# FraudDroid: Automated Ad Fraud Detection for Android Apps

Feng Dong[1], Haoyu Wang[1*], Li Li[2], Yao Guo[3], Tegawende F. Bissyande[4],
Tianming Liu[1], Guoai Xu[1], Jacques Klein[4]
[1] Beijing University of Posts and Telecommunications, China [2] Monash University, Australia [3] Peking University, China
[4] University of Luxembourg, Luxembourg

## ABSTRACT

Although mobile ad frauds have been widespread, state-of-the-art approaches in the literature have mainly focused on detecting the so-called *static placement frauds*, where only a single UI state is involved and can be identified based on static information such as the size or location of ad views. Other types of fraud exist that involve multiple UI states and are performed dynamically while users interact with the app. Such *dynamic interaction frauds*, although now widely spread in apps, have not yet been explored nor addressed in the literature. In this work, we investigate a wide range of mobile ad frauds to provide a comprehensive taxonomy to the research community. We then propose, FraudDroid, a novel hybrid approach to detect ad frauds in mobile Android apps. FraudDroid analyses apps dynamically to build UI state transition graphs and collects their associated runtime network traffics, which are then leveraged to check against a set of heuristic-based rules for identifying ad fraudulent behaviours. We show empirically that FraudDroid detects ad frauds with a high precision (∼ 93%) and recall (∼ 92%). Experimental results further show that FraudDroid is capable of detecting ad frauds across the spectrum of fraud types. By analysing 12,000 ad-supported Android apps, FraudDroid identified 335 cases of fraud associated with 20 ad networks that are further confirmed to be true positive results and are shared with our fellow researchers to promote advanced ad fraud detection.

## 1 INTRODUCTION

The majority of apps in Android markets are made available to users for free [5, 56, 61]. This has been the case since the early days of the Android platform when almost two-thirds of all 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 [27]. 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 [56]. 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*) [53].

While mobile advertising has served its purpose of ensuring that developers interests are fairly met, it has progressively become plagued by various types of frauds [37, 48]. 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 [13]. A recent report has estimated that mobile advertisers lost up to 1.3 billion US dollars due to ad fraud in 2015 alone [25], 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, 8, 46] and analysed its profit model [43]. Approaches have also been proposed to detect click frauds by analysing network traffic [44, 45] or by mining search engine's query logs [63].

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 DECAF [37] approach for detecting *placement frauds*: these consist in manipulating visual layouts of ad views (also referred to as elements or controls) to trigger undesirable impressions in Windows Phone apps. DECAF explores the UI states (which refer to **snapshots of the UI** when the app is running) in order to detect ad placement frauds implemented in the form of hidden ads, the stacking of multiple ads per page, etc. MAdFraud [13], on the other hand, targets Android apps to detect in-app click frauds by analysing network traffic.

Unfortunately, while the community still struggles to properly address well-known, and often trivial, cases of ad frauds, deception techniques used by app developers are even getting more sophisticated, as reported recently in news outlets [24, 31]. Indeed, besides the aforementioned click and placement frauds, many apps implement advanced procedures for tricking users into unintentionally clicking ad views while they are interacting with the app UI elements. In this work, we refer to this type of ad frauds as ***dynamic interaction frauds***.

Figure 1 illustrates the case of the app *taijiao music*[1] where an ad view gets unexpectedly popped up on top of the exit button when the user wants to exit the app: this usually leads to an unintentional ad click. Actually, we performed a user study on this app and found that 9 out of 10 users were tricked into clicking the ad view. To the best of our knowledge, such frauds have not yet been explored in the literature of mobile ad frauds, and are thus not addressed by the state-of-the-art detection approaches.

**This paper.** We perform an exploratory study of a wide range of new ad fraud types in Android apps and propose an automated approach for detecting them in market apps. To that end, we first provide a taxonomy that characterizes a variety of mobile ad frauds including both *static placement frauds* and *dynamic interaction frauds*. While detection of the former can be performed via analysing the

[*] Corresponding Author.

[1] An educational music player (*com.android.yatree.taijiaomusic*) targeting pregnant mothers for antenatal training.

*Conference'17, July 2017, Washington, DC, USA*
2018. ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

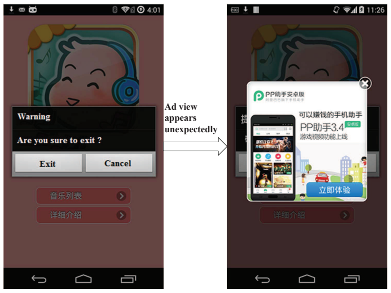


**Figure 1: An example of interaction fraud.** A rapid user might accidentally click on the ad instead of the intended "Exit/Cancel".

static information of the layout in a single UI state [37], detection of the latter presents several challenges, notably for:

- *Dynamically exercising ad views in a UI state, achieving scalability, and ensuring good coverage in transitions between UI states*: A UI state is a running page that contains several visual views/elements, also referred to as controls in Android documentation. Because dynamic interaction frauds involve sequences of UI states, a detection scheme must consider the *transition between UI states*, as well as background resource consumption such as network traffic. For example, in order to detect the ad fraud case presented in Figure 1, one needs to analyse both current and next UI states to identify any ad view that is placed on top of buttons and which could thus entice users to click on ads unexpectedly. Exercising apps to uncover such behaviours can however be time-consuming: previous work has shown that it takes several hours to traverse the majority UI states of an app based on existing Android automation frameworks [33].
- *Automatically 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 [37]), 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 [18] 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 straightforward 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:

- *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.
- *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 identified 335 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[2]

## 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.

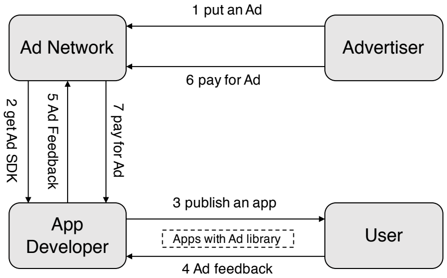


**Figure 2: An overview of the mobile advertising ecosystem.**

### 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

[2] We have created a new GitHub account and anonymized the author-revealing information.

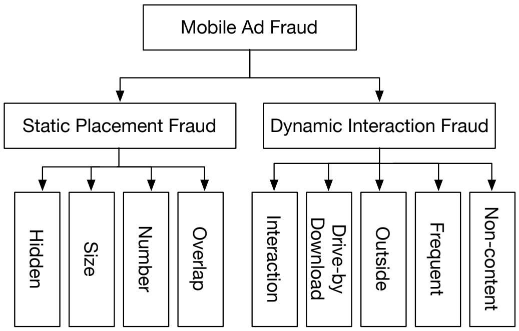


**Figure 3: A taxonomy of mobile ad frauds.**

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 [21, 26, 49] is regarded as ad frauds by the Google AdMob ad network. Besides, popular app markets [41, 42, 49, 60] 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-noticeably 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. Furthermore, dynamic interaction frauds have even not been explored to the best of our knowledge.

To build the taxonomy of Android ad frauds, we investigate in this work: (1) the usage policies provided by popular ad libraries [22, 26], (2) the developer policies provided by official Google Play market [49] and popular third-party app markets, including Wandoujia (Alibaba App) Market [60], Huawei App Market [41] and Tencent Myapp Market [42]. (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.

*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, and the number of ad views, 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 users 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 [26, 60].
(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 [60], allowing the ads to be viewed normally by users [23]. 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 [22, 60]. 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 [22, 26, 60]. 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 occur 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, 26, 60].
(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, 60]. Such behavior often heavily impacts user experience, and in most cases, drive-by downloads cannot even be readily cancelled.

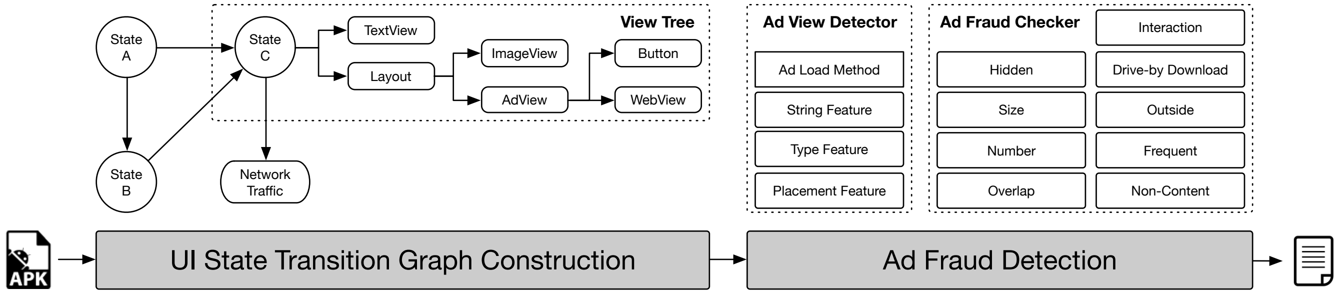


**Figure 4: Overview of FraudDroid.**

(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, 26, 41, 60]. 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, 26].

(9) The ***Non-content Ad* fraud**. To maximize the number of ad impressions and trick users into unintended clicks, app 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 [26].

## 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*.

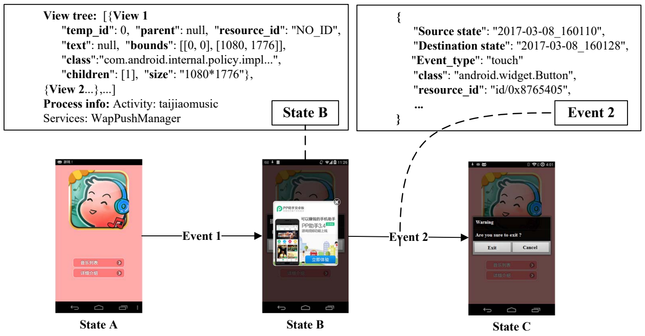
To efficiently search for ad frauds, one possible step before sending apps to FraudDroid is to focus on such apps that have included ad libraries. To this end, FraudDroid integrates a pre-processing step, which stops the analysis if the input app does not leverage any ad libraries, i.e., there will be 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` [34]. Then, we leverage LibRadar [36, 38], 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 [47]. 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*, 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, view class, source state) that is associated with the transition between two states.

Figure 5 illustrates an example of UI state transition graphs, 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 UI state transition graph 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.



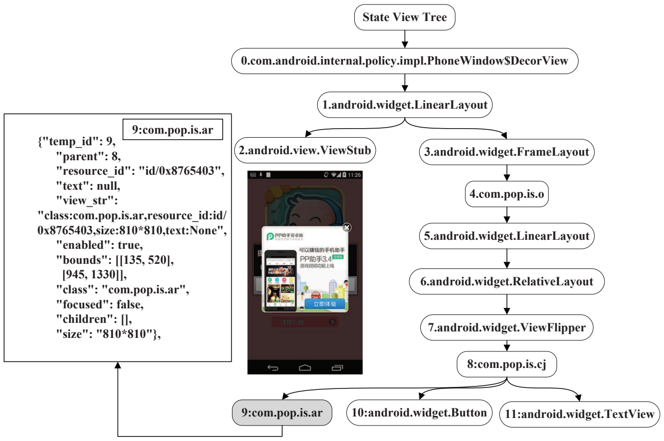
```
View tree:  [{View 1
    "temp_id": 0,  "parent": null,  "resource_id": "NO_ID",
    "text": null,  "bounds": [[0, 0], [1080, 1776]],
    "class":"com.android.internal.policy.impl...",
    "children": [1],  "size": "1080*1776"},
{View 2...},...]
Process info: Activity: taijiaomusic
Services: WapPushManager
```

```
{
    "Source state": "2017-03-08_160110",
    "Destination state": "2017-03-08_160128",
    "Event_type": "touch"
    "class": "android.widget.Button",
    "resource_id": "id/0x8765405",
    ...
}
```

**Figure 5: Simplified illustrative example of a UI state transition graph.**

*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 [47] 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 an ad-first exploration 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 at providing 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 (`810*810`) 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.


State View Tree
0.com.android.internal.policy.impl.PhoneWindow$DecorView
1.android.widget.LinearLayout
2.android.view.ViewStub
3.android.widget.FrameLayout
4.com.pop.is.o
5.android.widget.LinearLayout
6.android.widget.RelativeLayout
7.android.widget.ViewFlipper
8:com.pop.is.cj
9:com.pop.is.ar
10:android.widget.Button
11:android.widget.TextView
9:com.pop.is.ar

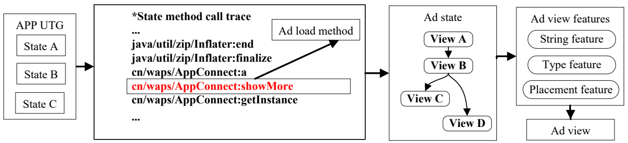
```
{"temp_id": 9,
     "parent": 8,
     "resource_id": "id/0x8765403",
     "text": null,
     "view_str":
"class:com.pop.is.ar,resource_id:id/
0x8765403,size:810*810,text:None",
     "enabled": true,
     "bounds": [[135, 520],
       [945, 1330]],
     "class": "com.pop.is.ar",
     "focused": false,
     "children": [],
     "size": "810*810"},
```

**Figure 6: An example of view tree for interactive fraud.**

*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.



```
*State method call trace
...
java/util/zip/Inflater:end
java/util/zip/Inflater:finalize
cn/waps/AppConnect:a
cn/waps/AppConnect:showMore
cn/waps/AppConnect:getInstance
...
```

Step 1: Finding the ad state by ad load method

Step 2: Finding the ad view by ad view features

**Figure 7: Detection of ad views** (UTG stands for UI state transition graph)**.**

*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 [47]. 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 name, can be indicative of whether it is an ad view. Most ad libraries indeed 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, gadget, load, adapter, adobe, etc.), we rely on a whitelist of English words containing "ad" based on the Spell Checker Oriented Word Lists (SCOWL) [32]. 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.

**Table 1: Ad features.**

| Aspects | Attribute | Value |
|---|---|---|
| String | Resource_id | AdWebview, AdLayout, ad_container, fullscreenAdView, FullscreenAd, AdActivity, AppWallActivity, etc. |
| Type | Class | ImageView, WebView, ViewFlipper |
| placement | Size (Bounds) | 620*800[Center], 320*50[Top or Bottom], 1776*1080[Full screen], etc. |

- **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" [17], "WebView" [19] and "ViewFlipper" [20], 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[26], 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 the view tree extracted 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 UI State Transition Graph, the *FraudChecker* module can assess their tag information to check whether a fraudulent behaviour can be spotted. This module of FraudDroid 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.* To detect such frauds, *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 [0.004, 0.005] 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 combined space occupied by all ads in a UI state must not exceed the space reserved to app content. When there are several ad views in a UI state along side app content (i.e., all as view tree leafs), 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 UI state transition graph 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 transition graph 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 this behaviour as an interactive fraud.

*Drive-by download Ad.* This fraud consists in triggering unwanted downloads (of apps or other files) without any confirmation by users after an ad is clicked on. *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 transition graph 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.* To detect such frauds, *FraudChecker* keeps track of all Activity names that are associated with the app. By going through all UI states, it checks whether a UI state contains an ad view that has its associated Activity name different from any of the expected ones. This indicates that the ad is shown outside the app environment.

*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[3] in the UI state transition graph, 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).

*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* [54] and *The Bro Network Security Monitor* [50], respectively, to harvest and analyse the network traffic.

[3] This number is configurable to FraudDroid.

# 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 a real physical device, namely a Nexus 5 smartphone. 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 [55].

## 4.1 Detection Accuracy

Accuracy is assessed for both the detection of ad views by *AdViewDectector* 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:

(1) Our first benchmark set, *AdViewBench*, includes 500 apps that were manually exercised to label 4,403 views among which 211 are ad views.

(2) Our second benchmark set, *AdFraudBench*, includes 100 ad-supported apps, 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 [59], 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 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].

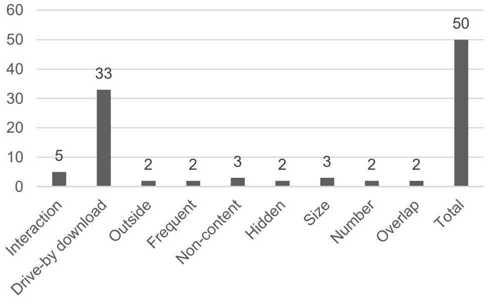


**Figure 8: Distribution of the 50 labelled apps via their ad fraud types.**

### 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. Table 2 summarizes the results on precision and recall of *AdViewDetector*. With over 90% of precision and recall, FraudDroid is accurate in detecting ad views, an essential step towards efficiently detecting ad frauds.

**Table 2: Confusion matrix of AdViewDetector on the *AdViewBench* benchmark.**

| | | **Predicted views** | |
|---|---|---|---|
| | | Ad-contained | Ad-free |
| **Labelled views** | Ad-contained | 197 True Positives | 11 False Negatives |
| | Ad-free | 14 False Positives | 4181 True Negatives |

We further investigate the false positives and false negatives results by the *AdViewDetector* module. On the one hand, we found by analysing the UI states collected during app automation that some ads are not successfully displayed: the library fails to load ad content due to network connection failures or loading time-outs. Nevertheless, because of the calls to ad loading methods, FraudDroid will anyway flag those views (without ads displayed) as ad views, which however are not considered as such during our manual benchmark building process, resulting in false positive results. On the other hand, we have found that some UI states, although they include ad views, do not include calls to ad loading methods: the ad views were inherited from prior states, and some events (e.g., such as drag or scroll events) that triggered the transition to new states do not cause a reloading of views. Such cases result in false negatives by *AdViewDetector*, which misses to report any ad view in the UI state.

### 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 and 4 false negatives, 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.**

| | | **Predicted frauds** | |
|---|---|---|---|
| | | Ad fraud | w/o Ad fraud |
| **Labelled frauds** | Ad fraud | 46 True Positives | 4 False Negatives |
| | w/o Ad fraud | 3 False Positives | 47 True Negatives |

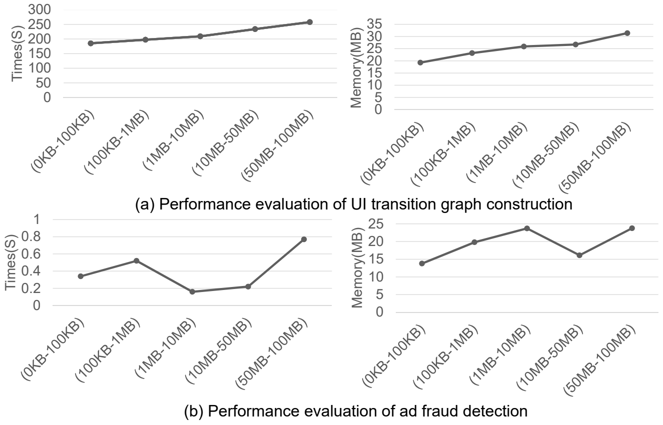
We further investigate the false positive/negative detection cases to understand the root causes of FraudDroid's failures. We find that all such cases are caused by misclassification results by the *AdViewDetector* module. FraudDroid misses ad frauds because the UI state is not marked as relevant since 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] For example, app `az360.gba.jqrgsseed` is found to implement both an interaction and a drive-by download fraud.

## 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[5]: 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 [10, 37], 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.



(a) Performance evaluation of UI transition graph construction

(b) Performance evaluation of ad fraud detection

**Figure 9: Performance evaluation of FraudDroid.**

## 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, Baidy, Tencent, PP, Mgyapp and 360 Mobile. Using LibRadar [38], 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* [12] 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*, which, to the best of our knowledge, we are the first to investigate with this work. 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 impressions and clicks, leading to more revenues.

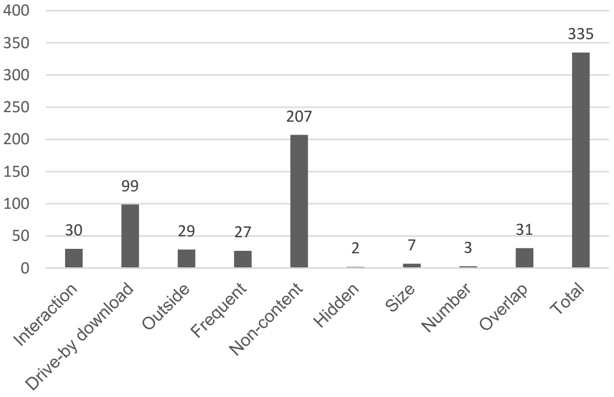


**Figure 10: Ad fraud distribution by type.**

4.3.3 *Ad Fraud Distribution by App Markets.* Table 4 enumerates the distribution of detected ad-fraud apps based on the markets where they were collected. The *PP Assistant* market includes the highest rate (4.84%) of ad frauds among its ad-supported apps in our study. Huawei Market includes the lowest rate (0.87%) of ad frauds. Finally the ad fraud rate in Google Play is slightly lower than the average across markets.

**Table 4: Dataset from eight major Android app markets.**

| App Market | #Ad Apps | #Fraud | Percentage |
|---:|:---:|:---:|:---:|
| Google Play | 3,630 | 94 | 2.59% |
| 360 Mobile Assistant | 1,012 | 25 | 2.47% |
| Baidu Mobile Assistant | 1,588 | 55 | 3.46% |
| Huawei | 1,145 | 10 | 0.87% |
| Xiaomi | 1,295 | 29 | 2.24% |
| Tencent Yingyongbao | 843 | 18 | 2.14% |
| Mgyapp | 1,288 | 46 | 3.57% |
| PP Assistant | 1,199 | 58 | 4.84% |
| ***Total*** | ***12,000*** | ***335*** | ***2.79%*** |

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

[5] E.g., for the scale 100KB, the apps must be of size between 80KB and 120KB.

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.

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

| Ad network | #Ad fraud | %Percent | Ad network | #Ad fraud | %Percent |
|---|---|---|---|---|---|
| Admob | 113 | 4.52% | Adwhirl | 2 | 0.4% |
| Appbrain | 68 | 13.6% | Dianjin | 9 | 1.8% |
| Waps | 48 | 9.6% | Vpon | 1 | 0.2% |
| feiwo | 29 | 5.8% | Inmobi | 6 | 1.2% |
| BaiduAd | 10 | 2% | Apperhand | 8 | 1.6% |
| Anzhi | 7 | 1.4% | Startapp | 5 | 1% |
| Youmi | 8 | 1.6% | Mobwin | 2 | 0.4% |
| Doodlemobile | 4 | 0.8% | Jumptap | 1 | 0.2% |
| Adsmogo | 5 | 1% | Fyber | 1 | 0.2% |
| Kugo | 7 | 1.4% | Domob | 1 | 0.2% |

Interestingly, although Google Admob has published strict policies [26] on how ad views 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* [30], 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 [11, 12, 30]. This evidence suggests that ad networks should not only publish explicit policies to regulate the usage of their ad networks, but also introduce reliable means to detect policy-violated cases for mitigating the negative impact on users and advertisers.

*4.3.5 Detection Results of VirusTotal.* We have uploaded all the detected 335 fraudulent apps to VirusTotal to explore how many of them could be flagged by existing anti-virus engines. There are 174 apps (51.9%) labelled as *AdWare* by at least one engine. Some apps are even flagged by more than 30 engines. For example, app "com.zlqgame.yywd" was flagged by 33 engines [58] and app "com.scanpictrue.main" was flagged by 30 engines [57]. However, roughly half of these fraudulent apps are not flagged, which suggests that ad fraud behaviours cannot be sufficiently identified by existing engines, especially for dynamic interaction frauds as 87.5% (141 out of 161) of these un-flagged apps contain only such frauds.

*4.3.6 Case Studies.* We now present real-world case studies to highlight the capability of FraudDroid for detecting a wide range of fraud types. Figure 11 illustrates nine examples of ad-fraud apps that are found by FraudDroid from Google Play and third-party markets.

(1) An *Interaction Ad* fraud was spotted in app `com.android.yatree.taijiaomusic` where an ad view pops up spuriously above the exit dialogue.
(2) A *Drive-by Download Ad* fraud is found in app `com.hongap.slider` where a download starts once the ad view is clicked.
(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 four 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.

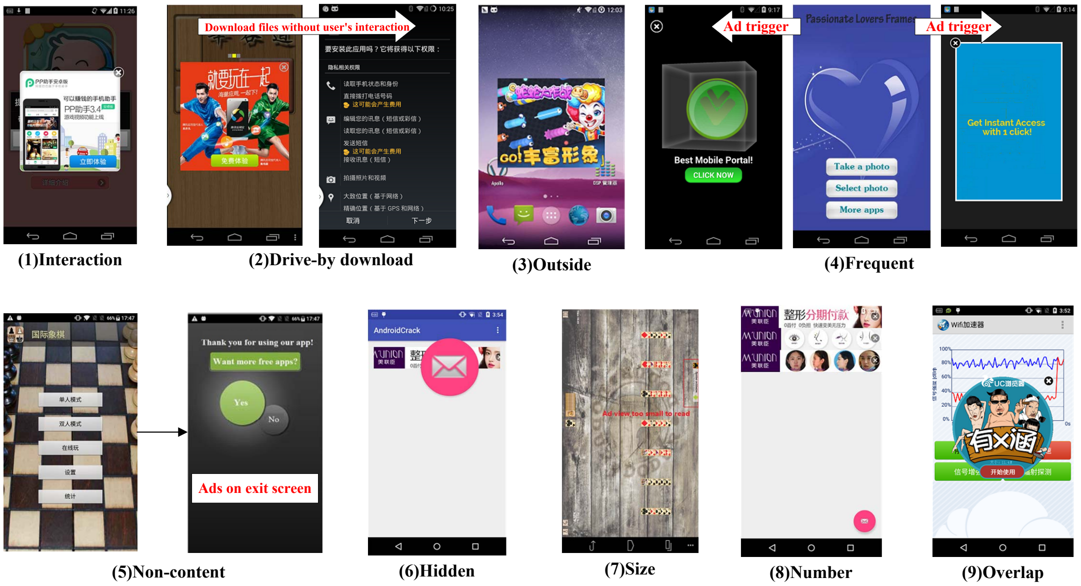
***Other types of ad frauds.*** Although we have considered nine types of ad frauds, including five new types of *dynamic interaction frauds*, which have not been explored before, our taxonomy may still be incomplete since it was built based on current policies and samples available. Other emerging types of ad frauds may have been missed. Nevertheless, the UI transition graph built by FraudDroid 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 two lines of research: automated app testing and ad fraud detection.

### 6.1 Automated App Testing

Automated app testing has been widely adopted for exploring apps at runtime. Several Android automation frameworks such as Hierarchy Viewer [15], UIAutomator [16] and Robotium [28] have been developed to facilitate app testing. One critical step to perform



(1)Interaction (2)Drive-by download (3)Outside (4)Frequent

(5)Non-content (6)Hidden (7)Size (8)Number (9)Overlap

**Figure 11: Case studies of ad-fraud apps.**

automated app testing is to generate reliable test inputs [10]. The most straightforward means to achieve that is to follow a random strategy (i.e., the so-called random testing), where the test inputs are generated randomly. Indeed, various tools such as Monkey [14], Dynodroid [39], Intent Fuzzer [51] and DroidFuzzer [62] have been introduced to our community following this strategy. However, random testing is likely to generate redundant events that may lead to wastes of resources. It is also 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.

Because of the aforementioned reasons, researchers have introduced a new type of strategy, namely model-based, to generate test inputs for automatically exploring Android apps. Indeed, many research-based tools such as GUIRipper [2], MobiGUITAR [3], A3E [7], SwiftHand [9], PUMA [29] have been proposed to perform model-based testing of Android apps. These tools usually leverage finite-state machines, e.g., activities as states and events as transitions, to model the app and subsequently implement a depth-first search (DFS) or breadth-first search (BFS) strategy to explore the states for generating more effective test inputs w.r.t. code coverage. More advanced tools such as EvoDroid [40] and ACTEve [4] use more 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 ad frauds. DroidBot [35] is a lightweight test input generator, which is able to interact with Android apps without instrumentation. It allows users to integrate their own strategies for different scenarios. Compared with them, we have implemented a more sophisticated ad view exploration strategy for automated scalable ad fraud detection in this work.

## 6.2 Ad Fraud Detection

Ad fraud in general is the No. 1 cybercrime counted in terms of revenues generated, ahead of tax-refund fraud [52]. 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 [44, 45] or search engine query logs [63], characterizing click frauds [1, 8, 46] and analysing profit models [43]. 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.

Existing studies on mobile ad frauds have attempted to identify ad-fraud apps where the fraudulent behaviours can be spotted statically (the so-called *static placement frauds*). As examples, Liu *et al.* [37] have investigated static placement frauds on Windows Phone via analysing the layouts of apps. Crussell *et al.* [13] have developed an approach for automatically identifying click frauds. Their approach is implemented mainly in three steps: (1) building HTTP request trees, (2) identifying ad request pages using machine learning, and (3) detecting clicks in HTTP request trees using heuristic rules. 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, we have introduced five new types of such frauds that have not yet been explored by our community.

## 7 CONCLUSION

Through an exploratory study, we characterize ad frauds by investigating policies set up by ad networks. 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 the 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.

## REFERENCES

[1] Sumayah A. Alrwais, Alexandre Gerber, Christopher W. Dunn, Oliver Spatscheck, Minaxi Gupta, and Eric Osterweil. 2012. Dissecting Ghost Clicks: Ad Fraud via Misdirected Human Clicks. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/2420950.2420954

[2] Domenico Amalfitano. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the Ieee/acm International Conference on Automated Software Engineering*. 258–261.

[3] D Amalfitano, A Fasolino, P Tramontana, and B Ta. 2014. MobiGUITAR – A Tool for Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2014), 1–1.

[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *ACM Sigsoft International Symposium on the Foundations of Software Engineering*. 1–11.

[5] AppBrain. 2018. Free vs. paid Android apps. (2018). https://www.appbrain.com/stats/free-and-paid-android-applications

[6] China Communications Standards Association. 2017. Mobile Intelligent Terminal Malicious Push Information To Determine The Technical Requirements. (2017). http://www.ccsa.org.cn/tc/baopi.php?baopi_id=5244

[7] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. *Acm Sigplan Notices* 48, 10 (2013), 641–660.

[8] Tommy Blizard and Nikola Livic. 2012. Click-fraud Monetizing Malware: A Survey and Case Study. In *Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software (MALWARE) (MALWARE '12)*. IEEE Computer Society, Washington, DC, USA, 67–72. https://doi.org/10.1109/MALWARE.2012.6461010

[9] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI testing of android apps with minimal restart and approximate learning. In *ACM Sigplan International Conference on Object Oriented Programming Systems Languages & Applications*. 623–640.

[10] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *Ieee/acm International Conference on Automated Software Engineering*. 429–440.

[11] RILEY CILLIAN. 2018. Google Play App: Font studio. (2018). Retrieved March 9, 2018 from https://play.google.com/store/apps/details?id=com.rcplatform.fontphoto

[12] CrazyFunnyApp. 2018. Google Play App: Ancient Tomb Run. (2018). Retrieved March 9, 2018 from https://play.google.com/store/apps/details?id=com.CrazyRunGame1.Templeqqq

[13] Jonathan Crussell, Ryan Stevens, and Hao Chen. 2014. Madfraud: Investigating ad fraud in android applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 123–134.

[14] Android Developers. 2017. The Monkey UI android testing tool. (2017). http://developer.android.com/tools/help/monkey.html

[15] Android Developers. 2017. Profile Your Layout with Hierarchy Viewer. (2017). https://developer.android.com/studio/profile/hierarchy-viewer.html

[16] Android Developers. 2017. UIautomator. (2017). https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html

[17] Android Developers. 2018. App Widgets. (2018). Retrieved February 12, 2018 from https://developer.android.com/guide/topics/appwidgets/index.html

[18] Android Developers. 2018. Layouts. (2018). Retrieved February 12, 2018 from https://developer.android.com/guide/topics/ui/declaring-layout.html

[19] Android Developers. 2018. Managing WebViews. (2018). Retrieved February 12, 2018 from https://developer.android.com/guide/webapps/managing-webview.html

[20] Android Developers. 2018. ViewFlipper. (2018). Retrieved February 12, 2018 from https://developer.android.com/reference/android/widget/ViewFlipper.html

[21] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Guoai Xu, and Shaodong Zhang. 2018. How Do Mobile Apps Violate the Behavioral Policy of Advertisement Libraries?. In *Proceedings of the 19th International Workshop on Mobile Computing Systems &#38; Applications (HotMobile '18)*. 75–80.

[22] DoubleClick. 2017. DoubleClick Ad Exchange Program Policies. (2017). https://support.google.com/adxseller/answer/2728052?hl=en

[23] Firebase. 2017. Banner Ads. (2017). https://firebase.google.com/docs/admob/android/banner

[24] Anne Freier. 2017. More fraudulent apps detected on GoogleâĂŹs Play Store. (2017). Retrieved February 12, 2018 from http://www.mobyaffiliates.com/blog/more-fraudulent-apps-detected-on-googles-play-store/

[25] MARIA GERSEN. 2016. MOBILE AD FRAUD: DEFINITION, TYPES, DETECTION. (2016). https://clickky.biz/blog/2016/12/mobile-ad-fraud-definition-types-detection/

[26] Google. 2017. Google AdMob & AdSense policies. (2017). https://support.google.com/admob#topic=2745287

[27] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 101–112.

[28] Robotium Developers Group. 2017. Robotium. (2017). https://github.com/RobotiumTech/robotium

[29] Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *International Conference on Mobile Systems, Applications, and Services*. 204–217.

[30] Heaveen. 2018. Google Play App: Thermo. (2018). Retrieved March 9, 2018 from https://play.google.com/store/apps/details?id=com.heaven.thermo

[31] Brandon Jones. 2017. Google Breaks Up Biggest Ad Fraud on Play Store. (2017). Retrieved February 12, 2018 from http://www.psafe.com/en/blog/google-breaks-biggest-ad-fraud-play-store/

[32] kevina. 2018. Spell Checker Oriented Word Lists. (2018). http://wordlist.aspell.net/

[33] Kyungmin Lee, Jason Flinn, Thomas J Giuli, Brian Noble, and Christopher Peplin. 2013. AMC: verifying user interface properties for vehicular applications. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 1–12.

[34] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. An Investigation into the Use of Common Libraries in Android Apps. In *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*.

[35] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 23–26.

[36] LibRadar. 2018. Detecting Third-party Libraries Used in Android Apps. (2018). https://github.com/pkumza/LibRadar

[37] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. 2014. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps.. In *NSDI*. 57–70.

[38] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. Libradar: Fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 653–656.

[39] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 224–234.

[40] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *The ACM Sigsoft International Symposium*. 599–609.

[41] Huawei Market. 2018. Huawei Market App Developer Policy. (2018). http://developer.huawei.com/consumer/cn/devservice/develop/mobile

[42] Tencent Myapp Market. 2018. Tencent Myapp Market App Developer Policy. (2018). http://open.qq.com/

[43] Damon McCoy, Andreas Pitsillidis, Grant Jordan, Nicholas Weaver, Christian Kreibich, Brian Krebs, Geoffrey M. Voelker, Stefan Savage, and Kirill Levchenko. 2012. PharmaLeaks: Understanding the Business of Online Pharmaceutical Affiliate Programs. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)*. 1–1.

[44] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2007. Detectives: Detecting Coalition Hit Inflation Attacks in Advertising Networks Streams. In *Proceedings of the 16th International Conference on World Wide Web (WWW '07)*. 241–250.

[45] Ahmed Metwally, Fatih Emekçi, Divyakant Agrawal, and Amr El Abbadi. 2008. SLEUTH: Single-pubLisher Attack dEtection Using correlaTion Hunting. *Proc. VLDB Endow.* 1, 2 (2008), 1217–1228.

[46] Brad Miller, Paul Pearce, Chris Grier, Christian Kreibich, and Vern Paxson. 2011. What's Clicking What? Techniques and Innovations of Today's Clickbots. In *Proceedings of the 8th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'11)*. 164–183.

[47] Suman Nath. 2015. Madscope: Characterizing mobile in-app targeted ads. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 59–73.

[48] Suman Nath, Felix Xiaozhu Lin, Lenin Ravindranath, and Jitendra Padhye. 2013. SmartAds: bringing contextual ads to mobile apps. In *Proceeding of the International Conference on Mobile Systems, Applications, and Services*. 111–124.

[49] Google Play. 2018. Google Play Developer Policy: Monetisation and Ads. (2018). https://play.google.com/intl/en-GB_ALL/about/monetization-ads/ads/ad-id/

[50] The Bro Project. 2018. The Bro Network Security Monitor. (2018). Retrieved February 12, 2018 from https://www.bro.org

[51] Raimondas Sasnauskas and John Regehr. 2014. Intent fuzzer: crafting intents of death. In *Joint International Workshop on Dynamic Analysis*. 1–5.

[52] Tommie Singeton and AICPA. 2013. The Top 5 Cyber Crimes. (2013). https://www.aicpa.org/content/dam/aicpa/interestareas/forensicandvaluation/resources/electronicdataanalysis/downloadabledocuments/top-5-cybercrimes.pdf

[53] Kevin Springborn and Paul Barford. 2013. Impression fraud in online advertising via pay-per-view networks. In *Usenix Conference on Security*. 211–226.

[54] Tcpdump-workers. 2018. Tcpdump. (2018). Retrieved February 12, 2018 from http://www.tcpdump.org

[55] Timothy Vidas and Nicolas Christin. 2014. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 447–458.

[56] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A Measurement Study of Google Play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'14)*. 221–233.

[57] VirusTotal. 2018. Detection Result of com.scanpictrue.main. (2018). https://www.virustotal.com/#/file/6dc84cb4b9ad3c47bb69797833a48d4405500394e594c582ae2ef91d66d41d86/detection

[58] VirusTotal. 2018. Detection Result of com.zlqgame.yywd. (2018). https://www.virustotal.com/#/file/7e59d0219ff4e465dcb083e12bacfbb0462c7b7a98af374e61996028bac496db/detection

[59] VirusTotal. 2018. VirusTotal. (2018). https://www.virustotal.com/

[60] Wandoujia. 2018. Wandoujia (Ali App) Developer Policy. (2018). http://aliapp.open.uc.cn/wiki/?p=140

[61] Haoyu Wang, Zhe Liu, Yao Guo, Xiangqun Chen, Miao Zhang, Guoai Xu, and Jason Hong. 2017. An Explorative Study of the Mobile App Ecosystem from App Developers' Perspective. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. 163–172.

[62] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *International Conference on Advances in Mobile Computing & Multimedia*. 68.

[63] Fang Yu, Yinglian Xie, and Qifa Ke. 2010. SBotMiner: Large Scale Search Bot Detection. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining (WSDM '10)*. 421–430.