularity and comprehensiveness. First, we select the most popular apps according to the rank in Google Play Store in favor of better popularity. Second, the selected apps fall into 46 different categories to guarantee a reasonable comprehensiveness. Note that when selecting apps, we skip various mobile browsers and apps that need to sign up with a credit card to avoid potential legal risk. We generate app traffic instances with the aid of open-source tools Monkey[1] and RERAN [16]. Monkey is an automated test tool, while RERAN is a record and replay tool for Android. For every app, we collect 50 traffic instances. To generate each traffic instance, we run apps on Google Pixel 2 smartphones with Monkey to generate pseudo-random streams of user events and simulate user operations. The maximum number of user events is set to be 5000. Meanwhile, network traffic of smartphones is captured by Tcpdump and stored in pcap format for network flow extraction; logcat files are saved for network flow labeling. In a nutshell, we collect $50,000$ traffic instances, consisting of $2,701,931$ network flows. The total file size is more than 1.29 TB.

7.2 Open-World App Traffic Recognition

We first evaluate FOAP in recognizing network flows.

[1] https://developer.android.com/studio/test/monkey

• **Experimental Setup.** Since FOAP establishes a model for every individual app of interest, we construct a training dataset and testing dataset for each app. Specifically, we randomly split traffic instance set into two subsets: 40 instances for training and 10 instances for testing. To construct the training dataset of an app, say $\mathbb{A}$, besides $\mathbb{A}$'s traffic instances, we randomly choose n_T other apps to involve their traffic instances as negative samples in the training dataset. As for $\mathbb{A}$'s testing dataset, similar to the experimental setting in [44], we consider another 20 apps, which are unseen in $\mathbb{A}$'s training dataset. In other words, apps considered as negative class in testing stage are *completely* different from those for model training. To mimic human behavior, we simulate the use of different apps as Poisson process, which is commonly used to model human behaviors [32, 51]. Specifically, we construct network traces by merging traffic instances from different apps in random order. The time interval between traffic instances is exponentially distributed with an average time interval 1 second. Additionally, we also change the value of n_T to evaluate how it influences the accuracy of app recognition. We elaborate on the process of hyperparameter tuning in § E.

• **Baseline.** We compare FOAP with the state-of-the-art tool AppScanner [40]. Other works, such as [9, 10, 44], have different threat models. For example, [9, 10] only handle unencrypted data and [44] consider destination features, which are unavailable in our threat model. Another line of works, such as [15, 22, 30] focus on different objects for analysis. For example, [30] analyzes packets in a series of time windows to identify user activities and [22] divides packet arrivals into segments and analyzes user activities within each segment. Therefore, we skip a direct comparison with them in our experiment. The feature set of the vanilla AppScanner is just a subset of FOAP's feature set, i.e., 51 features related to packet size characteristic and 3 features related to general characteristic. To conduct a fair comparison, we also extend AppScanner by using the full feature set of FOAP. AppScanner is a modular tool, comprised of different classifiers. Among these classifiers we choose Single Random Forest Classifier Per App to evaluate because i) it achieves the best accuracy and ii) it can be evaluated in the open-world setting. All parameters are consistent with those in the original paper [40].

• **Result.** Table 1 reports the experimental result. Generally speaking, FOAP consistently outperforms AppScanner in terms of precision, recall, and F1-score for different n_T. For example, FOAP improves average precision with ~ 0.3, average recall with ~ 0.14, and average F1-score with ~ 0.23, compared to AppScanner for $n_T = 20$. FOAP achieves the highest F1-score 0.911 when $n_T = 20$. Compared with AppScanner, a significant advantage of FOAP is its much higher precision. It is because FOAP takes advantage of inter-flow information while AppScanner doesn't. Such information is important for FOAP to reduce false positives in the open-world setting. Specifically, FOAP makes use of both intrinsic features from a network flow itself and inter-flow information

Table 1: Evaluating FOAP in open-world app traffic recognition (mean±standard deviation).

n_T	AppScanner			AppScanner (extended)			FOAP (our approach)		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
5	0.315 ± 0.171	0.866 ± 0.105	0.439 ± 0.174	0.307 ± 0.158	0.840 ± 0.111	0.427 ± 0.166	0.759 ± 0.253	$\mathbf{0.925} \pm 0.094$	0.803 ± 0.200
10	0.476 ± 0.183	0.814 ± 0.127	0.582 ± 0.161	0.470 ± 0.178	0.774 ± 0.128	0.568 ± 0.159	0.886 ± 0.170	0.906 ± 0.111	0.881 ± 0.140
15	0.568 ± 0.189	0.782 ± 0.137	0.641 ± 0.158	0.575 ± 0.182	0.729 ± 0.144	0.627 ± 0.155	0.926 ± 0.123	0.897 ± 0.114	0.902 ± 0.111
20	0.641 ± 0.180	0.755 ± 0.146	0.679 ± 0.151	0.646 ± 0.164	0.700 ± 0.148	0.659 ± 0.143	$\mathbf{0.945} \pm 0.102$	0.897 ± 0.126	$\mathbf{0.911} \pm 0.115$

to improve recognition accuracy. FOAP computes structural similarity to characterize how network flows within a time period are structurally similar to those from the app of interest. Based on this metric, FOAP can effectively filter out irrelevant network flows and significantly reduce false positives. On the contrary, AppScanner recognizes a network flow by only using its intrinsic features and thus it is prone to generate false positives caused by similar network flows. Additionally, AppScanner only recognizes which app a network flow comes from, while FOAP not only recognizes the app that generates a network flow but also identifies the specific EP methods that generate the traffic to infer fine-grained UI operations.

• **Case Studies of False Positives.** We observed two salient factors that increase false positive rate.

i) *Share of third-party libraries*. Different apps often use the same third-party libraries (TPLs) to shorten development cycle, potentially resulting in similar network flows generated by them. For example, the comic app `com.tencent.wecomic` uses the TPL `com.facebook`, which is widely used by other apps. When FOAP recognizes network traffic generated by this app, the average false positive rate (FPR) is 4.56×10^{-3}, while its FPR associated with the TPL `com.facebook` is 1.10×10^{-2}, significantly higher than the average FPR. A similar situation is found for the sport app `com.bundesliga`, which uses the TPL `com.bumptech.glide`. We observed its average FPR is 4.63×10^{-3}, while its FPR associated with the TPL `com.bumptech.glide` is 1.75×10^{-1}.

ii) *Apps from the same developer*. Developers may reuse their codes when developing various apps. It aggravates the false positive risk when distinguishing traffic from these apps. For example, both `com.google.android.apps.meetings` and `com.google.android.apps.classroom` are developed by Google. To recognize network flows generated by the former app, FOAP reports more false positives caused by the latter app (FPR $= 1.13 \times 10^{-2}$) than other apps not developed by Google (FPR $= 2.36 \times 10^{-4}$). Another example is found for two apps developed by Microsoft. When recognizing network flows of `com.microsoft.office.outlook`, the app `com.microsoft.translator` produces more false positives (FPR $= 1.62 \times 10^{-2}$) than other apps (FPR $= 6.57 \times 10^{-4}$).

• **Case Studies of False Negatives.** We identified two key factors leading to false negatives.

i) *Dynamic nature*. Diverse user behaviors bring highly dynamic traffic, especially for information-intensive apps, like news apps. In these apps, clicking on various articles/videos may trigger diverse network flows, thereby increasing false negative risk. Take an example from the news app `bbc.mobile.news.ww`. FOAP recognizes its network flows with a false negative rate (FNR) 7.58×10^{-1}. A closer look reveals the network flows missed by FOAP (i.e., FN) indeed exhibit substantial difference from network flows in the training dataset, since their average local similarity with this app is 0.291, much lower than that for network flows successfully recognized by FOAP, i.e., 0.753. Similar situations can be observed for map apps. For example, FOAP recognizes network flows generated by `com.baidu.BaiduMap` with an FNR 0.107. The network flows missed (resp. recognized) by FOAP have an average local similarity 0.381 (resp. 0.832) with this app.

ii) *Low-traffic apps*. We observed that some false negative cases for low-traffic apps are caused by traffic filtering. Specifically, some traffic instances of these apps may contain only a few network flows, and FOAP may erroneously filter out these network flows if they are mixed with other apps' network flows. For example, when recognizing network traffic generated by the app `opofficial.pdfmaker`, FOAP suffers an FNR 0.136 as it erroneously filters out network flows in 3 traffic instances, each of which contains only one network flow. In another example, we recognize network traffic generated by the app `tw.com.trtc.is.androideng`. The FNR is 0.160, and FOAP erroneously filters out network flows in 2 traffic instances of this app, each of which contains 2 network flows.

• **Generalized Open-World Setting.** In a more generalized open-world setting, the adversary will handle both app traffic and non-app traffic. To further evaluate how FOAP performs in such an setting, we collect various non-app traffic including

i) Mobile website traffic: It contains 580,335 network flows that are generated by visiting Alexa top 10000 websites using the android browser.

ii) IoT traffic from PingPong dataset [43]: It contains 378,797 network flows that are captured from 19 IoT devices.

iii) PC traffic: It is captured from 6 PCs, including 2 windows PCs, 3 macOS PCs, and 1 Linux Ubuntu PC for 1 day. There are 16,801 network flows in total.

Table 2: False positive rate for open-world network traffic recognition (lower is better).

Open-World Traffic	AppScanner	AppScanner (extended)	FOAP (our method)
Mobile Web	2.41×10^{-2}	2.18×10^{-2}	$\mathbf{9.93 \times 10^{-4}}$
IoT	3.39×10^{-2}	2.85×10^{-2}	$\mathbf{4.36 \times 10^{-5}}$
PC	1.21×10^{-2}	2.02×10^{-2}	$\mathbf{3.54 \times 10^{-4}}$
SC-App	3.24×10^{-2}	3.09×10^{-2}	$\mathbf{1.45 \times 10^{-2}}$
DC-App	2.12×10^{-2}	1.99×10^{-2}	$\mathbf{2.64 \times 10^{-3}}$

All these non-app traffic will naturally be viewed as negative samples, because the goal of FOAP is recognizing the app of interest from network traffic. Therefore, we employ the metric false positive rate (FPR) to evaluate whether FOAP will generate false positives over these non-app traffic. Table 2 reports the experimental results. We can find the false positive rates of FOAP for all non-app traffic are extremely low and much lower than those of baseline methods. It again verifies FOAP's advantage in the open-world setting. Other than non-app traffic, Table 2 also reports the FPR for app traffic. Intuitively, apps from the similar category may share common network behaviors, potentially resulting in more false positives. To test this hypothesis, we group apps used for negative class in testing into two categories, SC-App and DC-App. SC-App refers to apps belonging to the same category with $\mathbb{A}$, while DC-App refers to apps whose categories differ from that of $\mathbb{A}$. As shown in Table 2, FPR for SC-App is higher than that for DC-App, indicating more false positives. This result is consistent with the above hypothesis.

7.3 EP Method Identification

We next evaluate FOAP in identifying EP methods.

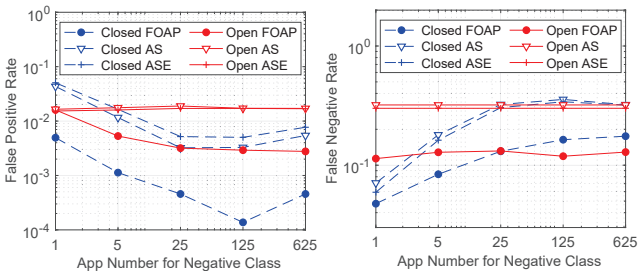
• **Experimental Setup.** EP method identification is an app-specific multi-label classification task. Therefore, FOAP establishes a model for each app. Similar to the previous setting, we randomly split traffic instance set of each app into two subsets: 40 instances for training and 10 instances for testing.

• **Result.** Multi-label classification can be viewed as the combination of multiple binary classification tasks. To evaluate the performance of FOAP in EP method identification, we consider a group of binary classification tasks corresponding to different EP methods. Specifically, we record the number of true positives, false positives, true negatives, and false negatives for each EP method and aggregate results of all EP methods in an app to compute the precision, recall, and F1-score. We employ micro-average method to compute these metrics due to the class imbalance in our problem.

Table 3: Evaluating FOAP in EP method identification (mean±standard deviation).

ε	Precision	Recall	F1-Score
0.5	0.888 ± 0.102	$\mathbf{0.883} \pm 0.105$	$\mathbf{0.885} \pm 0.101$
1	0.886 ± 0.104	0.868 ± 0.114	0.876 ± 0.106
2	0.885 ± 0.111	0.853 ± 0.126	0.866 ± 0.114
5	$\mathbf{0.892} \pm 0.105$	0.828 ± 0.137	0.856 ± 0.116

Table 3 reports the experimental results. FOAP achieves the highest average F1-score 0.885 when $\varepsilon = 0.5$ second. As the value of burst threshold ε increases, the precision of FOAP improves. The underlying reason is a larger ε results in more packets (therefore more information) involved in each in-flow burst, which helps reduce false positives. The highest average precision 0.892 is achieved when $\varepsilon = 5$ seconds. A side-effect of increasing ε is recall decline, which is caused by the majority voting mechanism used in burst labeling. We relax the multi-label classification problem to a series of multi-label classification subproblems at the cost of labeling a in-flow burst with only a single EP method, which implies a potential information loss of label. A larger ε will lead to more significant information loss and thus a lower recall. Consequently, the highest average recall 0.883 is achieved when $\varepsilon = 0.5$ second.



(a) False positive rate (FPR). (b) False negative rate (FNR).

Figure 4: Impact of experimental data size.

7.4 Impact of Experimental Data Size

To analyze how experimental data size influences the performance, we compare FOAP with baseline methods in both closed-world setting and open-world setting with varied app number in testing (i.e., $n_I = 1, 5, 25, 125, 625$).

• **Experimental Setup.** As for the closed-world setting, apps for model training are the same as those in testing. As for the open-world setting, apps that are considered as the negative class for model training and those in testing belong to two disjoint sets. We fix the number of apps that are considered as the negative class during model training to be $n_T = 20$ and change that in testing as $n_I = 1, 5, 25, 125, 625$.

• **Result.** We report the experimental results in Figure 4. Figure 4(a) presents a log-log plot about how FPR changes when n_I increases. In general, FOAP consistently outperforms baseline methods as it has lower FPR. In the open-world setting, unlike FPR for baseline methods, which tends to be constant, FPR of FOAP gradually decreases as n_I increases. The underlying reason is related to traffic filtering. When n_I is small, false positives caused by "noise", e.g., background network flows, within relevant traffic segments dominate FPR. When n_I is large, false positives are expected to be mainly caused by mis-classification of network flows from other apps. Thanks to the traffic filtering module of FOAP, most network flows from other apps are filtered out. Consequently, the FPR of FOAP decreases when n_I increases. In the closed-world setting, FPR of all methods first drops because the diversity of negative samples for model training increases with n_I. (Recall that $n_T = n_I$ in the closed-world setting.) A large value of n_T

results in extreme class imbalance, which is commonly recognized as a negative factor in machine learning. To mitigate the class imbalance, we employ the resampling strategy. Its side-effect is FPR for all methods rises again when $n_I > 125$. Figure 4(b) shows how FNR changes with n_I. In the open-world setting, FNR of all methods tends to be constant, while in the closed-world setting FNR increases with n_I, indicating more false negatives. It is caused by class imbalance during model training, despite a resampling strategy has been employed. Compared with baseline methods, FOAP shows advantage again in both settings.

7.5 Impact of Third-Party Libraries

The share of third-party libraries, such as advertisement library and third-party app authorization, among different apps potentially increases false positives, because the same third-party library may trigger similar network flows. We investigate this hypothesis in this experiment.

• **Experimental Setup.** We analyze how third-party libraries influence the performance of FOAP and baseline methods in the open-world setting. n_T and n_I are fixed to be 20. For each app, we first identify all TPLs used by it with the aid of Libradar [25]. Next, we correlate its EP methods with these TPLs. If the package name of an EP method matches a TPL, we regard this EP method as an EP method belonging to the TPL. Recall that we have labeled network flows with EP methods by leveraging method-level labeling. Combining the above two kinds of information, we can label a network flow with a TPL if any EP method belonging to this TPL triggers this network flow. Otherwise, we label it as a network flow triggered by the app-specific library (i.e., app class).

• **Result.** Table 4 reports FPR and FNR related with the top 5 third-party libraries in our dataset as well as the average performance for third-party libraries. As a comparison, we also present FPR and FNR related with app-specific libraries, which we refer to as *app class*. For all methods, the FPR related with third-party libraries is higher than that related with app class. Particularly, the FPR of FOAP in recognizing network flows associated with app class is 3.65×10^{-3}, while FPR rises to 5.80×10^{-3} for third-party libraries, which implies more false positives are generated. Another observation is third-party libraries do not increase false negatives. In fact, we even observe a slight FNR decline for FOAP. We conjecture a possible reason is when compared with app-specific libraries, third-party libraries may exhibit less diverse network behaviors because each of them is probably invoked for some fixed purposes, such as advertisement. To summarize, despite third-party libraries increase false positives, the FPR of FOAP is still reasonably low. In addition to third-party libraries, other factors may also influence the FPR of FOAP. We analyze the impact of automatic app generators in § G.

7.6 Cross-Dataset Evaluation

In this experiment, we evaluate to what extent FOAP trained based on a Monkey-generated dataset is able to recognize app traffic generated by humans.

• **Experimental Setup.** We randomly choose 100 out of 1000 apps to construct a dataset that reflects the true human-generated behaviors. Specifically, we recruit 5 volunteers to manually operate these apps and generate 5×100 traffic instances, each of which lasts for about 5 minutes. We compare three transfer settings. In the "M $\rightarrow$ M" setting, FOAP is trained over Monkey-generated dataset and tested over the same dataset. In the "M $\rightarrow$ H" setting, FOAP is trained over Monkey-generated dataset and tested over human-generated dataset. In the "EM $\rightarrow$ H" setting, the Monkey-generated dataset is enhanced by traffic instances generated by 4 volunteers and FOAP recognizes the traffic instance generated by the remaining volunteer. For statistical soundness, we rotate the volunteer for testing and report the average performance.

• **Result.** Table 5 reports experimental results for open-world app traffic recognition. Compared with "M $\rightarrow$ M" setting, the recall of FOAP in "M $\rightarrow$ H" setting reduces from 0.886 to 0.760, indicating some network flows generated by humans cannot be recognized and reported as false negatives. Nonetheless, it still outperforms baseline methods and even exhibits more significant advantage. Table 6 reports experimental results for EP method inference. Both precision and recall of FOAP exhibit a decline trend. We conjecture that the underlying reasons are two-fold. On one hand, Monkey test can hardly cover all UI operations that human may trigger in the wild. On the other hand, Monkey's behavior is highly random and thus UI operations triggered by monkey tend to be contextually independent. On contrary, UI operations triggered by humans are contextually correlated. Such a difference potentially results in different code paths and different network flows when the same UI operation (e.g., clicking a button) is triggered by Monkey and humans respectively.

To further explore how to improve the performance of FOAP, we enhance Monkey-generated dataset by involving human-generated traffic instances in the "EM $\rightarrow$ H" setting. Even though only 4 traffic instances are added to training dataset, the recall for open-world app traffic recognition is improved from 0.760 to 0.873. Likewise, both precision and recall for EP method inference rise. This experiment reveals a practical manner to construct training dataset when FOAP is applied in the wild. That is constructing FOAP's training dataset by combing traffic instances generated by both automatic test tools and humans, because i) it is less labor-intensive for automatic test tools to generate a large number of traffic instances, which empower FOAP with a high precision and fewer false positives, and ii) traffic instances generated by humans are beneficial to reducing false negatives and improving the recall of FOAP.

Table 4: Impact of third-party libraries.

Library	Coverage	AppScanner		AppScanner (extended)		FOAP (our approach)	
		FPR	FNR	FPR	FNR	FPR	FNR
App Class	100%	2.26×10^{-2}	3.10×10^{-1}	2.18×10^{-2}	3.04×10^{-1}	$\mathbf{3.65 \times 10^{-3}}$	$\mathbf{1.26 \times 10^{-1}}$
TPL (average performance)	86.2%	3.57×10^{-2}	3.30×10^{-1}	3.27×10^{-2}	3.29×10^{-1}	$\mathbf{5.80 \times 10^{-3}}$	$\mathbf{1.21 \times 10^{-1}}$
com.google.android.gms	65.4%	5.15×10^{-2}	2.75×10^{-1}	4.26×10^{-2}	3.48×10^{-1}	9.32×10^{-3}	2.20×10^{-1}
com.facebook	54.8%	3.17×10^{-2}	3.26×10^{-1}	3.13×10^{-2}	2.99×10^{-1}	3.99×10^{-3}	6.59×10^{-2}
com.google.firebase	23.0%	2.57×10^{-2}	4.43×10^{-1}	2.51×10^{-2}	4.15×10^{-1}	2.35×10^{-3}	9.99×10^{-2}
com.bumptech.glide	17.0%	3.85×10^{-2}	2.29×10^{-1}	2.18×10^{-2}	2.09×10^{-1}	4.53×10^{-3}	8.15×10^{-2}
com.urbanairship	3.9%	9.22×10^{-2}	3.93×10^{-1}	7.02×10^{-2}	3.49×10^{-1}	1.58×10^{-2}	2.81×10^{-2}

Table 5: Evaluating open-world app traffic recognition in cross-dataset experimental settings (mean±standard deviation).

Transfer Setting	AppScanner			AppScanner (extended)			FOAP (our approach)		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
M → M	0.625±0.167	0.775±0.120	0.681±0.135	0.634±0.148	0.700±0.136	0.655±0.127	**0.947**±0.074	**0.886**±0.157	**0.906**±0.114
M → H	0.506±0.222	0.521±0.195	0.481±0.185	0.436±0.212	0.508±0.201	0.439±0.181	**0.905**±0.163	**0.760**±0.219	**0.802**±0.194
EM → H	0.514±0.213	0.649±0.162	0.547±0.180	0.470±0.199	0.656±0.173	0.525±0.179	**0.929**±0.129	**0.873**±0.169	**0.882**±0.150

Table 6: Evaluating EP method identification in cross-dataset experimental settings (mean±standard deviation).

Transfer Setting	Precision	Recall	F1-Score
M → M	0.879±0.098	0.872±0.093	0.874±0.093
M → H	0.771±0.143	0.801±0.146	0.783±0.139
EM → H	0.830±0.145	0.834±0.150	0.831±0.143

7.7 Running Time of FOAP

In addition to accuracy, system efficiency also plays an important role. To this end, we evaluate the running time of FOAP by running it on a single core of a desktop equipped with an Intel Core i7-7700 CPU processor. FOAP takes an average of 1.77×10^{-2} second to recognize whether a network flow is from the app of interest. To further identify method-level user actions, FOAP spends another 5.91×10^{-2} second processing each network flow from the app of interest. The above experimental results indicate FOAP is highly efficient when applied in practice. Table 7 reports fine-grained running time for different modules of FOAP.

Table 7: Running time of FOAP.

Task	Major Steps	Running Time (second/flow)
Open-World App Traffic Recognition	Extract Flow Features	1.71×10^{-2}
	Traffic Segmentation	5.10×10^{-4}
	Traffic Filtering	3.61×10^{-6}
	Flow Recognition	7.73×10^{-5}
	Total	1.77×10^{-2}
Method-Level User Action Identification	Extract Burst Features	1.16×10^{-2}
	EP Method Inference	4.74×10^{-2}
	Total	5.91×10^{-2}

8 Application

We further demonstrate how FOAP can be applied in fine-grained user activity inference and user privacy analysis.

8.1 Fine-Grained User Activity Inference

A direct application of FOAP is inferring fine-grained semantic user activities from encrypted mobile traffic. We demonstrate this application through the case study of Twitter. To this end, we first map EP methods of Twitter to various user actions with the aid of UI Automator[2]. Specifically, we extract resource IDs of UI components that are associated with user actions listed in Table 8, (e.g., the button to share a tweet), using UI Automator and then compare them with the resource ID of EP methods obtained by FOAP's network flow labeling module to achieve a one-to-one mapping between user action and EP method. By doing so, we can reasonably infer user activities characterized by these user actions through identifying EP method invocation from encrypted mobile traffic.

We recruit two volunteers for this experiment. To construct the training dataset, the first volunteer operates Twitter to generate 20 traffic instances. The reason why we construct training dataset manually instead of using an automatic test tool (e.g., Monkey) is we consider many complicated user actions (e.g., retweet) in this case study and they are extremely difficult, if not entirely impossible, to trigger these operations with an automatic test tool. However, designing a smarter automatic test tool is out of the scope of this paper. To test how FOAP performs, the second volunteer is required to operate Twitter and meanwhile record his operations, i.e., user actions and timestamps, to establish the ground truth. Additionally, he also randomly chooses another 20 apps and operates them to test whether FOAP generates false positives.

FOAP spends about 40 seconds processing all 1706 network flows (31 TP, 1673 TN, 0 FP, 2 FN). Figure 5 illustrates the inference result. For ease of representation, we depict all network flows associated with user actions that have been successfully identified (denoted by F1, F2,..., F5) but omit other network flows even if they are from Twitter. FOAP successfully identify 12 user actions. It also makes some mistakes. Specifically, FOAP fails to report "search" at the 295th second but misreports it at the 352nd second. At the 317th second, it erroneously identifies "retweet" as "favorite". To summarize, FOAP achieves a high precision in recognizing network flows generated by Twitter. Besides, the accuracy for identifying fine-grained user actions is also reasonable.

[2]https://developer.android.com/training/testing/ui-automator

Table 8: EP methods for inferring Twitter user activities.

User Action	EP Method
Login	com.twitter.android.p1.onClick
Like	com.twitter.ui.tweet.inlineactions.InlineActionView$b.onAnimationEnd
Favorite	com.twitter.android.va$k.onClick
Retweet	xbb.onClick
Follow	com.twitter.app.profiles.y0.onClick
Notification	tcb.onClick
Send Message	com.twitter.app.dm.widget.DMConversationMessageComposer.onClick
Navigation	com.google.android.material.tabs.TabLayout$i.performClick
Search	km4.onClick

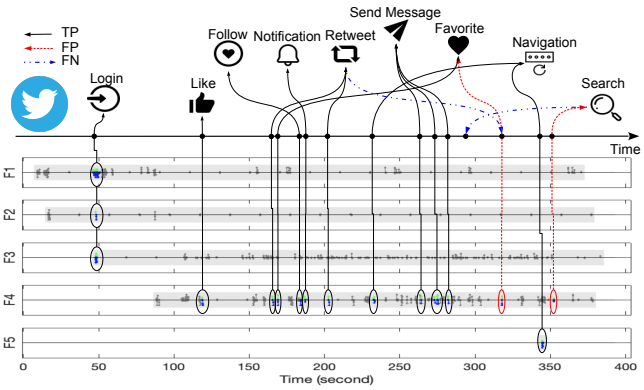


Figure 5: Inferring Twitter user activities.

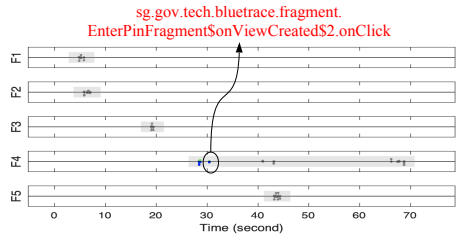


Figure 6: Identifying positive reporting in TraceTogether.

8.2 User Privacy Analysis

FOAP facilitates user privacy analysis by locating network flows associated with privacy-sensitive EP methods.

• **Sensitive Method Identification.** The invocation of sensitive EP methods in an app may leak out private user information, such as health status, sexual preference, and search queries. FOAP is able to identify them to infer user privacy.

The COVID-19 pandemic has rapidly spread across the world. To fight against the pandemic, contact tracing has been proven to be an effective strategy in monitoring virus spreading. To this end, lots of contact tracing apps are developed and published. Unfortunately, some of these apps may suffer the risk of privacy leakage. By analyzing the encrypted mobile traffic, one may infer if a user tests positive. Figure 6 presents an example about COVID-19 contact tracing apps, i.e., TraceTogether[3] The experimental setting is similar to that for Twitter. FOAP spends about 39 seconds in processing all 2250 network flows and recognizes 5 network flows generated by TraceTogether (no FP and FN). FOAP successfully identifies that the TraceTogether user (i.e., the second volunteer) clicks the button for positive test reporting, which invokes the EP method `sg.gov.tech.bluetrace.fragment.EnterPinFragment$onViewCreated$2.onClick`. FOAP captures this sensitive information by identifying in-flow burst associated with this method.

Table 9: Packet sequences associated with searching doctor of different hospital departments in ECEasyBook.

Department	Packet Sequence (Byte)
General Clinics	$\cdots 673\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1180}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$
Gynaecology	$\cdots 673\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1272}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$
Dermatology	$\cdots 673\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1266}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$
Chinese Medicine	$\cdots 674\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1306}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$
Paediatrics	$\cdots 673\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1238}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$
Geriatrics	$\cdots 673\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1292}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$
Psychiatry	$\cdots 674\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1247}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$

"↓" stands for inbound packets and "↑" stands for outbound packets.

• **In-Method Side-Channel Leaks.** FOAP also helps the analysis of in-method side-channel leaks, where diverse packet sequences triggered by the same EP method may leak sensitive private information. To demonstrate how the adversary may infer user privacy, e.g., the illness of the user, from in-method side-channel, we present an example about ECEasyBook[4], a clinic appointment app. In this app, the user searches doctors in different hospital departments will invoke the same EP method `com.ecbook.eceasy.ui.main.c.c$e.onClick` but trigger network flows with some differences in packet sequence. For example, the difference between searching doctors of dermatology and searching doctors of paediatrics occurs in the twelfth packet. The packet size for dermatology is 1266 bytes, while that for paediatrics is 1238 bytes. By leveraging this difference, the adversary may infer the illness of the user. We list packet sequences associated with searching doctor of different hospital departments and highlight the major differences in Table 9. Although FOAP did not directly infer user privacy in this example, it plays an important role in locating privacy-sensitive methods, enabling the adversary to zero in on network flows that leak out user privacy.

9 Discussion

9.1 Countermeasure

To defend against traffic analysis attacks, existing countermeasures aim to obfuscate traffic features using different strategies. There are three common obfuscation strategies: i) padding packets to obfuscate features related to packet sizes [24, 48], ii) injecting dummy packets to obfuscate features related to packet directions [7, 17], iii) delaying packets

[3] https://www.tracetogether.gov.sg/

[4] https://eceasybook.com/

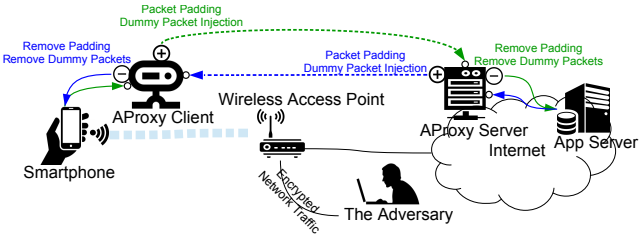
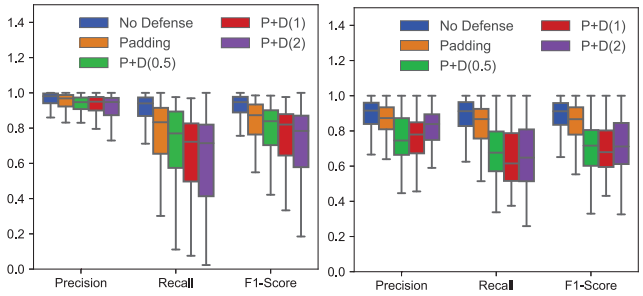


Figure 7: Design of our AProxy countermeasure.

to obfuscate features related to packet timing [11, 24]. Inspired by IMProxy [7], a proxy-based obfuscation system against traffic analysis attacks on messaging apps, we design an obfuscation system, named by AProxy, to implement traffic obfuscation in the form of proxy-based relayers and incorporate two obfuscation strategies that have been proven to be effective in defending against WF attacks, i.e., packet padding and injection of dummy packets. We do not consider the packet delaying strategy because it greatly degrades user experience if applied to apps. As illustrated in Figure 7, AProxy is comprised of an AProxy client and an AProxy server. The AProxy client is deployed on the Android smartphone while the AProxy server is a remote proxy server. Before relaying the outbound traffic to the AProxy server, the AProxy client first adds padding to the outbound packets so that all packet sizes equal 1500 bytes (i.e., MTU). It also injects outbound dummy packets with MTU size into each flow. The arrival time of dummy packets follows a Poisson process. After receiving the manipulated traffic, the AProxy server removes padding and dummy packets, and further relays it to the app server. As for the inbound traffic, the AProxy client and server switch roles. The AProxy server adds padding and injects inbound dummy packets while the AProxy client removes them and relays purified inbound packets to the app.

We experimentally evaluate the effectiveness of AProxy. Figure 8 reports the defense effect. "Padding" represents only packet padding is conducted and "P+D(x)" represents the dummy packet number is x times the original packet number. Generally, more dummy packets result in a worse performance of FOAP. For example, when the dummy packet number is twice the original packet number, the average F1-score for app traffic recognition drops from 0.911 to 0.714. A counterintuitive example is the precision for P+D(2) is higher than that for P+D(1) in EP method identification. A closer look reveals the precision decline is mainly caused by dummy bursts. Specifically, dummy packets may form dummy bursts, which are injected into original network flows and potentially undermine EP method identification as noises. FOAP will not distinguish these noises from original in-flow bursts and still predicts their labels, i.e., EP methods. Once the predicted EP methods are inconsistent with EP methods that exactly trigger the network flows where these noises are injected, false positives are generated. Therefore, more dummy bursts are expected to result in larger precision decline. At first glance,



(a) Open-world app traffic recognition. (b) EP method identification.

Figure 8: Evaluating AProxy against FOAP.

P+D(2) injects more dummy packets than P+D(1). However, we observed P+D(2) injects about 0.02 dummy burst per flow, while P+D(1) injects about 2.1 dummy bursts per flow. It is because denser dummy packets reduce inter-packet interval and tend to form bigger but fewer dummy bursts. Consequently, P+D(1) leads to a lower precision of FOAP than P+D(2).

9.2 Limitation

• **Separability of Flow.** Since FOAP recognizes apps by analyzing network flows, the prerequisite for doing so is different network flows are separable. In other words, packets from the same network flow can be grouped together to extract flow features. It implies FOAP's incompetence in the scenarios where network flows are inseparable, such as Tor.

• **Offline Recognition.** FOAP works in an offline fashion, because some procedures, e.g., traffic segmentation and EP method inference, are not stream-oriented. To enable quasi-real-time analysis, an immediate improvement is specifying sidling time windows and analyzing network traffic within each window independently. A larger window size potentially helps improve recognition accuracy, but introduces a higher computational overhead because more information will be involved in each time window. A smaller sliding step facilitates timeliness at the cost of more frequent analysis. Therefore, the proper window size and sliding step are chosen based on a trade-off between accuracy, timeliness, and overhead.

• **Model Integration.** To achieve fine-grained open-world app fingerprinting, FOAP needs to integrate multiple machine learning (ML) models. On one hand, these models are necessary to tackle various challenging issues in our problem. On the other hand, each model may need separate data science effort for model training and parameter tuning, which increases difficulty, albeit controllable, to apply FOAP in the wild. Specifically, for each app of interest, FOAP collects its traffic instances and constructs an app-level training dataset to train ML models for traffic segmentation, traffic filtering, and flow recognition. FOAP also needs to construct a method-level training dataset to train the spatial-temporal context model for EP method inference. Fortunately, FOAP features

the capability of automatic labeling of training data, thereby drastically minimizing the overhead of training data preparation. Additionally, different models may need separate effort for parameter fine-tuning. Nevertheless, our experiments demonstrate default parameters of these models are extensively suitable across different apps, hence alleviating the difficulty in parameter fine-tuning.

• **Limited Open-Worldness.** FOAP aims at open-world app traffic recognition. The number of available apps in the Google Play Store is about 3.5 million [37]. We evaluate FOAP using 1000 apps, around 0.29‰ of apps hosted in Google Play store. Such a percentage is comparable to those for open-world WF attacks in [34, 47]. There are around 1.9 billion websites in total [23], among which [47] considered 80,100 websites (around 0.04‰ of total websites) in the open-world evaluation, while [34] considered about 400 thousand websites (around 0.21‰ of total websites). Despite the number of apps used in this paper is substantially larger than that in most existing works [5, 14, 22, 30, 40], there is still a gap between our evaluation and realistic open world. We will collect more apps for evaluation in future work.

• **Synthesized Traffic.** To reduce the cost for data collection, we use traffic instances as negative samples when constructing the testing data for different apps. Consequently, we generate synthesized testing traffic by merging traffic instances from different apps and employing Poisson process to mimic a random use of apps. While Poisson process has been widely used to model human behaviors [13, 18, 32, 51], further investigation is needed to evaluate whether Poisson process can accurately characterize the use of apps. First, the use of apps depends on human dynamic, e.g., work and rest, and thus the use of an app may exhibit temporally heterogeneous property, which may not be well characterized by Poisson process. Second, the use of some apps may be temporally correlated for real human. However, we randomly arrange the order of different apps without considering their correlation. Therefore, the synthesized traffic resulting from merging traffic instances based on Poisson process might not exactly reproduce a real environment, and thus the performance of FOAP in the real world might be affected. We will further evaluate FOAP using non-synthetic traffic in future work.

10 Related Work

As an important branch of traffic analysis, traffic fingerprinting techniques aim to infer user behavior by leveraging statistic features of network traffic without accessing packet payload plaintext, thereby immune to traffic encryption.

App Fingerprinting. Our work falls into the category of app fingerprinting (AF), which focuses on the analysis of mobile network traffic. There are lots of efforts towards the identification of apps [2–4, 14, 40–42, 44]. For example, Taylor et al. proposed a modular framework AppScanner for automatic fingerprinting and real-time identification of Android apps from encrypted network traffic [40, 41]; Aceto et. al proposed a multimodal deep learning framework, dubbed MIMETIC, for mobile encrypted traffic classification to identify different apps [5]. To carry out more fine-grained characterization of user behaviors, some works [15, 22, 30] aim at the identification of selected user activities of interest. For example, Saltaformaggio et al. presented NetScope, a technique that utilizes traffic behavioral clues to automatically build a detector for smartphone app activities [30]; Fu et al. proposed CUMMA to classify service usages of mobile messaging apps by jointly modeling user behavioral patterns, network traffic characteristics, and temporal dependencies [15]; Liu et al. developed a recursive time continuity constrained KMeans clustering algorithm for traffic flow segmentation and classified the segmented traffic to identify in-app user activity [22].

Compared with existing AF techniques, FOAP is more fine-grained because we conduct method-level fingerprinting and infer EP methods associated with various UI components. By doing so, FOAP is able to not only identify fine-grained user activities but also infer sensitive private information (e.g., illness). Besides, the vast majority of existing AF techniques (e.g., [15, 22, 30, 40, 41]) work under the closed-world assumption. In this paper, we consider a more practical but challenging scenario, i.e., open-world setting. In literature, [9, 10, 44] also consider the open-world scenario. Unfortunately, their methods cannot solve our problem because they have different threat models. [44] took advantage of destination features while [9, 10] only handle unencrypted data.

Website Fingerprinting. Website fingerprinting (WF) focuses on inferring sensitive websites visited by users via encrypted proxies or anonymity networks, e.g., Tor. In the light of how to extract traffic features, existing WF techniques (e.g., [19, 28, 29, 31, 33, 34, 47, 49]) can be roughly categorized into the feature-engineering-based and the deep-learning-based. The former category [19, 28, 31, 47, 49] requires crafted features extracted using specific domain knowledge and employs a variety of machine learning algorithms, such as random forest, SVM, and KNN, to train classifiers for website identification, while the latter category needs either an enormous training set for automatic feature extraction [29, 33] or samples from a large number of similar tasks to conduct meta learning [34]. Due to different threat models, WF cannot be directly applied to our problem. Nevertheless, some of these works inspire us in traffic feature extraction. For example, we extract traffic features about interactive pattern inspired by [28]. In a recent work [47], Wang proposed three novel classes of precision optimizers to improve the precision of open-world website fingerprinting (WF) at the cost of recall decline. He also argued that it is more important to optimize the precision of WF attacks than their recall. Unfortunately, such a method is not applicable for our problem where precision and recall are comparably important.

11 Conclusion

In this paper, we took the first step to identify method-level fine-grained user action of Android apps in the open-world setting. To this end, a systematic solution dubbed FOAP was proposed. It features i) the capability of effectively reducing the false positive risk in open-world app recognition through adaptive traffic filtering and ii) method-level user action identification via synthesizing traffic and binary analysis. We have evaluated the effectiveness of FOAP through extensive experiments. FOAP significantly outperforms the baseline approaches by improving the F1-score from 0.679 to 0.911 for app recognition. It also achieves a reasonable accuracy in EP method identification. The average F1-score is up to 0.885. We also demonstrated the practicality of FOAP in fine-grained user activity inference and user privacy analysis.

Acknowledgements

We sincerely thank Prof. Andrea Continella for shepherding our paper and the anonymous reviewers for their constructive comments. This work is partly supported by Hong Kong RGC Projects (No. PolyU15223918, PolyU15227916) and National Science Foundation under Grant No. 1951729, 1953813, and 1953893.

References

[1] "Introduce network operations on a separate thread," https://developer.android.com/training/basics/network-ops/connecting, 2020.

[2] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "Traffic classification of mobile apps through multi-classification," in *Proc. GLOBECOM*, 2017.

[3] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "Mobile encrypted traffic classification using deep learning," in *Proc. TMA*, 2018.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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 *Proc. ACSAC*, 2017.

[10] 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 *Proc. SAC*, 2020.

[11] X. Cai, R. Nithyanand, T. Wang, R. Johnson, and I. Goldberg, "A systematic approach to developing and evaluating website fingerprinting defenses," in *Proc. CCS*, 2014.

[12] M. R. Chernick, W. González-Manteiga, R. M. Crujeiras, and E. B. Barrios, "Bootstrap methods," 2011.

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

[14] 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 *Proc. CODASPY*, 2015.

[15] 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.

[16] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *Proc. ICSE*, 2013.

[17] J. Gong and T. Wang, "Zero-delay lightweight defenses against website fingerprinting," in *Proc. USENIX Security*, 2020.

[18] A. Gupta and R. Sekar, "An approach for detecting self-propagating email using anomaly detection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2003, pp. 55–72.

[19] J. Hayes and G. Danezis, "k-fingerprinting: A robust scalable website fingerprinting technique," in *Proc. USENIX Security*, 2016.

[20] J. Kim, J.-h. Park, and S. Son, "The Abuser Inside Apps: Finding the Culprit Committing Mobile Ad Fraud," in *Proc. NDSS*, 2021.

[21] S. Kumar and M. Hebert, "Discriminative random fields," *International Journal of Computer Vision*, vol. 68, no. 2, pp. 179–201, 2006.

[22] 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 *Proc. SIGKDD*, 2017.

[23] I. live stats, "Total number of Websites," https://www.internetlivestats.com/total-number-of-websites/, 2021.

[24] 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 *Proc. NDSS*, vol. 11, 2011.

[25] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: fast and accurate detection of third-party libraries in android apps," in *Proc. ICSE*, 2016.

[26] W. Meng, R. Ding, S. P. Chung, S. Han, and W. Lee, "The price of free: Privacy leakage in personalized mobile in-apps ads." in *Proc. NDSS*, 2016.

[27] J. Niemeyer, F. Rottensteiner, and U. Soergel, "Contextual classification of lidar data and building object detection in urban areas," *ISPRS journal of photogrammetry and remote sensing*, vol. 87, pp. 152–165, 2014.

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

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

[30] 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 *Proc. WOOT*, 2016.

[31] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *Proc. USENIX Security*, 2019.

[32] 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.

[33] P. Sirinam, M. Imani, M. Juarez, and M. Wright, "Deep fingerprinting: Undermining website fingerprinting defenses with deep learning," in *Proc. CCS*, 2018.

[34] P. Sirinam, N. Mathews, M. S. Rahman, and M. Wright, "Triplet fingerprinting: More practical and portable website fingerprinting with n-shot learning," in *Proc. CCS*, 2019.

[35] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 465–488, 2017.

[36] 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/.

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

[38] Statista, "Number of smartphones sold to end users worldwide from 2007 to 2021," https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/.

[39] C. Sutton and A. McCallum, "An introduction to conditional random fields for relational learning," *Introduction to statistical relational learning*, vol. 2, pp. 93–128, 2006.

[40] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic," in *Proc. EuroS&P*, 2016.

[41] 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.

[42] M. Tian, P. Chang, Y. Sang, Y. Zhang, and S. Li, "Mobile application identification over https traffic based on multi-view features," in *Proc. ICT*, 2019.

[43] R. Trimananda, J. Varmarken, A. Markopoulou, and B. Demsky, "Packet-level signatures for smart home devices," in *Proc. NDSS*, 2020.

[44] 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 *Proc. NDSS*, 2020.

[45] H. Varian, "Bootstrap tutorial," *Mathematica Journal*, vol. 9, no. 4, pp. 768–775, 2005.

[46] O. Vinyals, C. Blundell, T. Lillicrap, D. Wierstra *et al.*, "Matching networks for one shot learning," in *Proc. NIPS*, 2016.

[47] T. Wang, "High precision open-world website fingerprinting," in *Proc. S&P*, 2020.

[48] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg, "Effective attacks and provable defenses for website fingerprinting," in *Proc. USENIX Security*, 2014.

[49] T. Wang and I. Goldberg, "On realistically attacking tor with website fingerprinting," *Proc. Priv. Enhancing Technol.*, vol. 2016, no. 4, pp. 21–36, 2016.

[50] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman, "Identifying diverse usage behaviors of smartphone apps," in *Proc. IMC*, 2011.

[51] C. C. Zou, D. Towsley, and W. Gong, "Email worm modeling and defense," in *Proc. ICCCN*. IEEE, 2004.

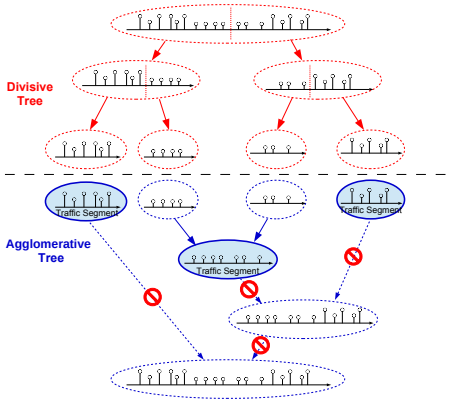


Figure 9: An example of divisive-agglomerative tree.

[52] O. Zungur, G. Stringhini, and M. Egele, "Libspector: Context-aware large-scale network traffic analysis of android applications," in *Proc. DSN*, 2020.

Appendices

A Divisive-Agglomerative Tree

A.1 Divisive Tree

As shown in Figure 9, we construct a divisive tree in a top-down fashion to greedily minimize the loss in (1) without considering the regularization term. Before the first splitting, there is only one traffic segment denoted by $\mathbf{s}_1 = (f_1, f_2, \ldots, f_M)$. In the tth splitting, without loss of generality, we assume that $\mathbf{s}_k$ starts at j. Splitting $\mathbf{s}_k$ at a will yield two sub-segments $\mathbf{s}'_k = (f_j, f_{j+1}, \ldots, f_{a-1})$ and $\mathbf{s}''_k = (f_a, f_{j+1}, \ldots, f_{j+|\mathbf{s}_k|-1})$, and the resulting loss reduction is computed by

$$\Delta D_k^a = \frac{(\sum_{i=j}^{j+|\mathbf{s}_k|-1} q_i)^2}{|\mathbf{s}_k|} - \frac{(\sum_{i=j}^{a-1} q_i)^2}{a-j} - \frac{(\sum_{i=a}^{j+|\mathbf{s}_k|-1} q_i)^2}{|\mathbf{s}_k|+j-a}, \quad (4)$$

subject to $j+n_{\min} \le a \le j+|\mathbf{s}_k|-n_{\min}$, $t_{a-1}-t_j \ge \tau_{\min}$, and $t_{j+|\mathbf{s}_k|-1}-t_a \ge \tau_{\min}$ according to constraints in (1). We therefore split $\mathbf{s}_{k^*}$ at a^*, where $(k^*, a^*) = \arg\min_{k,a} \Delta D_k^a$ is the optimal splitting point in the tth splitting. After each splitting, we reindex traffic segments. The divisive tree grows with an iterative splitting at the optimal splitting point until we cannot find a splitting point.

A.2 Agglomerative Tree

In contrast to divisive tree, the construction of agglomerative tree is from leaf to root. Leaves of agglomerative tree are those of divisive tree. Segments are merged to greedily minimize the loss in (1). Merging nodes results in a loss reduction because it reduces the complexity of segmentation, represented by the regularization term. In the tth round of merging, we assume that $\mathbf{s}_k$ starts at j. The merging of $\mathbf{s}_k$ and $\mathbf{s}_{k+1}$ will cause a loss change

$$\Delta M_k = \frac{(\sum_{i=j}^{j+|\mathbf{s}_i|-1} q_i)^2}{|\mathbf{s}_i|} + \frac{(\sum_{i=j+|\mathbf{s}_i|}^{j+|\mathbf{s}_i|+|\mathbf{s}_{i+1}|-1} q_i)^2}{|\mathbf{s}_{i+1}|} - \frac{(\sum_{i=j}^{j+|\mathbf{s}_i|+|\mathbf{s}_{i+1}|-1} q_i)^2}{|\mathbf{s}_i|+|\mathbf{s}_{i+1}|} - \lambda. \quad (5)$$

When finding the optimal merging $k^* = \arg\min_k \Delta M_k$, We merge $\mathbf{s}_{k^*}$ and $\mathbf{s}_{k^*+1}$. After each merging, we reindex traffic segments. The iterative merging stops if any merging will increase the loss, i.e. $\Delta M_k > 0$ for $\forall k$. As shown in Figure 9, segment merging marked by dotted arrow will not actually happen because they will increase the loss. The result of traffic segmentation are the remaining $\mathbf{s}_k$ for $\forall k$ when the agglomerative tree stops growing.

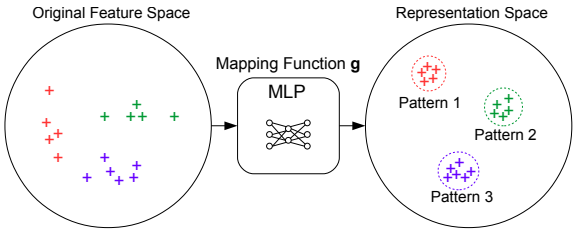


Figure 10: Representing feature vectors of network flows in representation space in favor of pattern mining.

B Metric Learning for Flow Pattern Mining

B.1 Flow Feature Mapping

In the representation space, we expect the euclidean distance between network flows in the same pattern will be minimized whereas that between network flows in different patterns will be maximized as shown in Figure 10. We approximate the mapping function $g(\cdot)$ using a multilayer perceptron (MLP), which is trained with the aid of dataset for very similar tasks, i.e., website fingerprinting [19, 28, 29, 31, 33, 34, 47, 49]. Specifically, we randomly select 1000 websites from Alexa top 1 million website list[5] and automatically visit each of them 10 times to capture network flows. We assume that i) network flows associated with the same remote endpoints exhibit one or more patterns and ii) if two network flows are associated with different endpoints they are expected to exhibit different patterns. Remote endpoints are identified by the server IP address and port. Taking into account the widespread adoption of load balancing mechanism by various network services, two endpoints with the same port but different IP addresses that however correspond to the same domain name are deemed to be the same endpoint.

[5] https://www.alexa.com/

Let $\Phi = \{\phi_i\}_{i=1}^{N_e}$ be the set of endpoints and $\mathcal{F} = \cup_{i=1}^{N_e} \mathcal{F}_i$ be the set comprised of all network flows, where $\mathcal{F}_i$ is a subset of $\mathcal{F}$ comprised of network flows associated with ϕ_i. The distance between a network flow f and $\mathcal{F}_i$ is defined by

$$D(f, \mathcal{F}_i) = \begin{cases} \min\limits_{f' \in \mathcal{F}_i / f} \|g(\mathbf{F}(f)) - g(\mathbf{F}(f'))\|_2, & f \in \mathcal{F}_i, \\ \min\limits_{f' \in \mathcal{F}_i} \|g(\mathbf{F}(f)) - g(\mathbf{F}(f'))\|_2, & f \notin \mathcal{F}_i, \end{cases} \tag{6}$$

where $\mathbf{F}(\cdot)$ is a function that returns network flow's feature vector. We train the MLP to approximate $g(\cdot)$ by minimizing the cross-entropy loss

$$L_g(\mathcal{F}) = \mathbb{E}_{\mathcal{F}_a, \mathcal{F}_b \sim P(\mathcal{F}_i), \mathcal{F}_a \neq \mathcal{F}_b} [\mathbb{E}_{f \sim P_{\mathcal{F}_a}(f)} [h(f, \mathcal{F}_a, \mathcal{F}_b)]], \tag{7}$$

where

$$h(f, \mathcal{F}_a, \mathcal{F}_b) = -\log \frac{e^{-D(f, \mathcal{F}_a)}}{e^{-D(f, \mathcal{F}_a)} + e^{-D(f, \mathcal{F}_b)}}. \tag{8}$$

In (7), $P(\mathcal{F}_i)$ is a discrete uniform distribution over all network flow subsets and the network flow f is drawn from $P_{\mathcal{F}_a}(f)$, which is a discrete uniform distribution over all network flows belonging to $\mathcal{F}_a$.

B.2 How Would Metric Learning Help?

Let $\mathcal{P}_a$ and $\mathcal{P}_b$ be two clusters. The distance between them is computed by

$$d_{ab} = \min \left\{ \|g(\mathbf{F}(f)) - g(\mathbf{F}(f'))\|_2 : f \in \mathcal{P}_a, f' \in \mathcal{P}_b \right\}. \tag{9}$$

Let δ be the clustering threshold. If $d_{ab} \leq \delta$, $\mathcal{P}_a$ and $\mathcal{P}_b$ will be merged as one. This "bottom-up" merging process repeats until the distance between any two clusters is larger than δ.

According to (9), we can derive the distance from a network flow f to a flow pattern $\mathcal{P}_k$ by

$$D(f, \mathcal{P}_k) = \begin{cases} \min\limits_{f' \in \mathcal{P}_k / f} \|g(\mathbf{F}(f)) - g(\mathbf{F}(f'))\|_2, & f \in \mathcal{P}_k, \\ \min\limits_{f' \in \mathcal{P}_k} \|g(\mathbf{F}(f)) - g(\mathbf{F}(f'))\|_2, & f \notin \mathcal{P}_k. \end{cases} \tag{10}$$

Without loss of generality, we assume that $f \in \mathcal{P}_i \subset \mathcal{F}_a$, where $\mathcal{P}_i$ is a flow pattern of $\mathcal{F}_a$. Equation (8) can be rewritten by

$$h(f, \mathcal{F}_a, \mathcal{F}_b) = -\log \frac{e^{-D(f, \mathcal{P}_i)}}{e^{-D(f, \mathcal{P}_i)} + e^{-\min_{\mathcal{P}_j \subset \mathcal{F}_b} D(f, \mathcal{P}_j)}}. \tag{11}$$

Recall that we optimize the mapping function $g(\cdot)$ by minimizing the cross-entropy loss $L_g(\mathcal{F})$. Equation (7) and (11) indicate that when $L_g(\mathcal{F})$ has been minimized, i.e., $L_g(\mathcal{F})$ approaching 0, $l(f, \mathcal{F}_a, \mathcal{F}_b)$ will approach 0 and thus we obtain

$$D(f, \mathcal{P}_i) \ll \min_{\mathcal{P}_j \subset \mathcal{F}_b} D(f, \mathcal{P}_j) \leq D(f, \mathcal{P}_j). \tag{12}$$

Given that f belongs to $\mathcal{P}_i$, it is easy to prove that $D(f, \mathcal{P}_i) \leq \delta < D(f, \mathcal{P}_j)$ for $\mathcal{P}_j \neq \mathcal{P}_i$. Inequality (12) reveals that representing network flows in the representation space determined by $g(\cdot)$ effectively facilitates the robustness of clustering analysis because the resulting clusters become not sensitive to the selection of δ.

C Details of Spatial-Temporal Context Model

C.1 Model Formulation

Let $\mathcal{M}_{\mathbb{A}} = \{m_k\}_{k=1}^{K}$ be a set comprised of EP methods of $\mathbb{A}$ that may trigger network flows. We denote by $\mathbf{b} = (b_1, b_2, \ldots, b_N)$ a sequence of in-flow bursts extracted from $\mathbb{A}$'s network flows and $\mathbf{y} = (y_1, y_2, \ldots, y_N) \in \mathcal{M}_{\mathbb{A}}^N$ the underlying EP method invocations that trigger these in-flow bursts. Given $\mathbf{b}$, the posterior distribution of $\mathbf{y}$ can be expressed in the following form

$$p(\mathbf{y}|\mathbf{b}) = \frac{1}{Z(\mathbf{b})} \prod_{i=1}^{N} \Omega_i(y_i, b_i) \prod_{i=1}^{N} \prod_{j \in \mathcal{C}_i^S} \Psi_{ij}(y_i, y_j, \mathbf{b}) \prod_{i=1}^{N} \prod_{j \in \mathcal{C}_i^T} \Phi_{ij}(y_i, y_j, \mathbf{b}), \tag{13}$$

where $\Omega_i(y_i, b_i)$ is a unary potential characterizing the relationship between y_i and b_i, $\Psi_{ij}(y_i, y_j, \mathbf{b})$ (resp. $\Phi_{ij}(y_i, y_j, \mathbf{b})$) is a spatially (resp. temporally) pairwise potential characterizing the relationship between b_i and its spatially (resp. temporally) contextual burst b_j, $\mathcal{C}_i^S$ (resp. $\mathcal{C}_i^T$) is a set comprised of all spatially (resp. temporally) contextual bursts of b_i, and $Z(\mathbf{b})$ is a normalizing constant. We consider unary potential and pairwise potential while ignoring higher-order dependency between contextual bursts to simplify model structure in favor of more efficient learning and inference and reduce the risk of overfitting. We assume that the posterior distribution $p(\mathbf{y}|\mathbf{b})$ follows the exponential family distribution due to its convenience for training and inference [39]. As such, unary potential and pairwise potential are defined as exponential family functions. Specifically, we compute $\Omega_i(y_i = m_a, b_i)$ by

$$\Omega_i(m_a, b_i) = \exp\{\alpha \cdot p_u(m_a|b_i)\}, \tag{14}$$

where α is a model parameter reflecting the weight of bursts' features and $p_u(m_a|b_i)$ is the probability of m_a triggering b_i. When formulating CRF, discriminative classifiers are often taken advantage of in the formulation of unary potential [21, 27]. In this paper, we train a random forest classifier RF_u to compute $p_u(m_a|b_i)$. As for the spatially pairwise potential, we compute $\Psi_{ij}(y_i = m_a, y_j = m_b, \mathbf{b})$ by

$$\Psi_{ij}(m_a, m_b, \mathbf{b}) = \exp\{\beta \cdot p_S(m_b|m_a) p_c(\langle m_a, m_b \rangle | c(b_i, b_j))\}, \tag{15}$$

where β is a model parameter reflecting the weight of spatially contextual dependency, $p_S(m_b|m_a)$ is the conditional probability that m_b triggers a *spatially* contextual burst of the in-flow burst which is triggered by m_a, and $p_c(\langle m_a, m_b \rangle | c(b_i, b_j))$ is the probability that m_a triggers b_i and m_b triggers b_j. Likewise, we compute temporally pairwise potential by

$$\Phi_{ij}(m_a, m_b, \mathbf{b}) = \exp\{\gamma \cdot p_T(m_b|m_a) p_c(\langle m_a, m_b \rangle | c(b_i, b_j))\}, \tag{16}$$

where γ is a model parameter reflecting the weight of temporally contextual dependency and $p_T(m_b|m_a)$ is the conditional probability that m_b triggers a *temporally* contextual burst of the in-flow burst which is triggered by m_a,

We empirically learn $p_S(m_b|m_a)$ and $p_T(m_b|m_a)$ from the training dataset, where EP methods associated with in-flow bursts can be observed directly. A larger value of $p_S(m_b|m_a)$ (resp. $p_T(m_b|m_a)$) indicates a closer spatially (resp. temporally) contextual dependency between the methods m_a and m_b and thus a bigger importance of the pairwise potential $\Psi_{ij}(m_a, m_b, \mathbf{b})$ (resp. $\Phi_{ij}(m_a, m_b, \mathbf{b})$) being considered when computing $p(\mathbf{y}|\mathbf{b})$. To compute $p_c(\langle m_a, m_b\rangle|c(b_i, b_j))$, we train another random forest classifier RF_c, where the feature vector $c(b_i, b_j)$ is constructed by concatenating feature vectors of b_i and b_j, i.e., $\mathbf{F}(b_i)$ and $\mathbf{F}(b_j)$, while the label is the method invocation pair.

C.2 Model Parameter Learning

The model parameters α, β, and γ balance the relative importance between local features (i.e., within-flow bursts themselves), spatially contextual features, and temporally contextual features. We optimize α, β, and γ in an app-specific fashion. To avoid the mutual interference, we decouple the training of random forest classifiers (i.e., RF_u and RF_c) and model parameter learning by leveraging bootstrap resampling method [12, 45], similar to the training of low-level classifier and high-level classifier in § 5.3.

We expect the the optimal model parameters can maximize the marginal probability of actual method invocation. From (13), we obtain the marginal probability of y_i

$$p(y_i = m_a|\mathbf{b}, \tilde{\mathbf{y}}_i; \alpha, \beta, \gamma) = \frac{\exp\{z_i^a(\alpha, \beta, \gamma)\}}{\sum_{k=1}^{K} \exp\{z_i^k(\alpha, \beta, \gamma)\}}, \tag{17}$$

where $\tilde{\mathbf{y}}_i = (y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_N)$ is a sequence consisting of all EP method invocations except y_i and

$$z_i^k(\alpha, \beta, \gamma) = \alpha p_u(m_k|b_i) + \beta \sum_{j \in C_i^S} p_S(y_j|m_k) p_c(\langle m_k, y_j\rangle|c(b_i, b_j)) + \gamma \sum_{j \in C_i^T} p_T(y_j|m_k) p_c(\langle m_k, y_j\rangle|c(b_i, b_j)).$$

We learn the optimal model parameter $\hat{\alpha}$, $\hat{\beta}$, and $\hat{\gamma}$ by minimizing the cross entropy loss over the training dataset

$$\min - \sum_{i=1}^{N} \sum_{k=1}^{K} \mathbf{1}\{y_i = m_k\} \log p(y_i = m_k|\mathbf{b}, \tilde{\mathbf{y}}_i; \alpha, \beta, \gamma). \tag{18}$$

D EP Method Inference Algorithms

We infer EP method in a greedy algorithms. We detail it in Algorithm 1. Its input includes a sequence of in-flow bursts $\mathbf{b} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N)$ and the maximum iteration number n_{greedy}. It outputs EP method associated with each in-flow burst.

Algorithm 1: Greedy EP method inference.

Input: $\mathbf{b} = (b_1, b_2, \ldots, b_N)$, maximum iteration number n_{greedy};
Output: EP method sequence $\hat{\mathbf{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_N)$;

Initialize a set $\mathcal{Y} \leftarrow \emptyset$;
$\mathbf{y}^{(0)} = (y_1^{(0)}, y_2^{(0)}, \ldots, y_N^{(0)})$, where $y_i^{(0)} = \arg\max_{m_k} p_u(m_k|b_i)$;
$p^{(0)} \leftarrow Z(\mathbf{b}) p(\mathbf{y}^{(0)}|\mathbf{b})$;
Add $\mathbf{y}^{(0)}$ to $\mathcal{Y}$;
for $k = 1, 2, \ldots, n_{greedy}$ **do**
  for $i = 1, 2, \ldots, N$ **do**
    $\tilde{\mathbf{y}}_i^{(k-1)} \leftarrow (y_1^{(k-1)}, \ldots, y_{i-1}^{(k-1)}, y_{i+1}^{(k-1)} \ldots, y_N^{(k-1)})$;
    Compute the conditional probability $p(y_i = m_a|\mathbf{b}, \tilde{\mathbf{y}}_i^{(k-1)}; \hat{\alpha}, \hat{\beta})$ for $\forall m_a \in \mathcal{M}_{\mathbb{A}}$ from (17);
    $y_i^{(k)} = \arg\max_{m_a} p(y_i = m_a|\mathbf{b}, \tilde{\mathbf{y}}_i^{(k-1)}; \hat{\alpha}, \hat{\beta})$;
  end
  $\mathbf{y}^{(k)} \leftarrow (y_1^{(k)}, y_2^{(k)}, \ldots, y_N^{(k)})$;
  $p^{(k)} \leftarrow Z(\mathbf{b}) p(\mathbf{y}^{(k)}|\mathbf{b})$;
  if $\langle \mathbf{y}^{(k)}, p^{(k)} \rangle \in \mathcal{Y}$ **then**
    break;
  else
    Add $\mathbf{y}^{(k)}$ to $\mathcal{Y}$;
  end
end
$\hat{\mathbf{y}} \leftarrow \arg\max_{\mathbf{y}^{(k)} \in \mathcal{Y}} p^{(k)}$;

E Hyperparameter Tuning of FOAP

We select the best hyperparameters based on grid searching to maximize the average F1-score. Table 10 summarizes the hyperparameter tuning process.

Table 10: Hyperparameter tuning.

Parameter	Search Space	Selected Value
$n_{\min}$	$\{2, 5, 10, 20, 50, 100\}$	10
$\tau_{\min}$	$\{2s, 5s, 10s, 20s, 50s\}$	$20s$
$m_{embedding}$	$\{2, 5, 10, 15, 20\}$	5
δ	$\{0.1, 0.2, \ldots, 1.9, 2\}$	0.5
$S_{\min}$	$\{0.05, 0.1, \ldots, 0.45, 0.5\}$	0.1
$S_{\max}$	$\{0.05, 0.1, \ldots, 0.55, 0.6\}$	0.25
T	$\{2s, 10s, 20s, 50s, 100s\}$	$20s$

F Effect of Different Modules

To analyze how different modules of FOAP work together in favor of performance promotion in the open-world setting, we present a more fine-grained analysis on the effects of different modules. We particularly focus on traffic filtering and bilevel recognition model, since they are identified as modules most relevant to false positives and false negatives through empirical observation. Figure 11 presents a boxplot that reports the performance of FOAP with/without these modules. Removing traffic filtering, FOAP-F suffers a decline of average precision from 0.945 to 0.714, indicating traffic filtering indeed represses false positives. Removing the bilevel recognition model, the average recall of FOAP-B drops from 0.897 to

0.748, indicating the bilevel recognition model effectively reduces false negatives. Removing both modules, FOAP-F-B achieves the lowest average F1-Score 0.779.

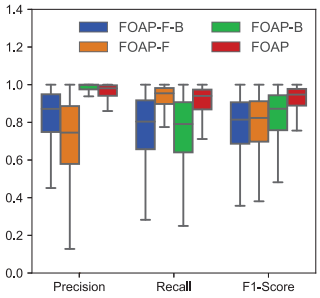


Figure 11: Effect of different modules of FOAP.

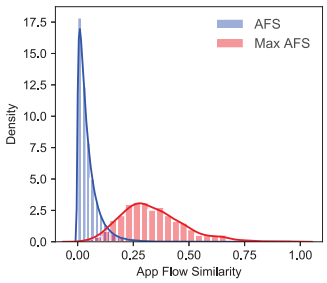


Figure 12: The distribution of app flow similarity (AFS).

G Impact of Automatic App Generator

Automatic app generators enable app development using wizard-like web interfaces to customize apps' UI without writing codes. As a result, apps developed on top of the same generators may exhibit very similar network behaviors. We investigate how FOAP performs in face of these apps.

• **Experimental Setup.** Because apps in our dataset are the most popular apps, few of them are developed on top of automatic app generators. To explore the impact of the automatic code generator, we generate 10×3 apps using three popular online automatic app generators, including AppYet, Mobincube, and AppsGeyser [6], to analyze their network behaviors. For each app, we employ Monkey to generate 50 traffic instances. FOAP still works in the open-world setting. For each automatically generated app, we choose $n_I = 9 + 10$ apps as negative class in the testing. These apps include other 9 apps generated by the same generator and 10 randomly chosen apps from our 1000 popular app dataset.

• **Result.** As shown in Table 11, The FPR of FOAP for apps generated by the same generator (aka. same family) is higher than that for other apps. It implies FOAP will generate more false positives when distinguishing network flows from apps generated by the same generator. To explore the root-cause why false positives increase, we propose a novel metric, named app flow similarity (AFS). For network flows generated by an app $\mathbb{X}$, we denote by $\bar{S}_l(\mathbb{X},\mathbb{A})$ their average local similarity with $\mathbb{A}$. The AFS between two apps $\mathbb{A}$ and $\mathbb{B}$ is defined by $\text{AFS}(\mathbb{A},\mathbb{B}) = \frac{\sqrt{\bar{S}_l(\mathbb{A},\mathbb{B})\cdot\bar{S}_l(\mathbb{B},\mathbb{A})}}{\sqrt{\bar{S}_l(\mathbb{A},\mathbb{A})\cdot\bar{S}_l(\mathbb{B},\mathbb{B})}}$. A smaller value of AFS between two apps indicates they are less similar in terms of network flows. AFS between an app and itself is 1. We measure pairwise AFS for apps in our dataset. Figure 12 shows an empirical distribution of AFS. The average AFS between two arbitrary apps is 0.046 and 99.3% values of AFS is smaller than 0.25, which implies two apps seldom exhibit exactly identical network behaviors. For an app, we also find its most similar app and record the maximum (MAX) AFS. The average MAX AFS is 0.336 and 99.1% values of MAX AFS is smaller than 0.75. As for apps generated by the same generator, we find their average AFSes are all above 0.6, which is significantly higher than that between two arbitrary apps and even MAX AFS. By manually checking values of AFS, we find some of them even approach 1, indicating it is hard to distinguish network flows from these apps.

Table 11: Impact of automatic app generators.

Automatic App Generator	FPR (same family)	FPR (other app)	Average AFS
AppYet	1.74×10^{-1}	5.30×10^{-3}	0.726
Mobincube	2.80×10^{-1}	1.99×10^{-3}	0.612
AppsGeyser	4.03×10^{-1}	1.07×10^{-3}	0.668

[6] AppYet: http://www.appyet.com/, Mobincube: https://mobincube.com/, AppsGeyser: https://appsgeyser.com/