IEEE TRANSACTIONS ON MOBILE COMPUTING

AdSherlock: Efficient and Deployable Click Fraud Detection for Mobile Applications

Chenhong Cao, Yi Gao, *Member, IEEE,* Yang Luo, Mingyuan Xia,Wei Dong, *Senior Member, IEEE,* Chun Chen, *Member, IEEE,* and Xue Liu, *Fellow, IEEE*

Abstract—Mobile advertising plays a vital role in the mobile app ecosystem. A major threat to the sustainability of this ecosystem is *click fraud*, i.e., ad clicks performed by malicious code or automatic bot problems. Existing click fraud detection approaches focus on analyzing the ad requests at the server side. However, such approaches may suffer from high false negatives since the detection can be easily circumvented, e.g., when the clicks are behind proxies or globally distributed. In this paper, we present AdSherlock, an *efficient* and *deployable* click fraud detection approach at the *client side* (inside the application) for mobile apps. AdSherlock splits the computation-intensive operations of click request identification into an offline procedure and an online procedure. In the offline procedure, AdSherlock generates both exact patterns and probabilistic patterns based on URL (Uniform Resource Locator) tokenization. These patterns are used in the online procedure for click request identification and further used for click fraud detection together with an ad request tree model. We implement a prototype of AdSherlock and evaluate its performance using real apps. The online detector is injected into the app executable archive through binary instrumentation. Results show that AdSherlock achieves higher click fraud detection accuracy compared with state of the art, with negligible runtime overhead.

Index Terms—Click fraud detection, mobile advertising, ad requests identification.

1 Introduction

Mobile advertising plays a vital role in the mobile app ecosystem. A recent report shows that mobile advertising expenditure worldwide is projected to reach $247.4 billion in 2020 [1]. To embed ads in an app, the app *developer* typically includes *ad libraries* provided by a third-party mobile *ad provider* such as `AdMob` [2]. When a mobile user is using the app, the embedded ad library fetches ad content from the network and displays ads to the user. The most common charging model is PPC (Pay-Per-Click) [3], where the developer and the ad provider get paid from the *advertiser* when a user clicks on the ad.

A major threat to the sustainability of this ecosystem is *click fraud* [4], i.e., clicks (i.e., touch events on mobile devices) on ads which are usually performed by malicious code programmatically or by automatic bot problems. There are many different click fraud tactics which can typically be characterized into two types: *in-app frauds* insert malicious code into the app to generate forged ad clicks; *bots-driven frauds* employ bot programs (e.g., a fraudulent application) to click on advertisements automatically. To quantify the in-app ad fraud in real apps, a recent work MAdFraud [5] conducts a large scale measurement about ad fraud in real-world apps. In a dataset including about 130K Android apps, MAdFraud reports that about 30% of apps make ad requests while running in the background. Focusing on bots-driven click fraud, another recent work uses an automated click generation tool ClickDroid [4] to empirically evaluate eight popular advertising networks by performing real click fraud attacks on them. Results [4] show that six advertising networks out of eight are vulnerable to these attacks.

Aiming at detecting click frauds in mobile apps, a straightforward approach is a threshold-based detection at the *server-side*. If an ad server is receiving a high number of clicks with the same device identifier (e.g., IP address) in a short period, these clicks can be considered as fraud. This straightforward approach, however, may suffer from high false negatives since the detection can be easily circumvented when the clicks are behind proxies or globally distributed. In the literature, there are also more sophisticated approaches [6], [7] focusing on detecting click frauds at the server-side. The precisions of these server-side approaches, however, are not sufficient enough for the click fraud problem. For example, in a recent mobile ad fraud competition [6], the best three approaches achieve only a precision of 46.15% to 51.55% using various machine learning techniques.

Given the insufficient precision of server-side approaches, a natural question comes up: how about client-side approaches? In fact, compared with the server-side approaches, it is easier to tell whether there is an actual user input at the client side. However, the attacker of the click fraud could be the app developers themselves, since the developers will get paid for those fraudulent ad clicks. Due to this conflict-of-interest problem, we cannot assume the existence of coordination from developers in designing a client-side approach for click fraud detection, e.g., a click fraud detection SDK. Therefore, in this paper, we focus on designing a *client-side* approach to detect click frauds in mobile apps, without coordination from developers.

There are two major challenges in designing such a system.

- *C. Cao, Y. Gao, Y. Luo, W. Dong, and C. Chen are with the College of Computer Science, Zhejiang University, Zhejiang 310027, China, and Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Zhejiang 310027, China. E-mail: {caoch,gaoyi,luoy,dongw,chunc}@zju.edu.cn.*
- *C. Cao is also with the School of Computer Engineering and Science, Shanghai Institute for Advanced Communication and Data Science, Shanghai University, Shanghai 200444, China. E-mail: caoch@shu.edu.cn.*
- *M. Xia and X. Liu are with School of Computer Science, McGill University, Montreal, Quebec H3A 0G4, Canada. E-mail: mingyuan.xia@mail.mcgill.ca,xueliu@cs.mcgill.ca.*

Digital Object Identifier 10.1109/TMC.2020.2966991

First, for a mobile client, its resources are constrained in terms of computation, memory, and energy. Therefore, the proposed approach must perform the complete fraud detection process *efficiently*, without causing significant overhead. This means that we need to design new algorithms to detect click frauds since existing machine-learning algorithms used by server-side approaches are not suitable for the client side. Second, the click fraud detection should be able to execute under practical user scenarios, instead of a controlled environment dedicated to fraud detection. In MAdFraud [5], a controlled environment (i.e., only one app is running and the HTTP requests are collected for offline analysis) is used to measure the ad fraud behavior of a vast number of apps. However, in our case, the click fraud detection should happen inside the mobile client without outside support, i.e., be *deployable* in real-world scenarios.

In this paper, we propose AdSherlock, an *efficient* and *deployable* click fraud detection approach for mobile apps at the *client side*. Note that as a client-side approach, AdSherlock is orthogonal to existing server-side approaches. AdSherlock is designed to be used by app stores to ensure a healthy mobile app ecosystem. AdSherlock's high accuracy helps market operators to fight both in-app frauds and bots-driven frauds. Note that, AdSherlock can also be used by any third parties to detect in-app frauds. For example, ad providers can employ AdSherlock to check whether apps embedding their libraries have in-app fraudulent behaviors.

To achieve these goals, AdSherlock relies on an accurate *offline pattern extractor* and a lightweight *online fraud detector*. AdSherlock works in two stages. At the first stage, the *offline pattern extractor* automatically executes each app and generates a set of traffic *patterns* for efficient ad request identification, i.e., extracts common token patterns across different ad requests. Specifically, after tokenization of the network requests, AdSherlock generates both exact patterns and probabilistic patterns for robust matching. Using the *offline pattern extractor*, AdSherlock can perform the computation and I/O intensive pattern generation operations in an offline manner, without degrading the online fraud detection operations. At the second stage, the *online fraud detector* as well as the generated patterns are instrumented into the app and run with the app in actual user scenarios. Inside the app, AdSherlock uses an *ad request tree* model to identify click requests accurately and efficiently. Since the online fraud detector runs inside the app, it can obtain the fine-grained user input events which are further employed for click fraud detection.

We implement AdSherlock and evaluate its performance using real apps. Results show that AdSherlock achieves higher click fraud detection accuracy compared with state of the art, with negligible runtime overhead. The contributions of this paper are summarized as follows:

- We present the design and implementation of AdSherlock, the first system which can achieve efficient and deployable click fraud detection at the client side.
- We propose a pattern generation mechanism that generates patterns for ad requests and non-ad requests with high accuracy. We also propose an efficient method for online click fraud detection based on an ad request tree model.
- We implement AdSherlock and compare its performance with the state-of-art approach. Results show that AdSherlock achieves higher detection accuracy with lower overhead.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 presents the overview of AdSherlock. Section 4 and Section 5 describe the offline and online components of AdSherlock in detail. Section 6 gives the implementation details of AdSherlock. Section 7 gives the evaluation of AdSherlock, and finally, Section 8 concludes this paper.

2 Related Work

2.1 Click Fraud Detection in Web Advertising

In the context of Web advertising, researches on click fraud detection mainly focus on bots-driven click frauds. These approaches are usually performed at the server-side, analyzing network traffic and characterizing the features of click fraud behaviors. [8] and [9] aggregate ad traffics across client IP address and cookie IDs to observe the client who has deviated ad traffic behaviors. SBotMiner [10] detects search engine bots by looking for anomalies in query distribution. However, such server-side approaches are not robust against sophisticated bots who can vary their IP addresses and other traffic features. Different from them, AdSherlock is a client-side method exploiting the property of click events on the end device which is hard to bypass. Moreover, these server-side methods need to collect sufficient ad traffics for analysis while AdSherlock does not need. From the client-side, AdSherlock can detect and prevent click fraud promptly.

Others works such as [11] and [12] focus on detecting duplicate clicks, where a publisher inflates its clicks by clicking on the same ad many times. These server-side methods can be viewed as a supplementary on AdSherlock in that they can detect click fraud performed by real humans.

FcFraud [13] is the latest work on click fraud detection in web advertising from the user side and is very related to our work. It identifies ad clicks and examines whether real mouse events accompany them. However, it needs to collect a bundle of HTTP requests for the ad request classifier which will cause unbearable overhead for Andriod apps. AdSherlock, on the other hand, focuses on click fraud detection in mobile applications.

2.2 Fraud Detection in Mobile Advertising

Recent years, several works are fighting on ad frauds in mobile advertising. MadFraud [5] studies the in-app fraud by executing apps in Android emulators to observe deviated behavior to detect ad frauds. DECAF [14] analyzes the UI of apps to discover *display fraud* such as small ads, hidden ads, intrusive ads, etc. However, both of them are investigated in a controlled environment and are hard to detect bots-driven click frauds. Different from them, AdSherlock is deployable in a production environment and performs click fraud detection in an online manner.

Another recent work aims at bots-driven click frauds is ClickDroid [15]. It develops an automated click generation tool to simulate attacker and detects frauds by distinguishing human-generated touch events from program-generated touch events. It needs the Android kernel to be modified to filter out program-generated touch events. AdSherlock does not assume any modification on the Android kernel and is a more general approach that proactively targets both in-app click frauds and bots-driven click frauds.

There also exists a hardware-assisted solution [16] for fraud detection in mobile advertising. AdAttester [16] provides proofs

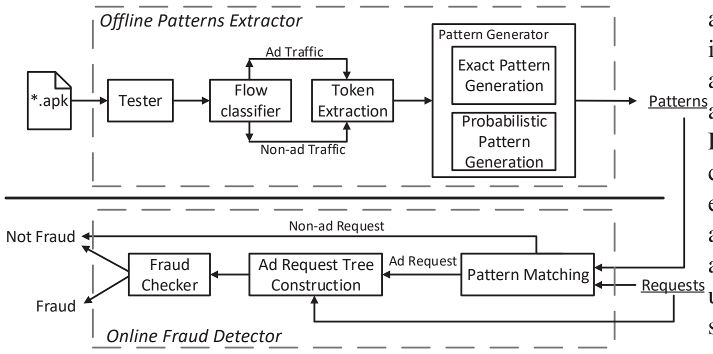


Fig. 1: Overview of AdSherlock.

of unforgeable clicks and verifiable display based on ARM's TruseZone. Different from AdAttester, AdSherlock does not need any hardware support.

3 AdSherlock Overview

AdSherlock is designed to be used by app stores. Before an app is released for download, the app store can use AdSherlock to analyze the app and instrument the online fraud detector into the app for click fraud detection at runtime. Only app binaries (e.g., APKs(Android application package)) are needed, and AdSherlock does not assume any developer input.

AdSherlock is mainly composed of two components: *offline pattern extractor* and *online fraud detector*. First, the offline pattern extractor takes the app as input and automatically executes the app to collect network traffics. Then, it classifies the traffics and extract traffic patterns for ad and non-ad traffics. Next, the online fraud detector is generated based on the extracted traffic patterns. The online fraud detector is responsible for network traffic monitoring, ad request identification, and click fraud detection. Finally, AdSherlock instruments the online fraud detector into the app binaries which are then released by the app store.

We show the main building blocks of AdSherlock in Fig 1. Each app is fed to the offline pattern extractor after uploading to the app store. This extractor automatically executes the app and generates traffic patterns for ad and non-ad traffics. These patterns are injected into the application together with the online fraud detector. When the app is running on the end user's device, the online fraud detector quickly checks every HTTP request with the offline generated patterns and identify the ad request. Next, the click request can be identified quickly by building an ad request tree. The abnormal click requests are detected by examining the related user input events. This operation is efficient. We mark a click request fraudulent if it does not accompany with any real user input events. For ease of understanding, we further introduce these two components in details.

Offline Pattern Extractor The offline pattern extractor automatically generates traffic patterns of ad and non-ad traffics for each app. To capture network traffics, we employ an automation tool to execute each mobile app binary automatically. The automation tool we build is called *Tester* using an Android testing tool, *monkeyrunner* [17].

Tester launches multiple instances of the Android emulator concurrently. Due to the scalability concerns, we do not drive the app along complete and exhaustive execution paths, which is time-consuming and not scalable. Instead, we allow incomplete app execution to ensure the scalability. For each app, it is executed in one emulator instance twice lasting 10 minutes, and the Tester automatically navigates through pages performing random clicks and swiping panels, etc. In practice, Monkeys have poor coverage. Particularly, when the app requires login passwords or have custom controls, Monkey may fail to interact with the app and explore less than ten pages.In such cases, we interact with Tester and manually operate on these apps to ensure that at least 20 or all pages are explored. This approach is enough because the ad usually repeats in many activities and the traffic flows have high structural similarity.

The captured network traffics then pass through a request classifier, which classifies network requests of the app into *potential ad traffics* and *non-ad traffics*. Classifying the HTTP requests has been studied in MadFraud [5]. It extracts features from the query parameters, the request trees and the HTTP headers. Then, it builds a RandomForestClassifier using these features. This current technique is not suitable for identifying ad requests at the client side in an online manner either because it is too slow or too costly. However, it is suitable to be used as a request classifier for AdSherlock in an offline manner. The design of request classifier is outside the scope of this paper, but we assume that the classifier we used for AdSherlock is imperfect. It may misclassify non-ad requests as potential ad traffics and vice versa. We refer to such misclassified requests as *noise*. It is challenging to generate patterns that cause low false positives with such noise.

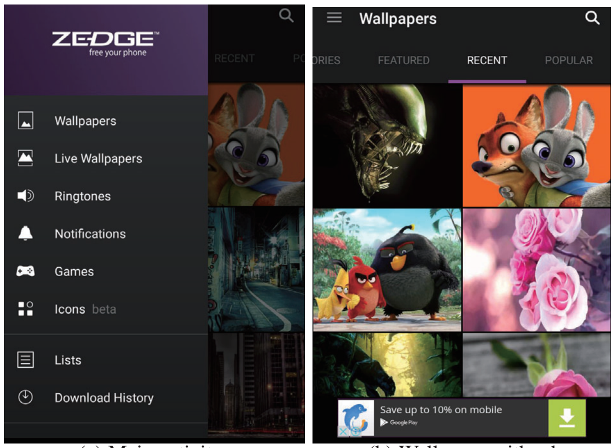
To solve this challenge, we generate two kinds of patterns: the *exact pattern* consists of a set of sequential substrings that occur in the HTTP header of the app's HTTP requests; the *probabilistic pattern* consists of a set of substrings, each of which associated with an ad score and a non-ad score. The details are described in Section 4.

Online Fraud Detector The online fraud detector identifies click requests and examines related user input events to detect click fraud. AdSherlock employs an *ad request tree* model to identify click requests accurately and efficiently. An ad request tree is a tree whose root is the ad request and other nodes are causally related HTTP requests. To build the tree, AdSherlock monitors the traffic through the *pattern matching* module and identifies ad requests using the offline-generated patterns. Then, ad requests are passed to the module of *ad request tree construction* to start building trees. Click can be identified from the ad request tree simply by finding the node of the request whose response header contains "location" field. Finally, the identified click request passes through the *fraud checker*. The checker evaluates related user input events to find out whether it is a fraudulent click. Details about the ad request tree construction and the fraud checker are given in Section 5.

4 Offline Pattern Generation

Generally, we divide network requests into two categories: ad traffic and non-ad traffic. Our goal is to extract traffic patterns of ad and non-ad traffics for each app. These extracted patterns are sets of substrings within the network traffic that can be used to distinguish these two kinds of traffics. In this section, we first give the key idea and challenges in automatically extracting traffic patterns. Then, we describe the pattern generation in details.

The key idea of extracting patterns is to find the *invariant* parts from network requests. To better illustrate the motivation



(a) Main activity (b) Wallpaper with ad

Fig. 2: The screen shots of Zedge.

```
GET /dl/wallpaper/9f1f4c3bc6930c04aa270c4160f0dbe8/
judy_and_nick.jpg?ref=android&type=mc&attachment=1 HTTP/1.1
Host: fsa.zedge.net

GET /dl/ringtone/222e5f3c1b7451b2ba8cf45dd3391af3/
lollipop.mp3?ref=android&type=mc&attachment=1 HTTP/1.1
Host: fsa.zedge.net

GET /dl/notification_sound/74c7e5d210dcaef6d1a3caa1bf66320b/
blackberry.mp3?ref=android&type=mc&attachment=1 HTTP/1.1
Host: fsb.zedge.net
```

(a) Non-ad traffics

```
GET /m/ad?v=6&id=3ee536462fd649aba0abd161bfadf256&...&fail=1
HTTP/1.1
Host: ads.mopub.com

GET /m/imp?appid=&cid=105ba158d8874b00868c3e48d06dbbe6&city=
Hangzhou&...&video_type= HTTP/1.1
Host: ads.mopub.com

GET /mads/gma?request_id=79835a07-ec6c-47c9-bc5b-5e79b7ba889e
&carrier=310260&...&url1=1499 HTTP/1.1
Host: googleads.g.doubleclick.net
```

(b) Ad traffics

Fig. 3: The network traffics of Zedge.

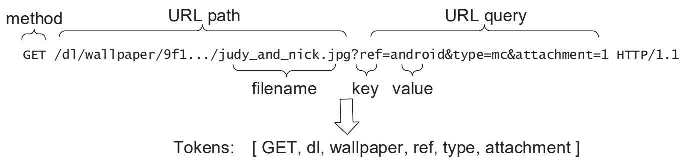

method
URL path
URL query
GET /dl/wallpaper/9f1.../judy_and_nick.jpg?ref=android&type=mc&attachment=1 HTTP/1.1
filename
key
value
Tokens: [GET, dl, wallpaper, ref, type, attachment]

Fig. 4: Token extraction example of Zedge request.

and the challenges, we consider Zedge [18], which is a top-rated app (more than 4M downloads) that is used for downloading wallpapers and ringtones. Zedge generates network requests for network behaviors such as loading an ad, previewing and downloading the wallpaper, etc. The typical user interactions with Zedge are as follows: a user start the app; then she clicks on the item of interest, for example, wallpapers (Fig. 2(a)); after that, a list of wallpapers are previewed for her, as well as an ad at the bottom of the page (Fig. 2(b)); then she can download a wallpaper or click on the ad of interest. Fig. 3 shows the ad traffics and non-ad traffics of these network behaviors. Here we only present the HTTP `GET` method and the URI of the HTTP requests for the ease of understanding. We envision that its principles can be applied elsewhere. The chief sources of invariant content we unearthed are fields in the HTTP header such as `Host`, `URL path` and the `URL query`. This is also true for HTTPS headers. Since we assume an in-app implementation of the online fraud checker, the HTTPS traffics can be intercepted before the encryption. For the simplicity of implementation, we only consider HTTP traffics in this paper. According to the study of Dai et al. [19], more than 70% of the apps do not use HTTPS. The ad traffics can also be CDN(Content Delivery Network) traffics since the ad provider usually provides CDN services to ensure the availability of ad content. The principle for extracting traffic patterns remains the same.

Our main objective for pattern generation is to generate high-quality patterns that offer low false positives for non-ad traffic and low false negatives for the ad traffics. There are three practical challenges for high-quality pattern generation:

Robust against multiple types of ad requests. As one app may contain more than one ad library, it may generate multiple types of ad requests. Different types of ad requests have significant differences from each other. For example, as shown in Fig. 3(b), Zedge makes ad requests to both MoPub [20] and Admob [2]. The longest invariant substring of the ad traffic is "GET" which causes 100% false positives in the non-ad traffic. Thus, a single pattern for ad traffics would be *too general* and cause high false positives or false negatives. Instead, we generate multiple patterns, each of which matches a subset of requests in the category. Multiple patterns altogether exhibit low false positives and low false negatives.

Robust to capture network requests that have not been seen in training yet. As described in the previous section, the network traffics we use for pattern generation are obtained by running the app in an emulator using an automation tool. There exists a circumstance where the user is taking a path that the automation tool does not explore. Our pattern generator should generate patterns that are general enough to capture such network behaviors.

Robust against noise. The inputs of the pattern generator consist of the ad traffic and non-ad traffic. However, there may be *noise* in these two categories due to the uncertainty of the classifier as described in the previous section.

To handle these challenges, we propose two pattern classes: *exact patterns* and *probabilistic patterns*. Both of them are built from invariant substrings in the HTTP header. We refer to these substrings as *tokens*. Exact patterns consist of a set of sequential tokens and match an HTTP request if and only if the request contains *all* tokens in the set with the same ordering. Probabilistic patterns consist of a set of tokens, each of which is associated with an *ad score*, and a *non-ad score*. We describe the details of pattern generation in the following sections.

4.1 Token Extraction

Token extraction is the first step to eliminate the irrelevant parts of the traffic flows which contain little information to identify ad requests. A *token* is a contiguous substring extracted from the content of an HTTP request that may stay invariant across multiple HTTP requests. After token extraction, we can use a set of tokens to represent each network flow.

To show how to extract tokens, we take a request from Fig. 3(a) as an example. The request is broken into various components. For the ease of illustration, we discuss how AdSherlock generates the URL tokens. The URL can be separated into two parts by the "?" delimiter. The former part is referred to as the URL path, and the latter part is referred to as the URL parameter. The URL path is further split into tokens using "/" as the delimiter. These tokens contain some page-components and filename [19]. The URL parameter is split into a set of key-value pairs. Only page-components and query-keys are considered to be used to build patterns. The extraction process is shown in Fig. 4.

4.2 Exact Pattern Generation

The exact pattern is made up of a list of ordered tokens. An HTTP request matches the pattern if and only if it contains all tokens in the pattern in the same order. This kind of patterns offers an exact matching rule.

4.2.1 Generating Single Pattern

We first describe the pattern generation for the simplest setting, i.e., an app only has one type of origin traffics that are sent to its own server, and makes requests to one ad provider. In this case, it only needs to generate a single pattern for each traffic category. Note that patterns generated here matching all (or most of) the potential ad traffic or the non-ad traffic are not resilient to noise or multiple ad providers.

To generate such sequential token pattern that matches every request in the pool, it simply needs to compute the intersection of request tokens in the same traffic.

4.2.2 Generating multiple Patterns

In practice, the ad traffic could contain more than one type of ad requests, as well as the non-ad traffic. We still want to generate patterns that have low false positives, i.e., match requests in the ad traffic and do not match non-ad traffic, and vice versa. In this case, AdSherlock generates multiple patterns, each of which matches a subset of requests in the category.

AdSherlock performs *clustering* on HTTP requests in each category by grouping requests having a similar structure. There are two demands for clustering: First, the clusters should not be too general. If the cluster mixes different types of ad requests or mixes the noise with the ad requests, the generated pattern would be too general that causes high false positives. The situation is similar to non-ad traffics. Second, the clusters should be too specific. Consider the extreme case when each cluster only contains one HTTP request. The pattern generated from the cluster will only match one request. The resulting pattern sets will be too large and cause high storage and computation overhead in online matching.

We employ *hierarchical clustering* [21] which is relatively efficient and does not need to know the number of clusters beforehand. We describe the procedure simply as follows. The inputs are request categories: the ad traffic category and the non-ad traffic category. For each category, we divide the requests into a set of clusters and generate a pattern for each cluster. In the beginning, we assume that there are n requests in the processing category. In the first step, they are clustered into n clusters; each contains one request. The second step is to merge these clusters iteratively. To determine which two of the clusters to merge, we generate patterns for each of the $\mathcal{O}(n^2)$ pairs of clusters and evaluate them in the other category. Two clusters are merged if the generated patterns have the lowest false positive rate. The next two clusters are chosen from the new merged one and the remaining $\mathcal{O}(n)$ clusters. Merging stops when the patterns generated by merging any two clusters would result in an unacceptably high false positive rate.

4.3 Probabilistic Pattern Generation

The exact patterns assume that the ad traffic and the non-ad traffic have distinct patterns of the request and will not change at runtime. However, there may have noise in each category, and the execution path may not be explored completely in an offline manner. In these cases, exact patterns cannot identify the request correctly. We propose another kind of pattern, *probabilistic pattern*, to solve the problem. The key insight of the probabilistic pattern is to exploit the different probability distributions of tokens presented in the ad traffic and the non-ad traffic to classify requests. Having the two different probability distributions, we can classify a request by determining which distribution its token set is more likely to be generated from. This type of patterns can be learned and are more resilient to noise or changes in traffic since it allows for probabilistic matching and classification.

We study this type of patterns from exploring how to classify requests with token distributions. Considering each token as a feature of the request, we find that the naive Bayes classifier is an appropriate choice. Compared to many other models, it needs far fewer examples and shows good performance when it is used with a vast number of dimensions (i.e., tokens in our case)[21]. The model has the following assumptions: the probability of a token presence in the ad traffic or the non-ad traffic is independent of the presence of other tokens in that traffic.

As described in section 4.1, after the first step token extraction, we obtain the token set of the specific app. We construct the request representation space $\mathbb{T} = [t_1, t_2, ..., t_n]$ including all tokens. An HTTP request of the app can be represented by a token vector in $\mathbb{T}$. Let R denote the HTTP request where $R = [R_1, R_2, ..., R_n], \forall 1 \leq i \leq n, R_i \in \{0, 1\}$ and the element R_i is set to 1 if the ith token t_i appears in the request.

Then we compute the empirical probability of a token appearing in each category. Let a_i denote the probability of token t_i appearing in the ad traffic and n_i the probability of token t_i appearing in the non-ad traffic. We compute the ratio of requests containing token t_i over the total number of requests in the ad traffic. Similarly, n_i is computed as the fraction of requests containing t_i in the non-ad traffic.

Let C denote request category where $C \in \{Ad, \neg Ad\}$. Given an HTTP request $\mathbf{r} = \{r_1, r_2, ..., r_n\}$, we need to compute $\mathbb{P}(C = Ad|R = r)$ and $\mathbb{P}(C = \neg Ad|R = r)$. The request is classified as an ad if:

$$\frac{\mathbb{P}(C = Ad|R = r)}{\mathbb{P}(C = \neg Ad|R = r)} \geq 1. \tag{1}$$

From Baye's rule,

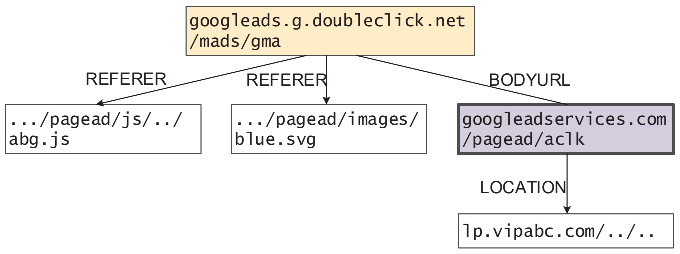
$$\mathbb{P}(C = Ad|R = r) = \frac{\mathbb{P}(R = r|C = Ad)\mathbb{P}(C = Ad)}{\mathbb{P}(R = r)}, \tag{2}$$

where according to the independence assumption:

$$\mathbb{P}(R = r|C = Ad) = \prod_{i=1}^{n} \mathbb{P}(R_i = r_i|C = Ad). \tag{3}$$

Hence, Equation 1 can be converted to:

$$\frac{\mathbb{P}(C = Ad) \prod_{i=1}^{n} \mathbb{P}(R_i = r_i|C = Ad)}{\mathbb{P}(C = \neg Ad) \prod_{i=1}^{n} \mathbb{P}(R_i = r_i|C = \neg Ad)} \geq 1. \tag{4}$$



googleads.g.doubleclick.net /mads/gma
REFERER
REFERER
BODYURL
.../pagead/js/../ abg.js
.../pagead/images/ blue.svg
googleadservices.com /pagead/aclk
LOCATION
lp.vipabc.com/../..

Fig. 5: A typical ad request tree of Admob.

It is hard to determine the value of $\mathbb{P}(C = Ad)$ and $\mathbb{P}(C = \neg Ad)$, i.e., the probability that any particular request is an ad request or not. We simply set them to be equal and convert the Equation 4 to its log term:

$$\sum_{i=1}^{n} \log \mathbb{P}(R_i = r_i | C = Ad) \geq \sum_{i=1}^{n} \log \mathbb{P}(R_i = r_i | C = \neg Ad). \tag{5}$$

Based on this equation, we assign each token an ad score and a non-ad score. An HTTP request is scored by adding the scores of tokens present in the request. If the resulting ad score is over the non-ad score, this HTTP request is classified to be an ad request.

5 Online Fraud Detection

5.1 Click Identification

We build ad request trees to identify click requests. For example, a browser loading a web page may fetch many other static resources, such as CSS, JavaScript or images, to embed in the HTML. In this case, a request tree, whose root is the request to the HTML page, can be built using the HTTP `referer` header.

The ad request tree construction starts when an ad request is identified. Each node represents a request and its corresponding response. For each request comes after the ad request, we add its corresponding node to the ad request tree if:

- The `referer` field of the latter request is set to be the request in the ad request tree. We consider the latter one as a child of the related request and mark the edge as “REFERER”.
- The `location` header in the response of the request in the tree is set along with a redirection status code to redirect the client to another URL. We consider the original request URL as the parent of the redirected URL and mark the edge as “LOCATION”.
- The latter request URL is in the response body of the ad request. We consider the latter request as a child of the ad request and mark the edge as “BODYURL”.

When a user clicks on an ad, the application generates an HTTP request to the ad network. The ad network then redirects the user to the advertiser’s page. The address of the advertiser’s page is typically provided in the `location` header of the response. Therefore, in our ad request trees, we mark a node as an ad click if its edge link to the child is marked as “LOCATION” and the child node represents a third-party website. Fig. 5 shows an example of a typical ad request tree for Admob. The node “`googleadservices.com/pagead/aclk`” is marked as an ad click.

The ad request tree construction stops when an ad click is identified. Then, the click request is passed to the fraud checker for further process.

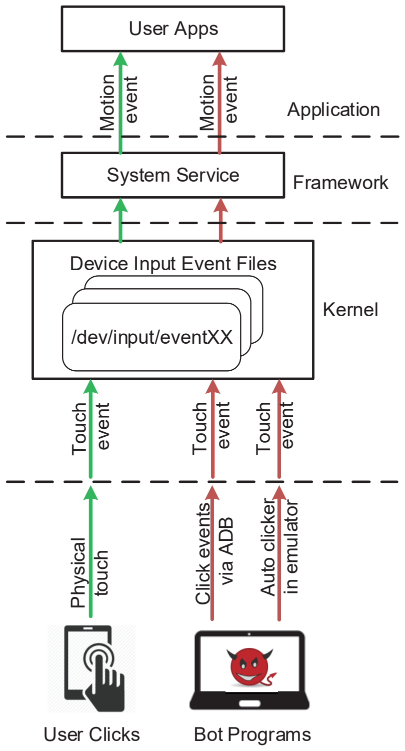


Fig. 6: The event flow of input events generated by human and non-human clicks.

5.2 Fraud Checker

When an ad click is identified, AdSherlock invokes the fraud checker. The checker exploits the information of fine-grained user input events to deviate human and non-human clicks.

To obtain the input events, we investigate the event flow generated both by human clicks and non-human clicks. We further divide the non-human clicks into bots-driven fraudulent clicks and in-app fraudulent clicks according to the fraud tactics. Since in-app fraudulent clicks do not generate any motion events and are easy to be identified, we focus on bots-driven fraudulent clicks. When the user clicks on the screen, touch events are generated and delivered to apps as motion events. However, such input events can be faked by bot programs. As illustrated in Fig 6, green lines show the event flow of human clicks while red lines show that of faked clicks. Bot programs fake human clicks by programmatically generating touch events in two ways: emulator-based fraudulence and ADB-based fraudulence. In emulator-based fraudulence, a bot program can automatically click on apps in emulators via auto clickers on PC. In ADB-based fraudulence, bot programs can send `sendevent` commands to the Android app via Android Debug Bridge (ADB). Since the ADB-based fraudulence requires root permission and is hard to perform in practice, AdSherlock focuses on detecting emulator-based fraudulence which is a widely used tactic. At the application level, both human clicks and fraudulent clicks are transferred to the app as motion events. We observe that these motion events contain information that can help us deviate human clicks and non-human clicks inside an app. In this section, we carefully analyze the useful properties contained in the motion event. We then indicate how to distinguish human clicks and non-human clicks using these properties.

Selected properties of the motion event. When the user clicks on the ad, a set of motion events are triggered and

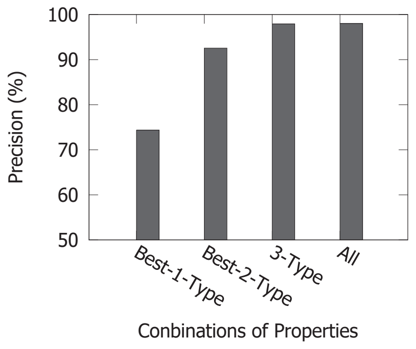


Fig. 7: The accuracy of different combinations of properties.

delivered to the application. The `MotionEvent` object contains movement data that is defined in terms of its action code along with a set of axis values and records a set of other movement properties [22]. Among those properties, we observe that three types of properties can be used to distinguish human clicks and non-human clicks. These properties are the pressure of the finger (`AXIS_PRESSURE`), size of the contact area (`AXIS_SIZE`, `AXIS_TOUCH_MAJOR`, `AXIS_TOUCH_MINOR`) and touch position (`AXIS_X`, `AXIS_Y`). We decide which combination of these properties to be used by evaluating their accuracy, which is in terms of Precision= $\frac{TP}{TP+FP}$ (TP represents true positives and FP false positives). We collect 1000 human clicks from 7 different real devices and 2000 non-human clicks from Android emulators. The goal is to select a minimum number of properties of the motion event that can accurately distinguish between human and non-human clicks. We combine different properties to find the right set of properties. As shown in Fig. 7, the precision is nearly 98% when using three types of properties mentioned. The precision is relatively lower if we only take only one or two types of them. The figure also shows that other properties of the motion event have little contribution to precision.

Bots-driven fraudulent clicks. As described above, bot programs can fake user clicks in two ways. The emulator-based fraudulence can be easily detected based on the observation that the values of those selected properties are inconsistent between human clicks while remaining the same for bots-driven clicks. It is straight forward to check changes in the selected properties to identify who is clicking. However, it is hard to detect the ADB-based fraudulence since it can modify the motion event properties via ADB. Since it requires root permission of the device and is hard to perform in practice, AdSherlock focuses on detecting emulator-based fraudulence which is a widely used tactic.

In-app fraudulent clicks. The in-app clicks do not generate any motion events and are easy to be identified if there is no other user interaction with the app. However, the situation becomes complicated if the user is interacting with the app at the same time. For example, a user is playing a game in the “front”, while the malicious code inside the app is sneakingly clicking on the invisible ad in the “back”. In this situation, the motion events generated by the user clicks are considered as noise and hinder the detection of in-app fraudulent clicks. To eliminate the noise, AdSherlock checks the click position to see whether it falls in the ad area. We obtain the area of the ad control using DECAF [14].

6 Implementation

We have implemented a prototype of AdSherlock. The offline pattern extractor is implemented in Python and runs on Ubuntu 14.04 equipped with 3.30GHz quad-core CPU and 12GB memory. The online fraud detector is implemented within a simple Android application, targeting Android API level 19 and running on a Nexus 5 device equipped 2.26 GHz quad-core and 2GB memory.

The online fraud detector is injected into the application archive through binary instrumentation. It intercepts the network traffic at runtime and logs the user touchscreen input events into the buffer. The network traffic is then fed into the pattern matching part to identify ad requests. The touchscreen input events, i.e., motion events, are used by the fraud checker to detect click frauds.

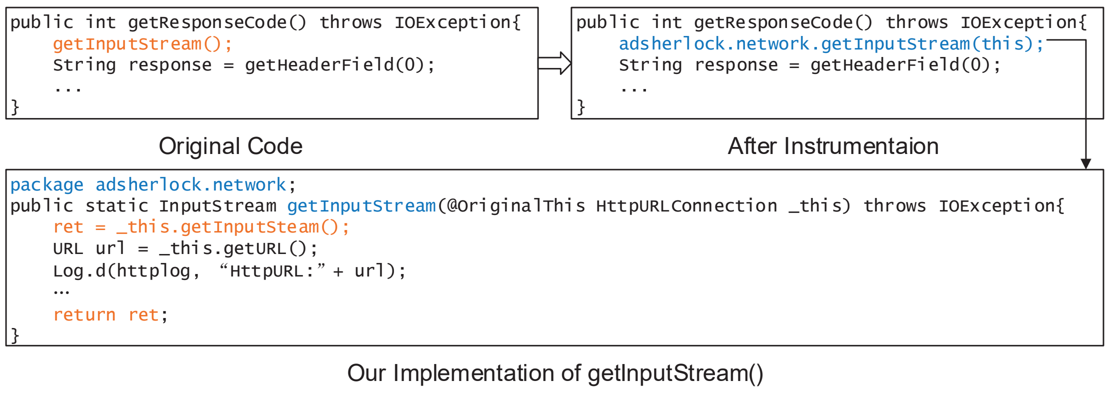
Binary instrumentation. The app executable archive for a given mobile application contains, amongst other things, the application binary (classes.dex) and a metadata file. In order to inject the online fraud detector into the application, we propose to decompile the application and inject a small patch into the bytecode before repackaging the application. First, the original application is disassembled by using the baksmali tool to obtain a human-readable Smali bytecode from the dex file. Then, the Smali code is annalized to find APIs calling the HTTP library. Hooks are injected to intercept the network traffics and redirect them to the fraud detector. Finally, the Smali code is assembled after modification and the application is repackaged. At run time, the injected fraud detector reports click frauds to the trusted third party, e.g., the app market.

Fig. 8 shows an example of the hook operation, which is written in Java for the ease of understanding. Note that, these changes are performed on the binary code in practice. We hook the `getInputStream` function in class `java.net.HttpURLConnection` to our implemented one, `adsherlock.network.getInputStream`. Our implementation calls the original function and writes the information of interests into logs.

Note that the developer signature key of the app will be lost after disassembling and reassembling. We assume that app developers trust the app market and build partnerships with it. Therefore, we can re-sign the app and embed a new signature key to the reassembled app.

Network traffic interception. Since the HTTP request can be generated from the native application code and the Webkit WebView, the traffic interception is performed through hooking the connection related methods in these libraries. For example in Fig. 8, we inject codes to hook the `connect` method in `HttpURLConnection` and the `execute` methods in `Apache HttpClient`. These two libraries are native network libraries provided by Android. According to [23], nearly 90% of the top 1000 popular Android apps from Google Play Store use these two native libraries. Other most widely used mobile network libraries, such as Volley and Android Asynchronous HTTP client [23], rely on these two libraries.

Motion events collection. Motion events are the input of the fraud checker. To collect motion events from the running app, we need to understand the data flow of motion events. As shown in Fig. 9, the Android system service first passes the motion event to the Android framework. Then, the motion event is delivered to the running application. In the application, the `Activity.dispatchTouchEvent()` method is always the first to be called. It is responsible for routing motion events to where they should go. Thus, we hook the `dispatchTouchEvent` method inside the app to record the motion event.



```
public int getResponseCode() throws IOException{
    getInputStream();
    String response = getHeaderField(0);
    ...
}
```

```
public int getResponseCode() throws IOException{
    adsherlock.network.getInputStream(this);
    String response = getHeaderField(0);
    ...
}
```

```
package adsherlock.network;
public static InputStream getInputStream(@OriginalThis HttpURLConnection _this) throws IOException{
    ret = _this.getInputSteam();
    URL url = _this.getURL();
    Log.d(httplog, “HttpURL:” + url);
    …
    return ret;
}
```

Fig. 8: An example of hook operation.

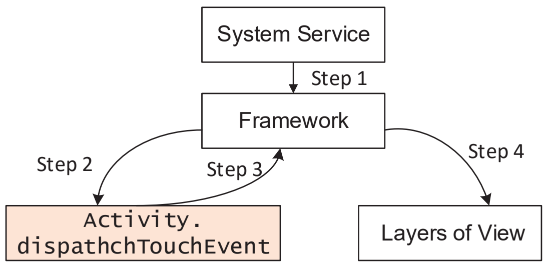


Fig. 9: The data flow of motion events in Android application.

7 Evaluation

In this section, we evaluate the overall performance of AdSherlock for a variety of apps in two real-world scenarios. We implement two fraudulent programs that automatically generate fraudulent clicks and use AdSherlock to detect them. We observe that the trickiest part of AdSherlock is the *ad request identification* since the detection accuracy of AdSherloack mainly depends on the accuracy of ad request identification. Thus, we focus on evaluating the performance of ad request identification and compare AdSherlock with the state-of-art approach, MadFraud [5]. Then, we present micro-benchmarks to quantify AdSherlock's overhead.

7.1 Experimental Setup

Traffic dataset. We collected a total number of 18,606 apps from Google Play during Nov. 2017. We then perform static analysis on those apps across tens of app categories to select apps with embedded ad libraries. About 61.3% of apps contain ads and most of them are from popular categories include Entertainment, Personalization, Music & Audio and Casual. From them, we then select 1750 free apps without the login requirement and making HTTP requests to at least one known ad provider. For each app, we run it in our Tester and collect their network traffics. To build the ground truth dataset for identifying ad requests, we manually investigate the requested pages of each app and use the most popular ad libraries as our references. We proceed from these ad pages to further identify ad requests. Overall, we labeled 16,751 ad requests from 230,626 instances of traffic flows.

Real-world scenarios. To evaluate the performance of click fraud detection, we implement two kinds of fraudulent programs to generate fraudulent traffics automatically in two different real-world scenarios.

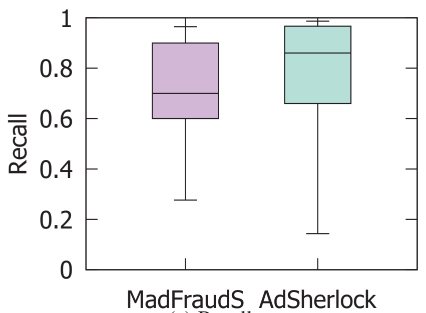
1) *Bot-driven fraudulent scenario*. In this scenario, a bot program automatically clicks on apps in emulators. We carefully select 16 apps and use AdSherlock to generate request patterns as well as injecting the online fraud detector into them. The selected apps are listed in Table 1. All apps are downloaded from Google Play and have more than 1M downloads. Then, we install the selected apps in an Android emulator and use the bot program to click on the app pages as well as the embedded ads. The generated network traces, as well as the motion event traces, are collected as the evaluation dataset. Besides, we also include traces generated by human clicks in the dataset. We install the selected apps on real devices and employ volunteers to execute them. In total, we collect 2717 ad requests from the 23078 HTTP requests as the ground truth for emulator requests, and 1206 ad requests from 11490 HTTP requests for human requests.

2) *In-app fraudulent scenario*. We implement a simple application which includes an ad library to display ads. This simple application use performClick method to generate click events for ads displayed automatically. We install the app on 2 real devices and employ volunteers to interact with it. The Android devices we implement AdSherlock are Google Nexus 5, which has quad-core QUALCOMM processors running at 2.26 GHz. The phone runs Android version 4.4 with Linux kernel version 3.10. The automatic click interval of the fraudulent program is set to be 3 seconds. We let the volunteers interact with the app to simulate the real scenario. We vary the interval between human interactions to study the performance of different strategies.

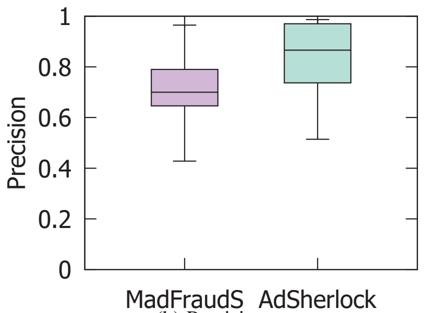
Baseline strategy. We select and implement MadFraud as our baseline since it is the state-of-art work on ad request classification. We implement a simplified version of the ad request classifier in MadFraud and refer to it as MadFraudS. MadFraudS is built on the top 16 features ranked by the importance in [5]. We combine it with our fraud checker to identify click frauds at runtime.

7.2 Evaluation Metrics

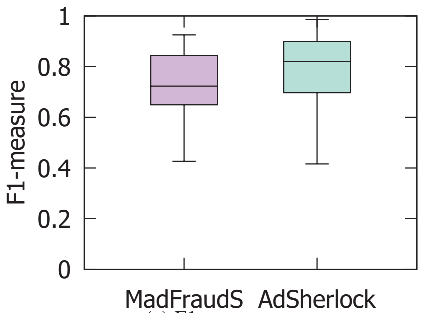
We evaluate our approach against standard metrics for evaluating binary classifiers –recall(true positive rate), precision, and F1–measure. F1–measure represents the overall performance since it is a combined metric (weighted harmonic mean) of precision and recall. We first denote TP , FP , TN and FN as true positive, false positive, true negative and false negative, respectively. A true positive (TP) is when both AdSherlock and ground-truth flag a request as an ad request; a true negative (TN)



(a) Recall.



(b) Precision.



(c) F1-measure.

Fig. 10: Performance of ad request classification.

TABLE 1: The selected apps and their categories.

App Name	Category
Zedge	Personalization
iTranslate	Business
Flashlight	Tools
4Pics1Word	Word
Horoscope	Lifestyle
Funshion	Media & Video
VivaReader	News & Magazines
Budejie	Entertainment
Lingshengduoduo (LSDD)	Music
Brixd	Personalization
theScore	Sport
WeatherBug	Utilities
Water Drink Reminder (WDR)	Health
Kamitu	Game
ColorTouch	Photo
Crossword	Entertainment

is similarly when both flag it as a non-ad request. A false positive (FP) is when AdSherlock flags a request as an ad request while the ground-truth does not, and vice-versa for false negative (FN). Then, the computation formulas for the metrics are as follows: Recall= $\frac{TP}{TP+FN}$, Precision= $\frac{TP}{TP+FP}$, F1–measure= $\frac{2*Precision*Recall}{Precision+Recall}$. Similarly, we also use these three metrics to evaluate the performance of click fraud detection.

7.3 Main Results

Ad request identification performance. We evaluate the ad request identification performance of AdSherlock on the traffic dataset and compare that with MadFraudS. We split the ground-truth dataset into three folds for each app, train classifiers on 2 of those folds and then evaluate its performance on the remaining fold. Since MadFraudS [5] adopts three-fold cross-validation in the ad request identification evaluation, we also use three-fold cross-validation to make a fair comparison with MadFraudS. Moreover, to make the results more representative, we randomly split the ground-truth dataset into three folds. Note that, for the click fraud detection evaluation in the next, we apply a similar strategy to randomly split the dataset into three folds and employ the three-fold cross-validation method for the sake of consistency. As shown in Fig. 10, we can see that both AdSherlock and MadFraudS can achieve a high recall. The F1-measure of AdSherlock is obviously higher than MadFraudS. This is because AdSherlock achieves higher precision than MadFraudS. Specifically, this is because MadFraudS has relatively higher false positives due to the misclassification of analytic traffics. In comparison, AdSherlock is more robust in such a case with the help of multiple patterns.

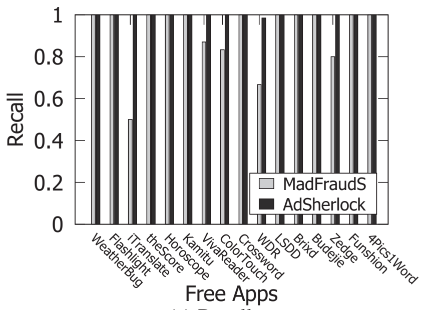
TABLE 2: Statistical results of click fraud detection.

Metrics	MadFraudS		AdSherlock	
	Average(%)	Stand.(%)	Average(%)	Stand.(%)
Recall	**91.69**	**14.47**	**99.90**	**3.08**
Precision	**71.69**	**15.22**	**96.54**	**4.96**
F1-measure	**79.92**	**12.74**	**98.13**	**2.71**

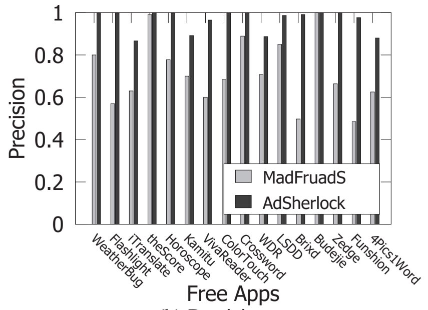
To evaluate the overall performance of ad request identification, we calculate the area under the precision-recall curve (AUPRC) of each method since it is proved to be appropiate for imbalanced datasets in previous studies [24], [25]. For AdSherlock, we use a normalized threshold representing the difference between the ad score and non-ad score of network requests and change the threshold from 0 to 1. On each threshold, a pair of precision/recall values are computed, and are finally combined to form a precision-recall (PR) curve. Then, we compute the AUPRC to describe the performance of the classification. Similarly, we obtain the PR curve of MadFraudS by varying the threshold of the random forests used and compute the AUPRC. As a result, we obtain the AUPRC of AdSherlock and MadFraudS as 0.911 and 0.797 respecativelly, indicating AdSherlock's better overall performance.

Click fraud detection performance. We evaluate the performance of click fraud detection in two real-world scenarios: bot-driven fraudulent scenario and in-app fraudulent scenario. Fig. 11 show the result of fraud detection in the bot-driven scenario. In this scenario, the fraud detection accuracy is determined by the performance of ad request identification. Fig. 11(a) demonstrates that both AdSherlock and MadFraudS have a high recall across different apps. Among them, the app iTranslate shows a relatively low recall under MadFraudS. The reason is that the traffic of iTranslate mainly consists of ad requests and analytic requests, which are hard for MadFraudS to deviate. Fig. 11(b) and (c) show that AdSherlock has a higher precision than MadFraudS as well as a higher F1–measure. This is because that MadFraudS has higher false positives. The false positives may come from: 1) analytic requests which have a very similar format to ad requests; 2) limited inputs. Instead of inputs collected from server-side in [5], the input of MadFraudS is collected from the client side for each app and misses cross-application features. We summarize the statistical results of click fraud detection in Table 2. We can see that AdSherlock achieves relatively higher average values of precision and F1-measure with lower standard deviation, which means a more robust detection strategy.

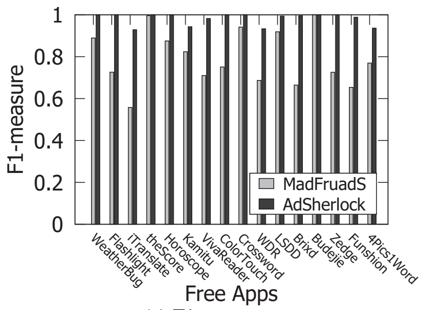
Fig. 12 shows the results of fraud detection in the in-app fraudulent scenario. We can see that both of them achieve low false positive rate and have an increasing true positive rate as



(a) Recall.

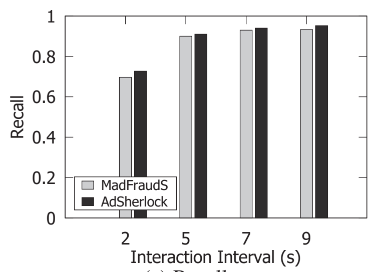


(b) Precision.

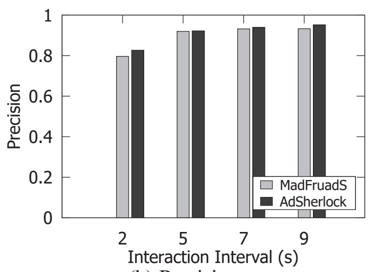


(c) F1–measure.

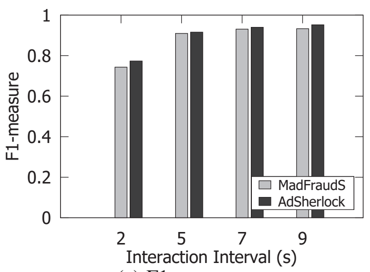
Fig. 11: Performance of click fraud detection in bot-driven fraudulent scenario.



(a) Recall.

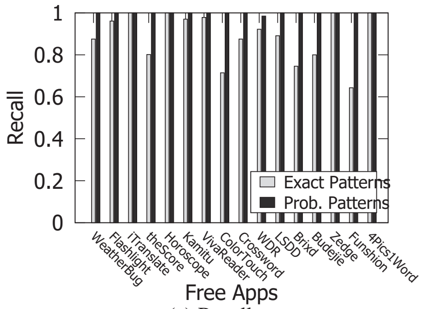


(b) Precision.

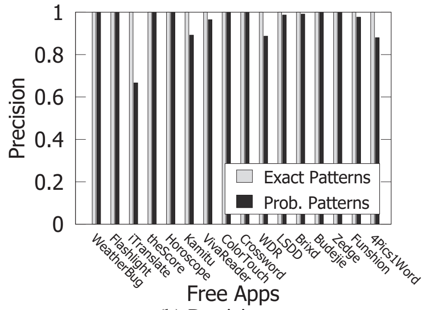


(c) F1-measure.

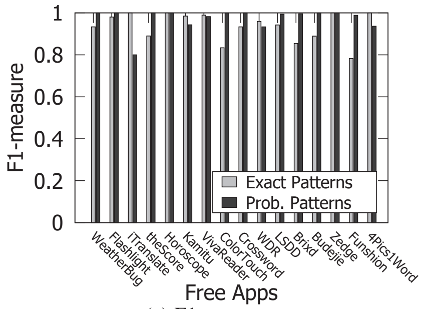
Fig. 12: Performance of click fraud detection in in-app fraudulent scenario.



(a) Recall.



(b) Precision.



(c) F1-measure.

Fig. 13: Performance of exact patterns and probabilistic patterns generated by AdSherlock.

well as the precision when the interaction interval increases. This is due to that the motion events generated by the user can hinder the detection of in-app fraudulent clicks. Larger interaction interval means less noise in the motion event traces and results in higher true positives as well as less false negatives. Specifically, MadFraudS has a higher true positive rate than AdSherlock. This is mainly due to that MadFraudS relies on statistical features which may introduce relatively high false negatives when the input data is limited.

7.4 System Insights

In this section, we first evaluate the impact of different patterns on the performance of AdSherlock. We evaluate the performance of each single pattern on test apps. Second, we evaluate the impact of noise on the performance of different patterns. Third, we evaluate the runtime overhead and memory usage of AdSherlock.

7.4.1 Performance of Different Patterns

We first evaluate the performance of exact patterns and probabilistic patterns on test apps. For each app, the patterns are generated according to its EmulateSet. Then, we evaluate their performance on RealSet. Fig. 13(a) shows that the true positive rate of probabilistic patterns is higher than exact patterns across all selected apps. For all apps, probabilistic patterns achieve a high true positive rate above 98%. This is because probabilistic patterns are more resilient to changes in traffic and have higher true positives. Fig. 13(b) shows that exact pattern performs better in terms of false positive rate. For all selected apps, both kinds of patterns achieve a low false positive rate under 0.4%. Among them, eight apps have higher false positives under probabilistic patterns than exact patterns. This is reasonable since probabilistic patterns are less rigid than exact patterns. This also results in a lower precision using probabilistic patterns as shown in Fig. 13(c).

7.4.2 Impact of noise

Now we evaluate the impact of noises on the performance of different kinds of patterns. The noises are misclassified requests by the request classifier. Since false positives are often considered more harmful than false negatives, we mainly focus on evaluating the false positives caused by noises. Fig. 14 shows the results

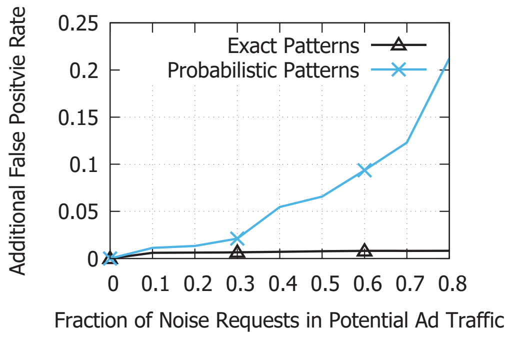


Fig. 14: Impact of noise in potential ad traffic.

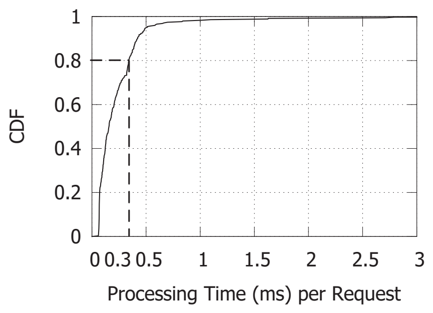


Fig. 15: The processing time of each request in AdSherlock.

of *additional false positives* with respect to different noise ratio. The additional false positives are false positives raised due to the additional noise, excluding false positives that raised by correct patterns. The noise ratio is the ratio of misclassified non-ad requests over the total number of ad requests. For exact patterns, the additional false positive rate is below 1% in all noise ratios. The reason is that the exact patterns are generated from hierarchical clustering without causing false positives. For probabilistic patterns, the additional false positive rate rises to 21% when the noise ratio growing beyond 80%. This because tokens in the noise requests have a relatively high ad score and cause more false positives.

7.4.3 Overheads

We now evaluate AdSherlock's overheads in terms of runtime efficiency and memory overhead.

The impact of AdSherlock on run time mainly comes from the pattern matching module and the ad tree construction module. We measure the *processing time* per request, taking into account the influence of both modules. As shown in Fig. 15, 80% of the requests are processed in less than 300 μs. Thus, the impact of AdSherlock on run time of the app is negligible.

Another critical factor of evaluation on mobile devices is the overhead posed on memory and storage, which are both limited resources on smartphones. AdSherlock introduces only $2.7k$ memory overhead on average for exact patterns storage and $3.0k$ memory overhead for motion event collection. Moreover, AdSherlock shows a tiny 130KB increase on the size of the application, being only 1.3% of the original.

8 Conclusion

AdSherlock is an *efficient* and *deployable* click fraud detection approach for mobile apps at the *client side*. As a client-side approach, AdSherlock is orthogonal to existing server-side approaches. It splits the computation intensive operations of click request identification into an offline process and an online process. In the offline process, AdSherlock generates both exact patterns and probabilistic patterns based on url tokenization. These patterns are used in the online process for click request identification, and further used for click fraud detection together with an ad request tree model. Evaluation shows that AdSherlock achieves high click fraud detection accuracy with a negligible runtime overhead. In the future, we plan to combine static analysis with the traffic analysis to improve the accuracy of ad request identification and explore attacks designed to evade AdSherlock.

Acknowledgments

This work is supported by the National Key R&D Program of China (No. 2019YFB1600700), National Science Foundation of China (No. 61872437 and No. 61772465), the Research and Development Program in Key Areas of Guangdong Province (No. 2018B010113001), and the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020001. Yi Gao is the corresponding author.

References

[1] "Mobile advertising spending worldwide." [Online]. Available: https://www.statista.com/statistics/280640/mobile-advertising-spending-worldwide/
[2] "Google admob." [Online]. Available: https://apps.admob.com/
[3] M. Mahdian and K. Tomak, "Pay-per-action model for online advertising," in *Proc. of ACM ADKDD*, 2007.
[4] G. Cho, J. Cho, Y. Song, and H. Kim, "An empirical study of click fraud in mobile advertising networks," in *Proc. of ACM ARES*, 2015.
[5] J. Crussell, R. Stevens, and H. Chen, "Madfraud: Investigating ad fraud in android applications," in *Proc. of ACM MobySys*, 2014.
[6] R. Oentaryo, E.-P. Lim, M. Finegold, D. Lo, F. Zhu, C. Phua, E.-Y. Cheu, G.-E. Yap, K. Sim, M. N. Nguyen, K. Perera, B. Neupane, M. Faisal, Z. Aung, W. L. Woon, W. Chen, D. Patel, and D. Berrar, "Detecting click fraud in online advertising: A data mining approach," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 99–140, 2014.
[7] B. Kitts, Y. J. Zhang, G. Wu, W. Brandi, J. Beasley, K. Morrill, J. Ettedgui, S. Siddhartha, H. Yuan, F. Gao, P. Azo, and R. Mahato, *Click Fraud Detection: Adversarial Pattern Recognition over 5 Years at Microsoft*. Cham: Springer International Publishing, 2015, pp. 181–201.
[8] A. Metwally, D. Agrawal, and A. El Abbadi, "Detectives: detecting coalition hit inflation attacks in advertising networks streams," in *Proc. of ACM WWW*, 2007.
[9] A. Metwally, D. Agrawal, A. El Abbad, and Q. Zheng, "On hit inflation techniques and detection in streams of web advertising networks," in *Proc. of IEEE ICDCS*, 2007.
[10] F. Yu, Y. Xie, and Q. Ke, "Sbotminer: large scale search bot detection," in *Proc. of ACM WSDM*, 2010.
[11] L. Zhang and Y. Guan, "Detecting click fraud in pay-per-click streams of online advertising networks," in *Proc. of IEEE ICDCS*, 2008.
[12] A. Metwally, D. Agrawal, and A. El Abbadi, "Duplicate detection in click streams," in *Proc. of ACM WWW*, 2005.
[13] M. S. Iqbal, M. Zulkernine, F. Jaafar, and Y. Gu, "Fcfraud: Fighting click-fraud from the user side," in *Proc. of IEEE HASE*, 2016.
[14] B. Liu, S. Nath, R. Govindan, and J. Liu, "Decaf: detecting and characterizing ad fraud in mobile apps," in *Proc. of USENIX NSDI*, 2014.
[15] G. Cho, J. Cho, Y. Song, D. Choi, and H. Kim, "Combating online fraud attacks in mobile-based advertising," *EURASIP Journal on Information Security*, vol. 2016, no. 1, p. 1, 2016.
[16] W. Li, H. Li, H. Chen, and Y. Xia, "Adattester: Secure online mobile advertisement attestation using trustzone," in *Proc. of ACM MobySys*, 2015.
[17] "Monkeyrunner." [Online]. Available: http://developer.android.com/studio/test/monkeyrunner/index.html
[18] "Zedge." [Online]. Available: https://play.google.com/store/apps/
[19] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, "Networkprofiler: Towards automatic fingerprinting of android apps," in *Proc. of IEEE INFOCOM*, 2013.
[20] "Mopub." [Online]. Available: https://www.mopub.com/

[21] J. Newsome, B. Karp, and D. Song, “Polygraph: Automatically generating signatures for polymorphic worms,” in *Proc. of IEEE S&P*, 2005.

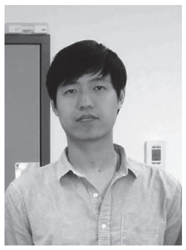
[22] “Android montionevent.” [Online]. Available: https://developer.android.com/reference/android/view/MotionEvent.html

[23] X. Jin, P. Huang, T. Xu, and Y. Zhou, “Nchecker: saving mobile app developers from network disruptions,” in *Proc. of ACM EuroSys*, 2016.

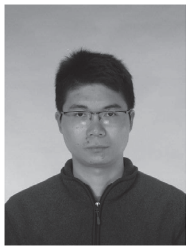
[24] J. Davis and M. Goadrich, “The relationship between precision-recall and roc curves,” in *Proc. of ACM ICML*, 2006.

[25] T. Saito and M. Rehmsmeier, “The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets,” *PloS one*, vol. 10, no. 3, p. e0118432, 2015.

Chenhong Cao received the B.S. and M.S. degrees in computer science from Northeastern University, China, in 2011 and 2013, respectively. She received the Ph.D. degree from Zhejiang University in 2018. She is currently an assistant professor in Shanghai University. Her research interests include network measurement, Internet of things, and mobile computing.



Yi Gao (M’15) received the B.S. and Ph.D. degrees in Zhejiang University in 2009 and 2014, respectively. He is currently an associate professor in Zhejiang University, China. From 2015 to 2016, he visited McGill University as a visiting scholar. His research interests include network measurement, Internet of things, and mobile computing. He is a member of the IEEE and the ACM.



Yang Luo received the MS degree from Zhejiang University, in 2017. He is now a soft-ware development engineer. His research interests include mobile computing and wireless sensing.

Mingyuan Xia received his Ph.D. degree from McGill Universirty in 2017. His research interests focus on mobile systems, program analysis and distributed storage. He received IBM Ph.D. Fellowship in year 2015.

Wei Dong (S’08–M’12) received the B.S. and Ph.D. degrees in computer science from Zhejiang University, Hangzhou, China, in 2005 and 2011, respectively. He is currently a professor in College of Computer Science, Zhejiang University. His research interests include networked embedded systems, network measurement and wireless sensing.

Chun Chen (M’08) received his Bachelor of Mathematics degree from Xiamen University, China, in 1981, and his M.S. and Ph.D. degrees in Computer Science from Zhejiang University, China, in 1984 and 1990 respectively. He is a professor in College of Computer Science, and an academician of the Chinese Academy of Engineering. His research interests include embedded systems, image processing, computer vision, and CAD/CAM.

Xue Liu is a professor and William Dawson Scholar in the School of Computer Science at McGill University. He received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 2006. He received his B.S. degree in Mathematics and M.S. degree in Automatic Control both from Tsinghua University, China. He has also worked as the Samuel R. Thompson Associate Professor in the University of Nebraska-Lincoln and HP Labs in Palo Alto, California. His research interests are in computer networks and communications, smart grid, real-time and embedded systems, cyber-physical systems, data centers, and software reliability. Dr. Liu has been granted 4 US patents and published over 200 research papers in major peer-reviewed international journals and conference proceedings, including the Year 2008 Best Paper Award from IEEE Transactions on Industrial Informatics, and the First Place Best Paper Award Security (WiSec 2011). He is a recipient of the Outstanding Young Canadian Computer Science Researcher Prizes from the Canadian Association of Computer Science. He is a Fellow of IEEE.