FraudDroid: Automated Ad Fraud Detection for Android Apps

Feng Dong
Haoyu Wang
Beijing University of Posts and Telecommunications, China

Li Li
Tianming Liu
Guoai Xu
Faculty of Information Technology, Monash University, Australia
Beijing University of Posts and Telecommunications, China

Yao Guo
Tegawendé F. Bissyandé

1 INTRODUCTION

The majority of apps in Android markets are made available to users for free [5, 40, 66, 76]. This has been the case since the early days of the Android platform when almost two-thirds of apps were free to download. Actually, developers of third-party free apps are compensated for their work by leveraging in-app advertisements (ads) to collect revenues from ad networks [31]. The phenomenon has become common and is now part of the culture in the Android ecosystem where advertisement libraries are used in most popular apps [66]. App developers get revenue from advertisers based either on the number of ads displayed (also referred to as impressions) or the number of ads clicked by users (also referred to as clicks) [61].

While mobile advertising has served its purpose of ensuring that developer interests are fairly met, it has progressively become plagued by various types of frauds [45, 56]. Unscrupulous developers indeed often attempt to cheat both advertisers and users with fake or unintentional ad impressions and clicks. These are known as ad frauds. For example, mobile developers can typically employ individuals or bot networks to drive fake impressions and clicks so as to earn profit [15]. A recent report has estimated that mobile advertisers lost up to 1.3 billion US dollars due to ad fraud in 2015 alone [28], making research on ad fraud detection a critical endeavour for sanitizing app markets.

Research on ad frauds has been extensively carried in the realm of web applications. The relevant literature mostly focuses on click fraud which generally consists of leveraging a single computer or botnets to drive fake or undesirable impressions and clicks. A number of research studies have extensively characterized click frauds [1, 10, 54] and analysed its profit model [51]. Approaches have also been proposed to detect click frauds by analysing network traffic [52, 53] or by mining search engine’s query logs [81].

Nevertheless, despite the specificities of mobile development and usage models, the literature on in-app ad frauds is rather limited. One example of work is the DECFA approach for detecting placement frauds: these consist in manipulating visual layouts of
While detection of the former can be performed via analysing the video a taxonomy that characterizes a variety of mobile ad frauds in-

\( \text{• Dynamically exercising ad views in a UI state, achieving scalability, of the latter presents several challenges, notably for:} \)

- 

\( \text{• Automaticaly distinguishing ad views among other views: In contrast with UI on the Windows Phone platform targeted by the state-of-the-art (e.g., DECAF [45]), Android UI models are generic and thus it is challenging to identify ad views in a given UI state since no explicit labels are provided to distinguish them from other views (e.g., text views). During app development, a view can be added to the Activity, which represents a UI state implementation in Android, by either specifying it in the XML layout [21] or embedding it in the source code. In preliminary investigations, we found that most ad views are actually directly embedded in the code, thus preventing any identification via XML analysis.} \)

Towards building an approach that achieves accuracy and scalability in Android ad fraud detection, we propose two key techniques aimed at addressing the aforementioned challenges:

- 

\( \text{• Transition graph-based UI exploration. This technique builds a UI transition graph by simulating interaction events associated with user manipulation. We first capture the relationship between UI states through building the transition graphs between them, then identify ad views based on call stack traces and unique features gathered through comparing the ad views and other views in UI states. The scalability of this step is boosted by our proposed ad-first exploration strategy, which leverages probability distributions of the presence of an ad view in a UI state.} \)

- 

\( \text{• Heuristics-supported ad fraud detection. By manually investigating various real-world cases of ad frauds, we devise heuristic rules from the observed characteristics of fraudulent behaviour. Runtime analysis focusing on various behavioural aspects such as view size, bounds, displayed strings or network traffic, is then mapped against the rules to detect ad frauds.} \)

These techniques are leveraged to design and implement a prototype system called FraudDroid for detecting ad frauds in Android apps. This paper makes the following main contributions:

1. We create a taxonomy of existing mobile ad frauds. This taxonomy, which consists of 9 types of frauds, includes not only previously studied static placement frauds, but also a new category of frauds which we refer to as dynamic interaction frauds.

2. We propose FraudDroid, a new approach to detect mobile ad frauds based on UI transition graphs and network traffic analysis. Empirical validation on a labelled dataset of 100 apps demonstrates that FraudDroid achieves a detection precision of 93% and recall of 92%. To the best of our knowledge, FraudDroid is the first approach that is able to detect the five types of dynamic interaction ad frauds presented in our taxonomy.

3. We have applied FraudDroid in the wild on 12,000 apps from major app markets to demonstrate that it can indeed scale to markets. Eventually, we have identified 355 apps performing ad frauds, some of them are popular apps with millions of downloads. 94 of such apps even come from the official Google Play store, which indicates that measures are not yet in place to fight fraudulent behaviour. We have released the benchmarks and experiment results to our research community at:

https://github.com/FraudDroid-mobile-ad-fraud

---

An educational music player (com.android.yatree.taijiaomusic\(^3\)) targeting pregnant mothers for antenatal training.
2 A TAXONOMY OF MOBILE AD FRAUDS

Before presenting the taxonomy of mobile ad frauds, we briefly overview the mobile advertising ecosystem. Figure 2 illustrates the workflow of interactions among different actors in the mobile advertising ecosystem.

Figure 2: An overview of the mobile advertising ecosystem.

2.1 Mobile Advertising

The role of an advertiser is to design ads that will be distributed to user devices. Such ads are displayed through third-party apps that are published by app developers. The ad network thus plays the role of a trusted intermediary platform, which connects advertisers to app developers by providing toolkits (e.g., in the form of ad libraries that fetch and display ads at runtime) to be embedded in app code. When a user views or clicks on an ad, the ad network (which is paid by advertisers) receives a feedback based on which the app developer is remunerated.

Because app developers earn revenues based on the number of ad impressions and clicks, it is tempting to engage in fraudulent behaviour to the detriment of users or/and advertisers. To avoid app developers tricking users into clicking ad views (and thus artificially increasing their revenues), ad networks have put in place some strict policies and prohibition guidelines on how ad views should be placed or used in apps. For example, violations of any of the AdMob program policies [24, 29, 58] is regarded as ad frauds by the Google AdMob ad network. Besides, popular app markets [49, 50, 58, 70] have released strict developer policies on how the ads should be used in apps. Nevertheless, unscrupulous app developers resort to new tricks to commit ad frauds that market screeners fail to detect while ad networks are un-noticably cheated. This is unfortunate as ad frauds have become a critical concern for the experience of users, the reputation of ad networks, and the investments of advertisers.

2.2 Ad Frauds

While the literature contains a large body of work on placement frauds in web applications and the Windows Phone platform, very little attention has been paid to such frauds on Android ecosystem. Furthermore, dynamic interaction frauds have even not been explored by the research community to the best of our knowledge.

Definition of Mobile Ad Fraud. To build the taxonomy of Android ad frauds, we first investigate the example policy violations listed by app markets and ad-networks, including: (1) the usage policies provided by popular ad libraries [25, 29], (2) the developer policies provided by official Google Play market [58] and popular third-party app markets, including Wanzhoujia (Alibaba App) Market [70], Huawei App Market [49] and Tencent Myapp Market [50], (3) the guidelines on ad behaviour drafted by a communication standards association [6], and (4) some real-world ad fraud cases. Figure 3 presents our taxonomy, which summarizes 9 different types of ad frauds, which represents by far the largest number of ad fraud types. Particularly, the five types of dynamic interaction frauds have never been investigated in the literature. Note that some cases tagged as frauds may actually be perceived as "annoying ads", and app markets view them as bad developer behaviours that break ad policies and take measures to remove the associated apps.

2.2.1 Static Placement Frauds. Many fraud cases are simply performed by manipulating the ad view form and position in a UI state. "Static" implies that the detection of these frauds could be determined by static information and occur in a single UI state. "Placement" implies that the fraudulent behaviour is exploiting placement aspects, e.g., size, location, etc. We have identified four specific types of behaviour related to static placement frauds:

1. The Ad Hidden fraud. App developers may hide ads (e.g., underneath buttons) to give the illusion of an "ad-free app" which would ideally provide better user experience. Such ads however are not displayed in conformance with the contract with advertisers who pay for the promotional role of ads [29, 70].

2. The Ad Size fraud. Although advice on ad size that ad networks provide is not mandatory, and there are no standards on ad size, the size ratio between the ad and the screen is required to be reasonable [70], allowing the ads to be viewed normally by users [26]. Fraudulent behaviour can be implemented by stretching ad size to the limits: with extremely small ad sizes, app developers may provide the feeling of an ad-free app, however cheating advertisers; similarly, with abnormally large ad size, there is a higher probability to attract users’ attention (while affecting their visual experience), or forcing them to click on the ad in an attempt to close it.

3. The Ad Number fraud. Since ads must be viewed by users as mere extras alongside the main app content, the number of ads must remain reasonable [25, 70]. Unfortunately, developers often include a high number of ads to increase the probability of attracting user interests, although degrading the usage experience of the app, and even severely affecting the normal functionality when ad content exceeds legitimate app content.

4. The Ad Overlap fraud. To force users into triggering undesired impressions and clicks, app developers may simply display ad views on top of actionable functionality-relevant views [25, 29, 70]. By placing ads in positions that cover areas of interest for users in typical app interactions, app developers create annoying situations where users must “acknowledge” the ad.
2.2.2 Dynamic Interaction Frauds. We have also identified cases of frauds that go beyond the placement of ad views on a single UI state, but rather involve runtime behavior and may then occur in an unexpected app usage scenario. "Dynamic" implies that the detection of these frauds occurs at runtime. "Interaction" implies that the fraudulent behavior is exploiting user interaction scenarios and may involve multiple UI states.

(5) The Interaction Ad fraud. In web programming, developers use interstitials (i.e., web pages displayed before or after an expected page content) to display ads. Translated into mobile programming, some ad views are placed when transitioning between UI states. However, frauds can be performed by placing interstitial ads early on app load or when exiting apps, which could trick users into accidental clicks since interaction with the app/device is highly probable at these times [6, 70].

(6) The Drive-by download Ad fraud. Ads are meant to provide a short promotional content designed by advertisers to attract user attention into visiting an external content page. When app developers are remunerated not by the number of clicks but according to the number of users that are eventually transformed into actual consumers of the advertised product/service, there is a temptation of fraud. A typical fraud example consists in triggering unintentional downloads (e.g., of advertised APKs) when the ad view is clicked on [6, 70]. Such behavior often heavily impacts user experience, and in most cases, drive-by downloads cannot even be readily cancelled.

(7) The Outside Ad fraud. Ads are supposedly designed to appear on pages when users use the app. Fraudulent practices exist however for popping up ads while apps are running in the background, or even outside the app environment (e.g., ad views placed on the home screen and covering app icons that users must reach to start new apps) [6, 29, 49, 70]. In some extreme cases, the ads appear spuriously and the user must acknowledge them since such ads can only be closed when the user identifies and launches the app from which they come.

(8) The Frequent Ad fraud. App developers try to maximize the probability of ad impressions and clicks to collect more revenue. This probability is limited by the number of UI states in the app. Thus, developers may implement fraudulent tactics by displaying interstitial ads every time the user performs a click on the app’s core content (e.g., even when the click is to show a menu in the same page) [6, 29].

(9) The Non-content Ad fraud. To maximize the number of ad impressions and trick users into unintended clicks, developers can place ads on non-content-based pages such as thank you, error, login, or exit screens. Ads on these types of UI states can confuse a user into thinking that the ads are real app content [29].

3 FRAUDDROID

To address ad frauds in the Android ecosystem we design and implement FRAUDDROID, an approach that combines dynamic analysis on UI state as well as network traffic data to identify fraudulent behaviours. Figure 4 illustrates the overall architecture of FRAUDDROID. The working process unfolds in two steps: (1) analysis and modelling of UI states, and (2) heuristics-based detection of ad frauds.

To efficiently search for ad frauds, one possible step before sending apps to FRAUDDROID is to focus on such apps that have included ad libraries. To this end, FRAUDDROID integrates a preprocessing step, which stops the analysis if the input app does not leverage any ad libraries, i.e., there will be no ad frauds in that app. Thus we first propose to filter apps that have no permissions associated with the functioning of ad libraries, namely INTERNET and ACCESS_NETWORK_STATE [39]. Then, we leverage LibRadar [43, 46, 72], a state-of-the-art, obfuscation-resilient tool to detect third-party libraries (including ad libraries) in Android apps.

3.1 Analysis and Modelling of UI States

While an app is being manipulated by users, several UI states are generated where ad views may appear. UI states indeed represent the dynamic snapshots of pages (i.e., Activity rendering) displayed on the device. One key hypothesis in FRAUDDROID is that it is possible to automate the exercise of ad views by traversing all UI states. To that end, we propose to leverage automatic input generation techniques to collect UI information. Nevertheless, since exhaustively exercising an app is time-consuming, the analysis cannot scale to market sizes. For example, in our preliminary experiments, the analysis of a single app required several hours to complete. Empirical investigations of small datasets of ad-supported apps have revealed however that over 90% of the UI states do not include an ad view [55]. We thus develop in FRAUDDROID a module, hereafter referred to as FraudBot, which implements an automated and scalable approach for triggering relevant UI states. This module exploits a fine-tuned exploration strategy to efficiently traverse UI states towards reaching most of those that contain ad views in a limited amount of time. FraudBot thus focuses on modelling a UI State Transition Graph (UTG for short), i.e., a directed graph where:

- A node represents a UI state, and records information about the network traffic that this state is associated with, the trace of method calls executed while in this state, and the view tree of the UI layout in this state.
- An edge between two nodes represents the test input (e.g., event) that is associated with the transition between two states.

Figure 5 illustrates an example of UTG, which is constructed on the fly during app automation where input event triggering state...
changes as well as information on the new state are iteratively added to an initially empty graph. Before discussing how FraudDroid exploits the UTG to unroll the app execution behaviour so as to detect ad frauds, we provide more details on the exploration strategy of FraudBot as well as on the construction of view trees.

3.1.1 Ad-first Exploration Strategy. We developed a strategy that prioritizes the traversal of UI states which contain ads to account for coverage and time constraints. To simulate the behaviour of real app users, FraudBot generates test inputs corresponding to typical events (e.g., clicking, pressing) that are used to interact with common UI elements (e.g., button, scroll). Inspired by previous findings [55] that most ads are displayed in the main UI state and the exit UI state, our exploration is biased towards focusing on these states. To prioritize ad views in a state, we resort to a breadth-first search algorithm where all views in a state are reordered following ad load method traces and other ad-related features which are further described in Section 3.2.1. Considering that loading an ad from a remote server may take time, we set the transition time in app automation to 5 seconds, a duration that was found sufficient in our network experimental settings. By using this strategy, we can reduce the app automation time from 1 hour per app to 3 minutes on average, which is a significant gain towards ensuring scalability.

3.1.2 View Tree Construction. For each state generated by FraudBot, a view tree is constructed to represent the layout of the state aiming to provide a better means to pinpoint ad views. The tree root represents the ground layout view on top of which upper views are placed. By peeling the UI state we build a tree where each view is represented as a node: parent nodes are containers to child nodes, and users actually manipulate only leaf nodes. Each node is tagged with basic view information such as position, size, class name, etc. Such attributes are used to identify which nodes among the leaf nodes are likely ad view nodes. Figure 6 illustrates an example of a view tree where a leaf node representing a pop-up view that is identified as an ad view. The identification is based on the string feature representing the in-app class name ("com.pop.is.ar", a customized type), the bound values of the view ([135, 520], [945, 1330]) corresponding to the center position of the device screen, as well as the size of view (810x810) which is common to interstitial ads. We describe in more detail in the next section the string and placement features that are used to detect ads.

3.1.3 Network Traffic Enhancement. Because some ad fraudulent behaviours such as drive-by download cannot be detected solely based on view layout information, we enhance the constructed view tree with network traffic data by connecting each view node with its associated HTTP request and the data (e.g., APK files) transmitted. The associated network data is helpful for FraudDroid to detect silent downloading behaviours where certain files such as APKs are downloaded without user’s interaction after an ad is clicked.

3.2 Heuristics-based Detection of Ad Frauds

Once a UI state transition graph is built, FraudDroid can find in it all necessary information for verifying whether an ad fraud is taking place in the app. These information include, for each relevant state, the layout information, the method call trace and the network traffic information. We implement in FraudDroid two modules to perform the ad fraud detection, namely AdViewDetector for identifying ad views in a state, and FraudChecker, for assessing whether the ad view is appearing in a fraudulent manner.

3.2.1 Detection of Ad Views. We recall that Android views lack explicit labels that would allow to readily discriminate ad views from other views. In a previous step of FraudDroid, the ad-first exploration strategy was guided towards reaching UI states which are likely to include ads, as recommended by Suman Nath [55]. For each of the reached UI state, the AdViewDetector module first checks if it has involved in ad-loading methods, as illustrated in Figure 7. After this step, most UI states are excluded from consideration as they are not really ad-contained states. For the remaining UI states, the AdViewDetector module of FraudDroid again excludes irrelevant states that are not ad-contained ones, where all the leaf nodes in the view tree associated to the UI states are checked against common ad features (i.e., String, Type and Placement features).

We have identified relevant ad features after manually labelling a large amount of ad views and normal views that we have compared them from different aspects (namely string, type and placement features). Table 1 presents some sample values for the ad detection features considered.
String. Manual investigations of our labelled datasets revealed that String attributes in view definitions, such as associated class names, can be indicative of whether it is an ad view. Most ad libraries often implement specialized classes for ad views instead of directly using Android system views ("android.widget.Button", "android.widget.TextView", etc.). We noted however that these specialized classes’ names reflected their purpose with the keyword “ad” included: e.g., “AdWebview”, “AdLayout”. Since simply matching the keyword “ad” in class names would lead to many false positives with common other words (e.g., shadow, load, etc.), we rely on a whitelist of English words containing “ad” based on the Spell Checker Oriented Word Lists (SCOWL) [36]. When a view is associated to a class name matches the string hint “ad” but is not part of the whitelist, we take it as a potential ad view.

Type. Since some ad views may also be implemented based on Android system views, we investigate which system views are most leveraged. All ad views in our labelled dataset are associated to three types: “ImageView” [20], “WebView” [22] and “ViewFlipper” [23], in contrast to normal views which use a more diverse set of views. We therefore rely on the type of view used as an indicator of probable ad view implementation.

Placement. In general, ad views have special size and location characteristics, which we refer to as placement features. Indeed, mobile ads are displayed in three common ways: 1) Banner ad is located in the top or bottom of the screen; 2)Interstitial ad is square and located in the centre of the screen; 3) Full-screen ad fills the whole screen. Furthermore, ad networks actually hard-code in their libraries the size for the different types of views implemented [29], which are also leveraged in FraudDroid to identify possible ad views.

Using the aforementioned features, we propose a heuristic-based approach to detect ad views from a UI state. First, string and type features help to identify respectively customised views that are good candidates for ad views. Subsequently, placement features are used to decide whether a candidate ad view will be considered as such by FraudDroid. We have empirically confirmed that this process accurately identifies ad views implemented by more than 20 popular ad libraries, including Google Admob and Waps.

3.2.2 Identification of Fraudulent Behaviour. Once ad views are identified across all UI states in the UTG, the FraudChecker module can assess their tag information to check whether a fraudulent behaviour can be spotted. This module is designed to account for the whole spectrum of frauds enumerated in the taxonomy of Android ad frauds (cf. Section 2.2). To that end, FraudChecker implements heuristics rules for each specific ad fraud type:

Ad Hidden. FraudChecker iteratively checks whether any ad view has boundary coordinate information which would indicate that it is (partially or totally) covered by any other non-ad views: i.e., it has a z-coordinate below a non-ad view and the 2-D space layout occupied by both views intersect at some point.

Ad Size. Although the ad view size may vary on different devices, our manual investigation has shown that the size ratio between ad view and the screen is relatively stable for legitimate ads. We empirically define a ratio of [0.05, 0.1] for banner ads, [0.2, 0.8] for interstitial ads and [0.9, 1] for full-screen ads. FraudChecker uses these values as thresholds, combined with position information (cf. Taxonomy in Section 2.2), to determine whether there is a fraudulent behaviour. Note that these values are configurable.

Ad Number. In a straightforward manner, FraudChecker verifies that a basic policy is respected: the number of banners ads should be less than three, and the combined space occupied by all the banner ads in a UI state must not exceed the space reserved to app content. When there are more than one banner ad views in a UI state along side app content (i.e., all as view tree leaves), the size of ad views in total should not exceed 50% of the screen size.

Ad Overlap. Similarly to the case of Ad Hidden, FraudChecker checks whether the space occupied by an ad view on the screen intersects with the space of other normal views. In contrast to Ad Hidden, the overlapping views are on the same level in z-coordinate.

Interactive Ad. This fraud occurs with an interstitial ad. FraudChecker first traverses the UTG to identify any UI state that contains interactive views (e.g., dialogue or button) which are tempting for users to click. Then, it checks the next UI state in the UTG to inspect whether there is an ad view that is placed on top of the aforementioned interactive views (i.e., dialogue or button from the preceding UI state). If so, FraudDroid flags it as an interactive fraud.

Drive-by download Ad. We follow policy statements and consider that any behaviour that triggers unwanted downloads (of apps or other files) via ad view where user interaction (such as double confirmation) is not required is a fraudulent ad behaviour. FraudChecker flags a given UI state as drive-by download as long as the following conditions are met: 1) there are ad views in this state; 2) there is a downloading behaviour; 3) the next state in the UTG is still associated with the same Activity (e.g., it does not switch to other interfaces); and 4) the state is triggered by a touch event.

Outside Ad. Outside Ad fraud only occurs in the interstitial ads. Note that for the full screen ads pop up after exit states, we regard it as Non-content Ad fraud. FraudChecker keeps track of all the interstitial ads. For each ad view, it records the screen shot of current state, and compare the four side of the screen shot with the original home screen to measure their similarity using the Perceptual hashing [79] and Histogram [78] algorithms.

Frequent Ad. We consider an app is suspicious to frequent ad fraud as long as it has interstitial ads or full-screen ads displayed more than three times (this number is configurable) in the UTG, where the three displays are triggered by different UI state transitions (i.e., several visits via the same transition path to the same ad-contained UI state are considered as one visit).

<table>
<thead>
<tr>
<th>Aspects</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Resource_id</td>
<td>AdWebview, AdLayout, ad_container, fullscreenAdView, FullscreenAd, AdActivity, etc.</td>
</tr>
<tr>
<td>Type</td>
<td>Class</td>
<td>ImageView, WebView, ViewFlipper</td>
</tr>
<tr>
<td>Placement</td>
<td>Size (Bounds)</td>
<td>620<em>800[Center], 320</em>50[Top or Bottom], 1776*1080[Full screen], etc.</td>
</tr>
</tbody>
</table>
FraudDroid: Automated Ad Fraud Detection for Android Apps

**Non-content Ad.** FraudChecker flags such UI states as Non-content Ad fraud when it identifies that interstitial ads or full screen ads exist before or after launch/login/exit states.

### 3.3 Implementation

We have implemented a lightweight UI-guided test input generator to dynamically explore Android apps with a special focus on UI states. The UI-guided events are generated according to the position and type of UI elements. Regarding network traffic, we leverage Tcpdump [64] and The Bro Network Security Monitor [59], respectively, to harvest and analyse the network traffic.

### 4 EVALUATION

We evaluate the effectiveness of FraudDroid with respect to its capability to accurately detect ad frauds (cf. Section 4.1), and its scalability performance (cf. Section 4.2) through measurements on runtime and memory cost of running FraudDroid on a range of real-world apps. Eventually, we run FraudDroid in the wild, on a collected set of 12,000 ad-supported apps leveraging 20 ad networks, in order to characterize the spread of ad frauds in app markets (cf. Section 4.3). All experiments are performed on Nexus 5 smartphones. We do not use emulators since ad libraries embed checking code to prevent ad networks from serving ads when the app is being experimented on emulator environments [65].

#### 4.1 Detection Accuracy

Accuracy is assessed for both the detection of ad views by AdViewDetector and the actual identification of frauds by FraudChecker. In the absence of established benchmarks in this research direction, we propose to manually collect and analyse apps towards building benchmarks that we make available to the community.

**AdViewBench.** Our first benchmark set includes 500 apps that were manually exercised to label 4,403 views among which 211 are ad views. Two authors manually labelled and cross-validated this dataset. If consensus was not reached among them, more authors were involved and a majority voting was implemented.

**AdFraudBench.** Our second benchmark set includes 100 apps that embed ads, 50 of which exhibit fraudulent behaviours that are manually picked and confirmed. To select the benchmark apps, we have uploaded more than 3,000 apps that use ad libraries to VirusTotal [69], and manually checked the apps that are labelled as AdWare by at least two engines from VirusTotal. We selected apps to cover all 9 types of frauds, from the static placement and dynamic interactive categories, and ensure that each type of fraud has at least two samples. Figure 8 illustrates the distribution of 50 labelled apps across the fraud types. We have obtained an overall of 54 ad fraud instances for 50 apps: some apps in the benchmark set indeed perform more than one types of ad frauds.

4.1.1 Accuracy in Ad View Identification. We run the AdViewDetector module on the UI states associated to the AdViewBench of 4,403 views. The detector reports 213 views as ad views, 197 of which are indeed ad views according to the benchmark labels. The detector however misses to identify 11 ad views in the benchmark.

4.1.2 Accuracy in Ad Fraud Detection. To evaluate the accuracy of our approach in detecting ad fraud behaviours, we run FraudDroid on the AdFraudBench benchmark apps. Table 3 provides the confusion matrix obtained from our experiments in classifying whether an app is involved in any ad fraud. Among the 100 apps in the benchmark, FraudDroid flags 49 as performing ad frauds: checking against the benchmarks, these detection results include 3 cases of false positives (1 Ad-Overlap and 2 Ad-Hidden) and 4 false negatives (1 Ad-Overlap and 3 Drive-by-download), leading to precision and recall metrics of 93.88% and 92% respectively.

Table 3: Confusion matrix of fraud detection by FraudDroid on the AdFraudBench benchmark.

<table>
<thead>
<tr>
<th>Predicted frauds</th>
<th>Ad fraud</th>
<th>w/o Ad fraud</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ad fraud</td>
<td>46 True Positives</td>
<td>4 False Negatives</td>
</tr>
<tr>
<td>w/o Ad fraud</td>
<td>3 False Positives</td>
<td>47 True Negatives</td>
</tr>
</tbody>
</table>

We further investigate the failure cases to understand the root causes of false positives/negatives. We find that all such cases are
caused by misclassification results of the AdViewDetector module. FraudDroid misses ad frauds because the UI state is not marked as relevant since no ad views were identified on it. Similarly, when FraudDroid falsely identified an ad fraud, it was based on behaviours related to a view that was wrongly tagged as an ad view.

4.2 Scalability Assessment
To assess the capability of FraudDroid to scale to app markets, we measure the runtime performance of the steps implementing the two key techniques in our approach: (1) the UI state transition graph construction, and (2) the heuristics-based detection of ad frauds. To comprehensively evaluate time and memory consumed by FraudDroid, we consider apps from a wide size range. We randomly select 500 apps in each of the following five size scales:\footnote{E.g., for the scale 100KB, the apps must be of size between 80KB and 120KB.}: 100KB, 1MB, 10MB, 50MB and 100MB. Overall, the experiments are performed on 2,500 apps for which we record the time spent to complete each step and the memory allocated.

The results are presented in Figure 9. The average time cost for constructing UI transition graphs (216.7 seconds) is significantly higher than that of detecting ad frauds (0.4 seconds). However, we note that, unlike for UI transition graphs, time to detect ad frauds is not linearly correlated to the app size. The UI transition graph is built by dynamically exploring the app where larger apps usually have more running states while detection solely relies on a pre-constructed graph. Nevertheless, thanks to the ad-first exploration strategy, FraudDroid is able to analyse every apps in the dataset within 3.6 minutes. Comparing with experiments performed in state-of-the-art app automation works \cite{12, 45}, FraudDroid provides substantial improvements to reach acceptable performance for scaling to app market sizes. Interestingly, memory consumption is reasonable (around 20MB), and roughly the same for both steps.

4.3 Detection In-the-Wild
We consider over 3 million Android apps crawled between April 2017 and August 2017 from eight app markets, namely the official Google Play store and alternative markets by Huawei, Xiaomi, Baidu, Tencent, PP, Mgyapp and 360 Mobile. Using LibRadar \cite{46}, we were able to identify ad libraries from 20 ad networks represented each in at least 500 apps. For the experiments in the wild, we randomly select 500 apps per ad network, except for Admob, for which we consider 2,500 apps to reflect the substantially larger proportion of ad-supported apps relying on Admob. Eventually, our dataset is formed by 12,000 Android apps, as shown in Table 4.

4.3.1 Overall Results. FraudDroid identified 335 (2.79\%) apps among the 12,000 dataset apps as implementing ad frauds. We note that ad frauds occur on a wide range of app categories, from Games to Tools. Some detected ad-fraud apps are even popular among users with over 5 million downloads. For example, the app Ancient Tomb Run \cite{14} has received over 5 million downloads, although FraudDroid has found that it contains a non-content ad fraud.

4.3.2 Ad Fraud Distribution by Type. Figure 10 shows the distribution of the apps based on the types of frauds. Static placement frauds only account for a small portion of the total detected ad fraud apps. Over 90\% of the fraud cases are dynamic interaction frauds. This is an expected result, since dynamic interaction frauds (1) are more difficult to detect, and thus they can easily pass vetting schemes on app markets, and (2) they are most effective in misleading users into triggering ad clicks, leading to more revenues.

These statistics show that no market is exempt from ad frauds, suggesting that, at the moment, markets have not implemented...
proper measures to prevent penetration of fraudulent apps into their databases. We also found that some markets do not even provide explicit policies to regulate the use of ads in apps.

4.3.4 Ad Fraud Distribution by Ad Networks. Table 5 presents the distribution of ad-fraud apps by ad networks. Although most ad frauds target some popular ad networks, no ad networks were exempt from fraudulent behaviours: Appbrain appears to be the most targeted with 13.6% of its associated apps involved in ad frauds. Interestingly, although Google Admob has published strict policies on how ads should be placed to avoid ad frauds, we still found 113 fraudulent apps associated to Admob. In some cases, we have found that several popular apps using a specific ad network library can exhibit serious fraud behaviours. For example, app Thermometer [34], with 5-10 million downloads in Google Play, is flagged by FraudDroid as implementing an interaction fraud. We also investigate user reviews of some fraudulent apps and confirmed that users have submitted various complaints about their ad fraud behaviours [13, 14, 34]. This evidence suggests that ad networks should not only publish explicit policies to regulate the usage of ad libraries, but also introduce reliable means to detect policy-violated cases for mitigating the negative impact on both app users and advertisers.

Table 5: Distribution of ad-fraud apps based on ad networks.

<table>
<thead>
<tr>
<th>Ad network</th>
<th>#Ad fraud</th>
<th>%Percent</th>
<th>Ad network</th>
<th>#Ad fraud</th>
<th>%Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admob</td>
<td>113</td>
<td>4.52%</td>
<td>Adwhirl</td>
<td>2</td>
<td>0.4%</td>
</tr>
<tr>
<td>Appbrain</td>
<td>68</td>
<td>13.6%</td>
<td>Dianjin</td>
<td>9</td>
<td>1.8%</td>
</tr>
<tr>
<td>Waps</td>
<td>68</td>
<td>9.6%</td>
<td>Vpon</td>
<td>1</td>
<td>0.2%</td>
</tr>
<tr>
<td>feiwo</td>
<td>29</td>
<td>5.8%</td>
<td>Innobi</td>
<td>6</td>
<td>1.2%</td>
</tr>
<tr>
<td>BaiduAd</td>
<td>10</td>
<td>2%</td>
<td>Aperhand</td>
<td>8</td>
<td>1.6%</td>
</tr>
<tr>
<td>Anzhi</td>
<td>7</td>
<td>1.4%</td>
<td>Startapp</td>
<td>5</td>
<td>1%</td>
</tr>
<tr>
<td>Youmi</td>
<td>8</td>
<td>1.6%</td>
<td>Mobwin</td>
<td>2</td>
<td>0.4%</td>
</tr>
<tr>
<td>Doodlemobile</td>
<td>4</td>
<td>0.8%</td>
<td>Jumptap</td>
<td>1</td>
<td>0.2%</td>
</tr>
<tr>
<td>Adsmogo</td>
<td>5</td>
<td>1%</td>
<td>Fyber</td>
<td>1</td>
<td>0.2%</td>
</tr>
<tr>
<td>Kugo</td>
<td>7</td>
<td>1.4%</td>
<td>Domob</td>
<td>1</td>
<td>0.2%</td>
</tr>
</tbody>
</table>

Ad network #Ad fraud %Percent Ad network #Ad fraud %Percent

(3) App com. natsume. stone. android implements an Outside Ad fraud where the ad view is displayed on the home screen although the app was exited.
(4) App com. funappdev. passionatelovers. frames includes a Frequent Ad fraud with an ad view popping up every time the main activity receives an event.
(5) App cc. chess is identified as showing a Non-content Ad since the ad view appears on the empty part of the screen (this may confuse users into thinking that the ad is an actual content of the host app).
(6) In app forest. best. livewallpaper, an Ad Hidden fraud has consisted in hiding an ad view behind the Email button.
(7) An Ad Size fraud is found in app com. dodur. android. golf with the ad view on the right side of the screen made too small for users to read.
(8) App com. maomao. androidcrack places three ad views on top of a page with little content, implementing an Ad Number fraud.
(9) App com. sysapk. wifibooster implements an Ad Overlap fraud where an ad view is placed on top of 4 buttons of the host app.

5 DISCUSSION
In this work, we empirically observe that ad frauds have penetrated into official and alternative markets. All ad networks are also impacted by these fraudulent practices, exposing the mobile advertising ecosystem to various threats related to poor user experience and advertisers’ losses. We argue that our community should invest more effort into the detection and mitigation of ad frauds towards building a trustworthy ecosystem for both advertisers and end users. FraudDroid contributes to such an effort by providing the building blocks in this research direction. The implementation of FraudDroid, however, carries several limitations.

Ad state coverage. Our experiment reveals that over 90% of ads are displayed in either the main UI state or the exit UI state, which provide a means for FraudDroid to optimize its UI exploration strategy in order to achieve a balance between time efficiency and UI coverage. However, this trade-off may cause certain ad views to be missed during UI exploration. Fortunately, FraudDroid provides parameters to customize the coverage (e.g., via traversal depth) and hence to improve the soundness of the approach to reach states where the ad fraud behaviour is implemented.

Other types of ad frauds. Although we have considered nine types of ad frauds, including five new types of dynamic interaction frauds, which have not been explored before, our taxonomy may still be incomplete since it was built based on current policies and samples available. Other emerging types of ad frauds may have been missed. Nevertheless, the framework is generic and can be reused to support the detection of potential new types of ad frauds.

6 RELATED WORK
This work is mainly related to three lines of research: automated app testing, ad fraud detection and analyzing ad libraries.

6.1 Automated App Testing
Automated app testing has been widely adopted for exploring apps at runtime [37]. Several Android automation frameworks such as Hierarchy Viewer [18], UIAutomator [19] and Robotium [32] have...
been developed to facilitate app testing. One critical step to perform automated app testing is to generate reliable test inputs [12]. The most straightforward means to achieve that is to follow a random strategy. Indeed, various tools such as Monkey [17], Dynodroid [47], Intent Fuzzer [60] and DroidFuzzer [80] have been introduced following this strategy. However, it is not guaranteed that random testing will reach a good coverage of the code explored, making it not suitable for some scenarios such as ad fraud detection where certain parts of the app (e.g., ad views) are expected to be covered.

Thus, researchers have introduced model-based strategy to generate test inputs for automatically exploring Android apps [2, 3, 7, 9, 11, 33, 42, 44, 63]. More advanced tools such as EvoDroid [48] and ACTEve [4] use sophisticated techniques such as symbolic execution and evolutionary algorithms to guide the generation of test inputs aiming at reaching specific points. However, most of them need to either instrument the system or the app, making them hard to be directly used to detect mobile ad frauds. Compared with them, we have implemented a sophisticated ad view exploration strategy for automated scalable ad fraud detection in this work.

6.2 Ad Fraud Detection

Existing work on mobile ad fraud is rather limited. Liu et al. [45] have investigated static placement frauds on Windows Phone via analysing the layouts of apps. Crussell et al. [15] have developed an approach for automatically identifying click frauds (fake impressions and clicks). Unfortunately, with the evolution of ad frauds, the aforementioned approaches are incapable of identifying the latest fraudulent behaviours, e.g., they cannot be used to identify dynamic interaction fraud. In this work, in addition to static placement frauds and click frauds, we have introduced five new types of such frauds that have not yet been explored by our community.

Although ad fraud has not been substantially explored in the context of mobile advertising, it has been extensively studied in the context of web advertising. Many research work have been proposed to pinpoint ad frauds on the web, e.g., the detection of click frauds based on network traffic [52, 53] or search engine query logs [81], characterizing click frauds [1, 10, 54] and analysing profit models [51]. We believe that these approaches can provide useful hints for researchers and practitioners in the mobile community to invent promising approaches for identifying mobile ad frauds.

6.3 Analyzing Ad Libraries

A large number of studies focused on the topic of mobile ad libraries in various directions including identifying ad libraries [8, 39, 41, 46, 72, 73], detecting privacy and security issues introduced by ad libraries [16, 30, 57, 62, 77], analysing the impact of ad libraries [71, 74, 75, 82], etc. We believe all the aforementioned approaches can be leveraged to supplement our work towards providing a better characterization of mobile ad frauds.

7 CONCLUSION

Through an exploratory study, we characterize ad frauds by investigating policies set up by ad networks and app markets. We then build a taxonomy as a reference in the community to encourage the research line on ad fraud detections. This taxonomy comprehensively includes four existing types of static placement frauds and five new types focusing on dynamic interaction frauds, which have not been explored in the literature. We subsequently devise and implement FraudDroid, a tool-supported approach for accurate and scalable detection of ad frauds in Android apps based on UI state transition graph and network traffic data. By applying FraudDroid to real-world market apps, we have identified 335 ad-fraud apps covering all considered nine types of ad frauds. Our findings suggest that ad frauds are widespread across markets and impact various markets. To the best of our knowledge, FraudDroid is the first attempt towards mitigating this threat to the equilibrium in the mobile ecosystem.

ACKNOWLEDGMENT

This work is supported by the National Key Research and Development Program of China (No.2018YFB0803603), the National Natural Science Foundation of China (No.61702045, No.61772042 and No.61402198), the Luxembourg National Fund under the project CHARACTERIZE C17/IS/11693861, and the BUPT Youth Research and Innovation Program (No.2017RC40).