Unveiling Collusion-Based Ad Attribution Laundering Fraud: Detection, Analysis, and Security Implications

Tong Zhu
tongzhu@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Chaofan Shou
shou@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Zhen Huang
xmhuangzhen@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Guoxing Chen*
guoxingchen@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Xiaokuan Zhang
xiaokuan@gmu.edu
George Mason University
Fairfax, Virginia, USA

Yan Meng
yan_meng@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Shuang Hao
shao@utdallas.edu
University of Texas at Dallas
Richardson, Texas, USA

Haojin Zhu*
zhu-hj@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

ABSTRACT

In recent years, the growth of mobile advertising has been driven by in-app programmatic advertising and technologies like Real-Time Bidding (RTB). However, this growth has also led to an increase in ad fraud, such as click injection, background ad activity, etc. While existing studies have primarily concentrated on ad fraud within individual apps or devices, this paper introduces a new form of collusion-based ad fraud, named ad attribution laundering fraud (*ALF*). *ALF* involves multiple apps collaborating to deceive advertisers by misrepresenting the app where ads are displayed. The collusion-based approach allows lower-quality apps to exploit the reputable identities of seemingly legitimate apps. This deceives advertisers or ad networks into believing that the advertisements they place are reaching potentially valid end-users on the legitimate app. The seemingly legitimate ad events and ad attribution procedures employed by individual apps in such attacks can evade detection by existing tools.

To detect *ALF*, we design and implement the *first* detection framework, AlfScan. It overcomes two challenges to extract apps' identities from diverse and obfuscated apps using both static and dynamic analysis techniques, then cross-check the identities to identify *ALF*. We evaluate AlfScan on a 200-app ground truth dataset, and it achieves 92% precision and 92% recall. We use AlfScan to conduct a large-scale analysis on 91,006 apps and identify 4,515 unique fraudulent apps and 1,483 fraudulent clusters, exposing patterns among fraudulent developers and revealing reliability issues in third-party app development frameworks. We also find that through *ALF*, fraudulent apps can generate invalid ad traffic that is 2.43 times to 33.33 greater than the ad traffic they would normally generate. After reporting our findings to 15 ad network companies, 4 companies expressed interest in testing AlfScan. In particular, we have submitted 344 apps to the Unity ad team, and they have confirmed that the apps were involved in fraudulent activities.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Ad Network; Mobile Ad Fraud; Ad Attribution Laundering Fraud; Static Analysis; Dynamic Analysis

ACM Reference Format:
Tong Zhu, Chaofan Shou, Zhen Huang, Guoxing Chen, Xiaokuan Zhang, Yan Meng, Shuang Hao, and Haojin Zhu. 2024. Unveiling Collusion-Based Ad Attribution Laundering Fraud: Detection, Analysis, and Security Implications. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670314

1 INTRODUCTION

In recent decades, in-app programmatic advertising has been the driving force behind the growth of mobile advertising, and the ad market is estimated to increase by USD 188.92 billion from 2021 to 2026 [50]. This is due to innovative ad technologies such as Real-Time Bidding (RTB), which have enabled the automatic buying and selling of ad inventory (the available advertising space in apps). A typical workflow of RTB is as follows: Firstly, each ad event (*e.g.*, ad impressions, ad clicks, app installations, etc.) in the ad inventory initiates the generation of ad traffic, which is subsequently dispatched to the advertising network; Secondly, the network tracks and analyzes this ad traffic to determine the source app that attributes to specific ad events. Finally, the network conducts settlements with

*Guoxing Chen and Haojin Zhu are the corresponding authors.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670314

app developers based on factors such as impression frequency or the number of impressions/clicks.

Propelled by financial motives, attackers engage in mobile ad fraud to steal advertising revenue. Both academia and industry have identified a series of mobile ad frauds. They fall into the following two dimensions: *fake ad event* and *false ad attribution*. (1) The fake ad events include fake clicks [13, 68, 14, 41, 48], fake installation [69, 62, 20], invalid impressions [30, 34, 18, 37, 17, 66, 64, 2, 48], or problematic ad content [66, 64, 2, 55, 63], where fraudulent apps create deceptive or counterfeit interactions between ads and users. (2) The false attribution of ads includes installation hijacking [8], and app misrepresentation fraud [25], where fraudulent apps manipulate the attribution data to falsely claim credit for legitimate ad events on other apps. However, existing detection schemes consider fraudulent activities within each app separately and resort to rule-based mechanisms [30], whitelists [25], and models trained by legitimate apps [68, 37], to determine whether each app is malicious or not, individually. In this paper, we ask the following research question: *Can a cluster of seemingly legitimate apps collude with each other to evade existing detection methods and engage in ad fraud?*

A novel collusion-based mobile ad fraud. We identified a collusion-based mobile ad fraud, named ad attribution laundering fraud (*ALF*). *ALF* involves genuine ad events and user interactions in each app, which makes it difficult to detect. *ALF* attributes the advertising traffic of a cluster of apps to a single app through collusion. Lower-quality apps can exploit the reputable identities of legitimate apps based on collusion to boost ad revenue. This results in the sale of low-valued or entirely illegitimate inventory across various apps to advertisers or networks, falsely conveying that the inventory is reaching potentially valid end-users on the legitimate app. In *ALF*, ad events and ad attribution processes on individual apps appear legitimate, distinguishing it from traditional ad fraud tactics involving fake ad events (*e.g.*, fake clicks) or manipulation of ad attribution processes (*e.g.*, installation hijacking). Consequently, traditional ad fraud detection tools such as ClickScanner[68] encounter challenges in identifying *ALF*, necessitating detection across multiple apps to identify shared identities among different apps. *ALF* undermines the integrity and transparency of the ad attribution ecosystem, violating industry authoritative guidelines outlined by the Media Rating Council (MRC) [40], Interactive Advertising Bureau (IAB) [28], and Google [23]. Our research is the *first* to unveil the collusion-based ad fraud in the mobile advertising ecosystem.

In this study, we take the first step toward detecting *ALF* by diving into the fundamental RTB attribution mechanism. Our key observation is that collaborative attackers leave identifiable traces that violate the rule of one-to-one association between apps and AppIDs. In detail, ad networks rely on the **unique identifier**[24], known as AppID, to precisely identify the specific app that is displaying an advertisement. This identification is critical for assessing the effectiveness of advertisements, attributing ads, and determining the distribution of ad revenue. Developers are required to maintain a **one-to-one** correlation between apps and AppIDs during ad integration, as emphasized by prominent networks like Google Admob, etc.[24, 43], which ensures that ads delivered through the RTB process are accurately linked to the respective apps. In *ALF*, when a high-profile APP_A colludes with a low-quality or illegitimate APP_B (*e.g.*, porn apps, malware, etc.), APP_B violates the one-to-one association rule by *collaboratively* selling ad inventory under the *same* AppID as APP_A. The collusion enables APP_B to monetize its subpar traffic under the guise of APP_A and enables the developers of both APP_A and APP_B to share the ad revenue. When considering the ad events and ad attribution process individually, these apps appear harmless, rendering existing ad fraud detection tools ineffective.

To detect the *ALF*, the key problem we need to tackle is to detect whether an AppID violates the aforementioned one-to-one association rule and is used across different apps. We observe that apps may have hard-coded AppIDs in their source code, or dynamically-loaded AppIDs from remote servers during runtime. To systematically extract these two types of AppIDs, we must address the following challenges.

- **Analyzing apps with diverse, evolving, and obfuscated ad SDKs in a general and robust way.** Extracting hard-coded AppIDs via static analysis involves tracing data dependencies from potential locations in code, such as tracking developer APIs provided by ad software development kits (SDKs), which allow developers to incorporate AppIDs into their code during ad integration. However, relying solely on identifying developer APIs is not foolproof. First, diverse and evolving ad SDKs often introduce new and different APIs for AppID retrieval, complicating the API tracking process. Second, in some fraudulent apps, the fraudulent activities stem from compromised ad SDKs manipulated by fraudsters, wherein they inject their own AppIDs. When developers are misled to use such SDKs, any AppID defined by developer APIs will be replaced with the fraudster's AppID within the SDK. As a result, tracking the values entered in developer APIs becomes futile. Moreover, such SDKs are often obfuscated, making it challenging to establish a robust code pattern to facilitate data flow analysis.
- **Imitating user interaction sequences to reach deep states inside targeted apps.** Initiating the ad-serving process by triggering ads is crucial, as dynamic loading of AppIDs frequently takes place in this phase. Triggering the ads and extracting dynamically loaded AppIDs often require complex and specific user interaction *sequences* such as scrolling a specific page and then clicking on a specific button. Existing dynamic analysis tools [46, 59, 58] treat a user interaction as an independent random input instead of a sequence of dependent inputs. As a result, they either fail to trigger ads hidden in a deep state, or require a significant amount of time for analysis.

AlfScan. To address these challenges, we propose AlfScan, the first automatic framework for the detection of *ALF* behaviors in apps. We unearth a fundamental step of *ALF* – HTTP/HTTPS data transmission with ad networks. AlfScan traces the code responsible for transmission to intercept hard-coded AppID, which is robust and agnostic to obfuscations and is applicable beyond the 15 SDKs analyzed in this study. To hunt remote manipulation of AppID and track them, we need to reach deep states to trigger ad operations via dynamic analysis. To solve this, AlfScan uses call-graph-based snapshotting to assist fuzzing, which significantly improves its performance. Finally, AlfScan cross-checks app package names and their associated AppIDs to identify instances where multiple

apps utilize the same AppID, indicating their collaborative efforts to increase the ad traffic deceptively.

To evaluate AlfScan, we assembled a ground truth dataset consisting of 100 benign and 100 malicious manually-labeled apps. The malicious apps were also confirmed by a third-party ad verification company. AlfScan achieves 92% precision and 92% recall on our ground truth dataset, demonstrating its effectiveness. We also conducted a large-scale analysis on 91,006 apps. AlfScan has uncovered a total of 4,515 unique fraudulent apps (4.96% of the analyzed apps). To further verify the accuracy of the results, we randomly sampled a set of 50 apps from those 4,515 apps for manual inspection. We found that each app among these 50 apps shared the same AppID with one or more collaborative apps from the 4,515 fraudulent apps, which validates our result. We also found that through *ALF*, fraudulent apps can generate invalid ad traffic that is 2.43 to 33.33 times greater than the ad traffic they would normally generate. Our exploration further unveiled distinctive patterns of fraudulent apps based on their certificates. Additionally, we exposed reliability issues associated with third-party app development frameworks and presented sophisticated fraud schemes in a detailed case study.

Contributions. We make the following contributions:

- *A new type of ad fraud.* We discover a new type of ad fraud, *ALF*, in which seemingly innocuous apps can collude with each other to conduct ad fraud. Each of them looks benign when checked individually using existing tools, thus bypassing state-of-the-art ad fraud defenses (Sec. 3).
- *A new detection framework.* We have implemented an automatic tool, AlfScan, to detect *ALF*. AlfScan utilizes control-flow-graph-based static analysis and snapshotting-based dynamic analysis to effectively unearth the key information from apps to perform detection (Sec. 4). AlfScan demonstrates high accuracy and efficiency in our 200-app ground truth dataset (Sec. 5).
- *Large-scale analysis.* AlfScan identifies 4,515 unique fraudulent apps from 91,006 apps. We observe that fraudulent apps have the capacity to inflate ad traffic, and we also discern patterns among fraudulent apps based on their certificates, as well as potential reliability issues within third-party app development frameworks (Sec. 6).
- *Case studies.* We investigate three sophisticated *ALF* schemes, including remote configurations that enable dynamic modification of AppID, disposable apps that shield high-profile apps from market bans, fraudsters that abuse ad network interfaces to circumvent SDK verification, etc (Sec. 7).

Responsible disclosure. We reported our findings to 15 ad network companies. We have received positive feedback from four teams (*i.e.*, Unity, Vungle, Baidu, and Tencent); they have expressed interest in AlfScan or would like to test it. Moreover, we submitted flagged AppIDs, associated with 344 malicious apps, to the Unity ad team, who confirmed their involvement in fraudulent activities. We discuss more details in Sec. 8.2.

Data availability. Our dataset can be found in https://github.com/Firework471/Ad-Attribution-Laundering-Apps-Dataset.

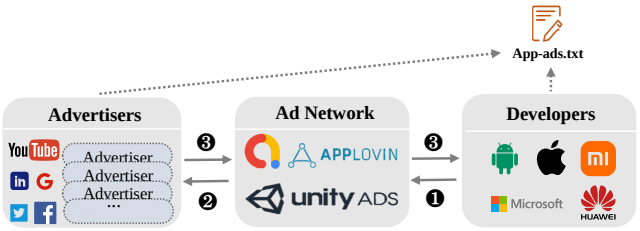

Fig. 1: Real-Time Bidding ecosystem: When a user uses a developer's app, the app employs the AppID provided by ad networks to label its ad inventory and puts this ad inventory for sale on the ad network in real time (Step ❶). Advertisers then bid for this ad inventory. Advertisement of the winning bid is notified to the advertisers (Step ❷), and ad networks initiate settlement procedures between the advertisers and the developers for the respective ad impression (Step ❸) based on the AppID. To mitigate fraud, advertisers use app-ads.txt of developers as an optional whitelist to ensure the legitimacy of specific inventory.

2 BACKGROUND ON REAL-TIME BIDDING

Real-time bidding (RTB) is a dynamic and automated auction-based system used in programmatic advertising, where ad inventory is bought and sold in real time[49]. Ad networks rely on the **unique identifier**[24], known as AppID, to identify the specific app where an ad is displayed. AppID is used for ad attribution, which eventually determines which app will receive the payment for displaying the ad. When developers integrate ads into their apps through ad SDKs, they include their app-specific AppIDs, which are provided by individual ad networks, within their code. Strictly adhering to the **one-to-one** association between apps and AppIDs during ad integration, as emphasized by major ad networks like Google Admob, etc.[24, 43], maintains the integrity and transparency of the advertising ecosystem, ensures accurate attribution of ads served through the RTB process. Developers enlist their ad inventory with their apps' AppIDs applied from the ad networks in real-time, which is aggregated by ad networks responsible for facilitating bidding on individual ad inventory. Advertisers strategically bid on the available inventory to target their ads effectively. This effort forms a supply chain for ads, with each entity playing a crucial role.

In detail, as shown in Fig. 1, when a user uses a developer's app, the associated ad inventory undergoes auction at an ad network, employing the AppID for unique identification (❶). Advertisers then submit bids based on their knowledge of the developers' apps for the ad inventory. The auction winner is subsequently notified (❷), and their ad impression is deployed to fill the ad inventory on the developer's app. Finally, the ad revenue is paid to the winners through ad networks based on the AppIDs to recognize their identities (❸).

To combat fraud, developers and advertisers employ an optional whitelist to verify the authority of the ad inventory. Developers create a publicly accessible whitelist (*i.e.*, app-ads.txt [25], an industry defense tool against ad fraud) on their websites, listing authorized *{AdNetwork, AppId}* pairs that they are using. Advertisers refer to the app-ads.txt of developers. They validate authorized sellers of specific

inventory and will only buy those ad inventory from authorized ad networks with authorized AppIDs.

Individual fraudulent apps may utilize the following two approaches to conduct fraudulent activities. For individual fraudulent apps that have no connections with high-profile apps, they can use the same AppID as high-profile app's in their ad traffic to disguise them as high-profile apps. In this case, advertisers will attribute this traffic directly to the high-profile app and settle the ad revenue with the high-profile app. Therefore, this strategy lacks profitability for fraudsters, providing little incentive for them to pursue this approach. On the other hand, they may disguise themselves as high-profile apps by, for instance, creating cloned apps [16] on other ad networks where the high-profile app does not participate, thereby obtaining another AppID to participate in ad bidding within those ad networks. However, this AppID can be regarded as unauthorized by app-ads.txt and this approach can be blocked in Step ❸.

3 OVERVIEW

As shown in Fig. 2, for the collaborative APP_B, it deceives advertisers by selling ad inventory under the same AppID within the same ad network alongside the high-profile APP_A. This AppID is authorized by the developers of the APP_A in its app-ads.txt. This creates an illusion that the ad is being delivered to authentic users on APP_A, which is actually displayed on APP_B. All the advertising revenue is credited to the APP_A, and it could subsequently share the profit with the APP_B.

3.1 Threat Model

In our threat model, advertisers and ad networks are *honest and legitimate*, but apps can be *fraudulent*.

- **Collaborative apps.** We consider the scenario where apps engage in collusion. A specific app successfully gets approval from the ad network and acquires an AppID. Other collaborative apps exploit this AppID to label their ad inventory in the mobile ad ecosystem. These collaborative apps may come from the same company or engage in collaboration, aiming to defraud advertising revenue.
- **Honest ad networks.** During the ad bidding process, we assume that ad networks trust the apps on standard app markets, i.e., such apps will faithfully report AppIDs sent from them to advertisers. Our assumption aligns with those made in prior studies [31] in this field.
- **Advertisers with the ability to acquire app-ads.txt.** Advertisers can obtain app-ads.txt published publicly on the website by developers to verify whether a certain AppID is authorized.

Objectives of the adversary. As shown in Fig. 2, the adversary (a fraudulent app) wants to make more profits via *ALF*, deceiving the user or the advertisers by selling ad inventory under the same AppID with other apps. There are mainly two objectives. 1) *Monetizing the illegitimate traffic.* Adversaries monetize the illegitimate ad traffic using the AppID associated with a high-profile app by misattribution. The advertising traffic generated by the illegitimate app (*e.g.*, APP_B in Fig. 2), which may be originally barred from passing the approval of the app market and advertising network, and unable to obtain a legal AppID for advertising activities, is now

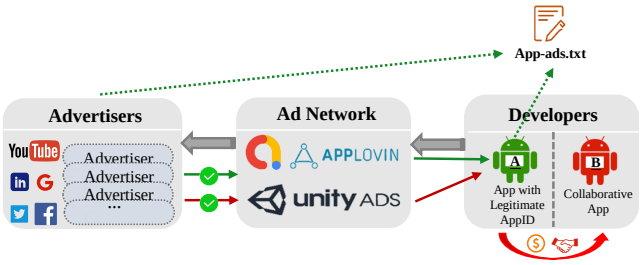

Fig. 2: Illustration of *ALF*'s workflow

erroneously attributed to the high-profile app APP_A. Consequently, the APP_B's subpar traffic can be monetized through the APP_A's AppID. 2) *Boosting the advertising value of relatively low-quality apps and amplifying its traffic.* Alternatively, distinct apps (*e.g.*, APP_A and APP_B in Fig. 2, which could have acquired different AppIDs respectively at lower ad prices), probably affiliated with the same entity, collude to concentrate all of their relatively low-quality app's traffic under a single AppID. The AppID belongs to APP_A and has received unauthorized advertising traffic from APP_B, creating a false impression of the associated APP_A as a very high-quality one with an inflated user base. This misconception boosts the advertising value of APP_A, leading advertisers to raise the ad price for its AppID in the ad auction. Simultaneously, traffic from APP_B, under the guise of the APP_A, siphons advertising revenue.

3.2 Challenges and Solutions

Our study aims to detect *ALF* across ad networks, focusing on identifying shared AppIDs among multiple apps. Extracting these AppIDs is challenging due to diverse formats and integration methods among obfuscated ad SDKs and the deep state of the apps. We present two motivating examples to illustrate specific challenges in systematic extraction.

3.2.1 Obfuscated and diverse ad SDKs. One running example in Fig. 3 illustrates a legitimate app using Google Admob SDK to access the developer's hard-coded AppID, which is pre-defined in the Manifest.xml file (*i.e.*, Line 2~3 in Fig. 3(a)). While in a fraudulent app, the sly fraudster manipulates the SDK code (*i.e.*, Line 4 in Fig. 3(b)), often obfuscated, to replace the developer's pre-defined AppID within the SDK, rendering the original developer pre-defined AppID ineffective.

Challenge C1: Analyzing apps with diverse, evolving, and obfuscated ad SDKs in a general and robust way. We first need to identify where a hard-coded AppID is used and subsequently backtrack its value. However, existing static analysis tools (*e.g.* [68, 52, 57]) fail to consistently and robustly identify these AppIDs due to their exclusive reliance on matching ad SDK developer APIs that receive AppIDs, which often proves inadequate.

Specifically, in this case, simply matching the signature of the SDK developer API receiving developer-defined AppID (*i.e.*, Manifest.xml file in this case) in other apps and backtracing the AppID value is impractical because this API's effectiveness could be undermined due to the potential manipulation of SDK code by fraudsters (*i.e.*, Line 4 in Fig. 3(b)). Static analysis tools struggle to establish a robust pattern for locating fraudster-specified AppIDs inside the SDK

```
1 private static String zzaf(Context arg2) {
2   v0 =  Wrappers.packageManager(arg2).getApplicationInfo(arg2.getPackageName(),
128).metaData;
3   return v0.getString("com.google.android.gms.ads.APPLICATION_ID");
4 }
```

(a) In the normal Google Admob SDK, it obtains the AppID predefined by the developer.

```
1 private static String zzaep(Context arg2) {
2     v0 = Wrappers.packageManager(arg2).getApplicationInfo(arg2.getPackageName(),
128).metaData;
3     v1 = v0.getString("com.google.android.gms.ads.APPLICATION_ID");
4     return "ca-app-pub-xx~xx";//Fraudsters manipulate SDK code which makes the ad SDK
developer APIs (i.e., Manifest.xml file in this case) receiving AppID ineffectively.
5 }
6 final void zzc(...) {
7   JSONObject v9_1 = new JSONObject();
8   v9_1.put("app_id", zzaep(arg2));
9 }
```

(b) Fraudsters manipulate the obfuscated Google Admob SDK's code to replace the developer-specified AppID within the SDK.

Fig. 3: The code snippet which replaces the developer-specified AppID within the SDK.

due to the randomization introduced by obfuscation techniques in the SDK.

Moreover, as shown in some mainstream ad SDKs' updating logs[26, 29, 1, 6, 22, 39, 47], developer APIs used to receive AppIDs vary across ad networks and may change with SDK version updates (*e.g.*, Mintegral, Ironsource). And with emerging ad formats, some SDKs (*e.g.*, Facebook, Huawei, AdColony, Inmobi, Applovin) might introduce more APIs for AppID loading, potentially complicating future identification and leading to false negatives.

Solution S1: Static analysis based on control-flow and data-dependency graphs. To tackle C1 and extract hard-coded AppIDs systematically, we propose a consistent method to locate where different SDKs handle AppIDs. Our key insight is that, at the end of the stack trace of the SDK developer APIs receiving AppIDs, we notice that all ad SDKs consistently require creating a key-value pair containing the AppID, which is essential for establishing the HTTP/HTTPS data transmission protocol to the ad network. This process involves system class-originating APIs (*i.e.*, *v9_1.put("app_id", zzaf(arg2))*, Line 8 in Fig. 3(b)) which remain stable even if the SDK code is obfuscated or the ad types and SDK versions are changed. We primarily focus on these system class-originating APIs with a specific control-flow and data-dependency graph. Based on this, AlfScan can handle all versions of the ad SDKs in our research, is robust and agnostic to obfuscations, and is applicable beyond the 15 SDKs analyzed in this study.

3.2.2 Deep state of the apps. We start by giving a running example in Fig. 4 to motivate and illustrate the second challenge addressed by this work. This example is simplified from a real-world app that can change its AppID in its ad traffic based on commands received from a remote server. We omitted non-essential details for clarity.

We can only determine that the app uses multiple AppIDs by running the app dynamically since this code snippet enables the app to dynamically load and switch AppIDs. To begin with, a SharedPreferences file is dynamically written after the app executes an online config agent SDK (Line 2). The app then gets a boolean configuration (Line 3), named *changeAdId*, and a remote AppID, referred to as *remoteAppid* (Line 4), from the SharedPreferences file. Subsequently,

```
1 private void changeAdId(){
2     configFile = OnlineConfigAgent.getSharedpreferences("online_config", 0);
3     boolean changeAdId = configFile.getString("use_erge_stream_ad", false);
4     String remoteAppid = configFile.getString("appid","");
5     String finalAdId = changeAdId ? "c9e0f226" : remoteAppid;
6     loadBannerAd(finalAdId);
7 }
```

Fig. 4: The code snippet which dynamically loads the AppIDs.

the app determines which AppID to use: the *remoteAppid* or the local AppID *"c9e0f226"* (Line 5). The resulting AppID is then passed to the ad SDK for ad loading (Line 6). By employing this code snippet, the app developer can drive ad traffic for various AppIDs and execute *ALF* remotely.

Challenge C2: Imitating user interaction sequences to reach deep states inside targeted apps. Automatically and systemically extracting dynamically loaded AppIDs (*i.e.*, *remoteAppid*) from this app poses the challenges: existing directed fuzzing tools [59, 58, 46] overlook the statefulness of apps. They overlook the complex user interaction sequences required to trigger ads, which are the prerequisites for extracting dynamically loaded AppIDs. In this example, the ad is only triggered when the user performs a specific action sequence like (1) opening the ringtone selection list, (2) selecting a ringtone, and (3) setting it as the default ringtone. Existing tools, primarily designed for detecting malware, focus only on exploring system inputs, such as matching random numbers or timestamps, while disregarding the precise sequences of user events. As a result, in this example, we failed to trigger the ads by the existing directed fuzzing tools [46, 59, 58] in **two hours**. To address this issue, it is crucial to optimize directed fuzzing techniques not only for identifying potential system inputs leading to the targets but also for discovering the exact sequence of user events needed to reach the target.

Moreover, existing solutions struggle with determining the program path for triggering ads in complex, optimized, and obfuscated real-world apps, leading to inefficiency and crashes. In this example, certain solutions [46] require instrumenting over 42,000 functions throughout the entire app, which consumes 1.4min for instrumentation and 6.2min on average to handle one user event. This causes potential crashes and significant time overhead. Conducting symbolic execution [59, 58] from the *loadBannerAd()* function can also lead to state explosions due to the lengthy call trace with intricate branch conditions and dependencies on system class-originating APIs.

Solution S2: Snapshotting-based and call-graph-based directed fuzzing. To solve C2 and automatically extract dynamically loaded AppIDs, we propose a lightweight and stateful directed fuzzing tool with minimal instrumentation. We use snapshotting and call graphs to explore the concrete sequences of user interactions for triggering the ads. In this case, we only need to instrument *changeAdId* and its parents on the call graph, reducing more than **99.4%** instrumentation. We successfully trigger the ads based on the snapshotting and call graphs in **5 minutes**. More detailed evaluations are shown in Sec. 5.2.

4 ALFSCAN

As shown in Fig. 5, AlfScan comprises three modules: the *Static Analysis Module*, *Dynamic Analysis Module*, and *Final Decision Module*. In this section, we elaborate on each module's details and present a prototype implementation of AlfScan.

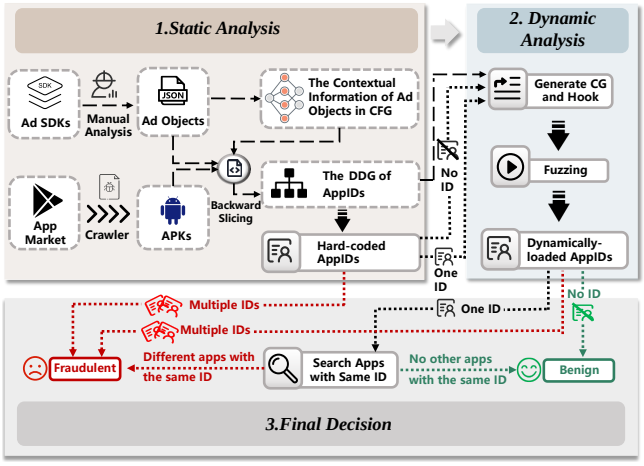

Fig. 5: The workflow of AlfScan.

4.1 Design

To detect *ALF* by revealing the app-AppID association efficiently, we adopt a strategy that accounts for the presence of both hard-coded and dynamic AppIDs within an app. Initially, we conduct static analysis to identify any instances of multiple hard-coded AppIDs. If multiple AppIDs are detected statically, the app is flagged as fraudulent. Conversely, if *no or only one* hard-coded AppID is identified statically, we proceed with the dynamic analysis to search for the dynamically loaded AppIDs. AlfScan finally detects *ALF* based on app-AppID association.

4.1.1 Static Analysis Module. The static analysis module of AlfScan established a reliable mechanism for identifying consistent patterns in AppID handling across ad SDKs and precisely pinpointing and extracting hard-coded AppID dependencies.

Extract consistent and robust pattern governing the handling of AppIDs. Our investigation has unveiled a general practice across ad SDKs, where all of them finally use the key-value pair to encapsulate the AppID. This key-value pair serves the purpose of establishing the HTTP/HTTPS protocol for the data transmission to the ad network. In the stack trace of the SDK developer API receiving AppID, the AppID is initially obtained, and finally, the key-value pair is always constructed using system class-originating APIs like *json.put()*. It is crucial to note that while SDKs' code may be obfuscated, system class-originating APIs remain stable as obfuscators typically cannot obfuscate methods outside of developers' classes [44]. As a result, our static analysis methodology primarily focuses on these consistent system class-originating APIs.

Locate ad-related system class-originating APIs precisely based on control flow graph and extract hard-coded AppID dependencies. The static analysis module then builds a control flow graph (CFG) to learn the contextual information of the system class-originating APIs above in each ad SDK in order to accurately locate them within each identified app. Afterward, AlfScan records the process within the identified app, beginning with the assignment of AppIDs, monitoring their transmission to the SDK, and concluding with their integration into the ad request via the system class-originating APIs above, all achieved through the backward slicing of the parameters (*i.e.*, AppID) of the system class-originating APIs above. This data is then transformed into a data dependency graph (DDG) to trace the generation process of AppIDs and ascertain their values.

4.1.2 Dynamic Analysis Module. The dynamic analysis module of AlfScan established a mechanism, overcoming tool limitations and minimizing the impact on app execution, to extract dynamically loaded AppIDs.

Probe user interaction sequence and extract dynamically loaded AppIDs with minimum instrumentation. If the static analysis module *does not* detect any hard-coded AppID, or if *only one* hard-coded AppID is identified, AlfScan utilizes the dynamic analysis module to search for dynamically loaded AppIDs.

This module utilizes a call graph (CG) to investigate the specific sequences of user interactions required to trigger ads. Hitting functions closer to the ads loading function on the CG increases the likelihood of triggering the ads, as these functions often serve as preconditions for ad display. Then, the dynamic analysis module hooks relevant methods for receiving and transferring AppIDs based on the DDG generated by the static analysis module. Subsequently, it snapshots the state when the fuzzing hangs at a dead end to restart exploration on a state with a high likelihood of leading to ads in the future without any heavy instrumentation. This can minimize the impact on the app's execution and overcome the limitations of existing directed fuzzing tools that rely on heavy deep program state exploration.

4.1.3 Final Decision Module. **Distinguish benign and fraudulent apps based on AppID usage.** The final decision module of AlfScan detects fraudulent apps by identifying whether the usage of AppIDs in our dataset violates the one-to-one association between apps and AppIDs.

If an app's AppID is undetectable by both the static and dynamic modules, it indicates dead ad calling codes, resulting in a benign classification. Multiple AppIDs in one app flag an app as fraudulent since an AppID should serve as a unique identifier to pinpoint the specific app where an advertisement is being displayed [24], while a single AppID in one app prompts AlfScan to further check for shared AppIDs among other apps in the dataset. If shared, all apps are deemed fraudulent; otherwise, they are classified as benign.

4.2 Implementation

4.2.1 Static Analysis Module. **Start with *json.put()* and *map.put()* system class-originating APIs.** Before conducting a large-scale detection, we conduct a manual analysis of ad SDKs from 15 different ad networks, including Google, Adcolony, Vungle, Ironsource, Unity, Facebook, Inmobi, Applovin, Amazon, Mintegral, Tencent, Bytedance, Baidu, Huawei, and Oppo. This comprehensive analysis includes both top-rated ad networks from AppBrain [4] and popular ad SDKs in the reports from two leading Chinese app stores and app development platforms [5, 11].

Our observation reveals a general practice among these ad SDKs' developer APIs used to handle AppIDs. At the end of the stack trace of these APIs, they consistently transfer AppIDs to a *JSON* or *Map* object and encapsulate them as key-value pairs (e.g., <

"app_id", *AppID_value* >). This encapsulation process is facilitated by system class-originating APIs (*i.e.*, *json.put()* and *map.put()*). Notably, methods originating from system classes won't be changed with the change or update of the ad SDKs and won't be obfuscated. Consequently, we designate *json.put()* and *map.put()* system class-originating APIs as the starting points for static analysis, ensuring AlfScan's robustness.

We then compile a list of keys (*e.g.*, *"app_id"*, *"mbridge_appkey"*, etc.) used by the system class-originating APIs in the 15 SDKs mentioned above to insert AppIDs into *JSON* or *Map* objects, enabling AlfScan to effectively pinpoint relevant code responsible for AppID handling by identifying these system class-originating APIs along with their associated keys. This manual analysis is per ad SDK, but AlfScan is designed for adaptability, capable of accommodating new SDKs through the specifications of system class-originating APIs and the key of key-value pairs.

Build CFG to learn the contextual information of the system class-originating APIs. Only using the keys to determine which *json.put()* or *map.put()* API handles the AppID leads to false positives because other SDKs or developers may use the same keys with the same system class-originating APIs but for different purposes. For instance, analytics SDKs may encapsulate the app's name with a key like "app_id" into a *JSON* or *Map* object to gather information about app usage, which is unrelated to advertising.

To address this issue, AlfScan takes a more robust approach. It considers not only the keys but also the contextual information of the system class-originating APIs above in each ad SDK, reducing the risk of misidentifying objects that are not relevant to advertising. By constructing a control flow graph (CFG) of the whole class which contains the system class-originating APIs above in each ad SDK, AlfScan learns the distinct contextual information associated with the APIs. The insight is that the contextual information of the system class-originating APIs with the same key but different purposes in different SDKs or classes differs. By comparing the contextual information extracted from the CFGs in the detected apps with the known contextual information of the 15 ad SDKs, AlfScan can identify whether an API serves an advertising-related purpose or another purpose.

Build DDG of the system class-originating APIs' parameters to get the AppID dependencies. The static analysis module utilizes CFGs to locate the system class-originating APIs responsible for encapsulating AppIDs into *JSON* or *Map* objects. These APIs' parameters (*i.e.*, AppIDs) are then treated as the target for backward slicing to get the value of AppIDs. To achieve this, we customize a backward slicing tool based on the work of Zhu et al. [68]. By applying this tool, we generate a data dependency graph (DDG). It includes all the data that make up the AppIDs, where each node represents the assignment statement for AppIDs, and each edge represents the dependency relation between the two statements. We can find out what data have been used to form the AppIDs and where the AppIDs are assigned in DDG.

Additionally, we have optimized the logic of the tool proposed in [68] to improve runtime efficiency. Irrelevant functions, such as the feature extraction module in [68], have been removed, focusing solely on detecting *ALF*. Moreover, we have restructured the enumeration loop operation using a map, resulting in a reduced time complexity of O(N).

Algorithm 1 Directed Fuzzing Algorithm

Input: $App, Target_{DDG}$
Output: Set_{AppID}
1: $CG \leftarrow GenerateCallGraph(App)$
2: $Map_{Target_{DDG},MinDistance} \leftarrow \infty$ for all $Target_{DDG}$
3: $Q_{Snapshot} \leftarrow \emptyset$ ▷ Priority queue
4: **while** $\neg(Timeout||SaturatedCoverage)$ **do**
5: **if** $Halted$ **then**
6: $s \leftarrow$ Sample($Q_{Snapshot}$)
7: $Recover(s)$
8: **end if**
9: $Hits, AppID \leftarrow RandomActions(App)$
10: $Set_{AppID} \leftarrow Set_{AppID} \cup AppID$
11: **for** $t \in Target_{DDG}$ **do**
12: $D \leftarrow \min(Dist(CG, Hits, t))$
13: **if** $Map_{Target,MinDistance}(t) > D$ **then**
14: $Q_{Snapshot} \leftarrow$ (Snapshot(), $1/D$)
15: $Map_{Target,MinDistance}(t) \leftarrow D$
16: **end if**
17: **end for**
18: **end while**
19: **return** Set_{AppID}

4.2.2 Dynamic Analysis Module. If the static analysis module only identifies zero or one hard-coded AppID, the directed fuzzing algorithm (Algorithm 1) takes an application (*App*) and a set of targets ($Target_{DDG}$) gained from the DDG generated by static module, which are the methods that retrieve, set, or transfer the value of AppIDs, as input. It then outputs the value of dynamically loaded AppIDs after running the apps.

Build call graph to explore the specific user interaction sequences triggering ads. AlfScan utilizes a call graph (CG) to investigate the specific sequences of user interactions required to trigger ads and prioritize component exploration based on their proximity to targets, increasing the likelihood of hitting relevant classes. In detail, the algorithm consists of a main loop (Line 4-15) that continues until a terminating condition is met. We employed a constant timeout or coverage saturation as the terminating condition, *i.e.*, triggering all the ads loading functions on the CG or when a timeout set by the user is reached (Line 4). If the application is halted (Line 5), which might be the app no longer showing up new screens, the algorithm samples a snapshot from a priority queue (Line 6), which is likely to be in a state closer to the target. The algorithm then performs random actions on the application, resulting in hit classes (*Hits*) and a dynamically loaded AppID (*AppID*). For each target (Line 11), the algorithm calculates the shortest distance (D) between the hit classes and the target using the call graph (Line 12). This distance serves as a heuristic to guide the exploration process toward the target, which are methods that retrieve, set, or transfer the value of AppIDs.

Use VM-level snapshotting to minimize the impact on app execution. Limited code instrumentation and VM-level snapshotting minimize the impact on app execution and enable the seamless

continuation of exploration towards the targets without additional app-level instrumentation or symbolic evaluations. In detail, we maintain a priority queue of snapshots instead of heavy methods like symbolic execution or complete coverage instrumentation to explore the paths to hit relevant targets. If the minimum distance for a target is greater than the calculated distance (Line 13), the priority queue and the minimum distance mapping are updated with a new snapshot and its priority (Line 14-15). This prioritization strategy ensures that the algorithm favors exploring paths closer to the target via snapshotting, minimizing instrumentation when exploring the paths to hit relevant targets.

4.2.3 Final Decision Module. If the static module fails to find more than one app's AppIDs, the dynamic module is employed for further analysis on whether the app has dynamically loaded AppIDs. If no AppID is found, it indicates that the app's ad loading code may be inactive, resulting in the app being categorized as benign. When multiple AppIDs are detected in an app, AlfScan regards it as fraudulent. For apps with a single AppID, AlfScan examines whether other apps in the dataset share the same AppID with it. If they do, all apps are classified as fraudulent; otherwise, they are deemed benign. Note that despite our dataset including over 91,000 apps, there remains a possibility of fraudulent apps sharing the same AppID with apps not included in our dataset. This limitation is discussed in Sec. 8.1.

5 EVALUATION

In this section, we illustrate the performance of AlfScan. All experiments are performed on a Windows 10 Desktop, equipped with an Intel i9-10900 CPU and 128 GB of RAM.

5.1 Data Collection

We use three different datasets in this paper. Two datasets contain the apps downloaded from Androzoo [3], a well-known collection of Android applications widely used in the research community [35, 12, 9, 42]. One dataset contains ad bid logs generated by mobile devices during one day. We briefly introduce each dataset below. Our dataset can be found in https://github.com/Firework471/Ad-Attribution-Laundering-Apps-Dataset.

Large-scale Dataset. We download unique apps (i.e., apps with different package names) from Androzoo between January 1st, 2021 and November 30th, 2022, resulting in a total of 155,000 unique apps. Since our focus is on detecting *ALF*, we specifically target apps that utilize ad SDKs. To this end, we select 15 popular ad SDKs, ten of which are highly rated ad networks according to AppBrain [4], while the remaining SDKs are highly rated ad networks according to the reports from two leading Chinese app stores and app development platforms [5, 11]. As shown in Table 1, by filtering out apps that do not integrate any of these 15 ad SDKs, we obtain a dataset of 91,006 unique apps as our large-scale dataset for performance evaluation and large-scale analysis. We use this dataset to perform *ALF* investigation in Sec. 6 and Sec. 7.

Ground-truth Dataset. In the absence of established benchmarks, we manually label 200 apps containing 100 fraudulent examples and 100 benign examples as our ground truth dataset for accuracy tests. We start our research with 14 fraudulent apps from our industry partner (a third-party ad verification company), which are thus included in the ground-truth. The remaining samples in the ground-truth are uniformly sampled from the whole dataset. Two Ph.D. students examined and ran apps to determine if they were involved in *ALF*. Fraudulent samples are double-checked with the industry partners for special case handling, such as scenarios like varying AppIDs from different download channels or different versions of the same app. The final decisions were made by aligning with industry standards and best practices. We use this dataset to evaluate the performance of AlfScan in Sec. 5.

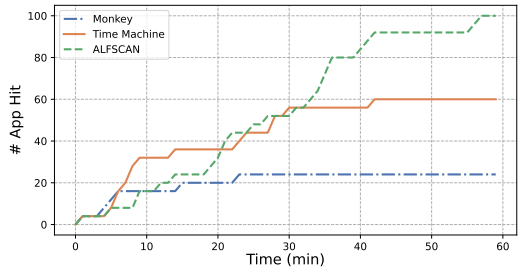

Fig. 6: The comparison of the performance of dynamic module in AlfScan with other fuzzing tools.

Ad Bidding Log Dataset. We also obtain one day's bidding logs from a **real-world scenario** of one of the aforementioned ad networks (*i.e.*, *Baidu Network*) through our collaboration with our industry partner. This access to real-world bidding logs enables us to gain valuable insights and conduct a more in-depth analysis of the fraudulent schemes employed by *ALF*. We use this dataset to estimate the loss *ALF* can cause in the real world in Sec. 6.

5.2 Performance of AlfScan

Effectiveness: We run AlfScan on our ground truth dataset. AlfScan achieves a precision of 92%, an accuracy of 94%, and a recall of 92%. The F-Score is 92%. Notably, there are eight false positives, and all of them contain multiple hard-coded AppIDs in their codes. Further investigation reveals that these developers own multiple apps and employ a shared code snippet containing all their legitimate AppIDs, but only return the corresponding AppID based on the app's package name. The static analysis techniques used in the static module struggle to handle these scenarios, mistakenly regarding the AppIDs within all conditional branches of the code snippet as the AppIDs that the app would use. Note that directly flagging apps with multiple AppIDs as potentially fraudulent in static analysis significantly reduces analysis time compared to additional dynamic analysis (15s vs. over 15mins per app). To achieve optimal accuracy, the dynamic module can be utilized to recheck apps with multiple hard-coded AppIDs detected in the static analysis module, thereby enhancing the accuracy of AlfScan. In the large-scale analysis conducted in Sec. 6, we prioritized accuracy over time efficiency and found no false positives in the randomly sampled 50 apps from those fraudulent apps detected in the large-scale analysis. Regarding the four false negatives, they are due to the limitations of the dynamic module in analyzing encrypted AppIDs or packed apps. We will discuss potential solutions for addressing false positives and false negatives in Sec. 8.1.

Table 1: Number of apps with each ad network's SDK in our large-scale dataset.

SDK Name	Google	Adcolony	Huawei	Unity	Facebook	Inmobi	Applovin	Amazon
Num of apps with this SDK	78,890	22,550	6,344	30,183	37,702	15,219	12,843	3,568
SDK Name	Ironsource	Oppo	Vungle	Tencent	Mintegral	Baidu	Bytedance	\
Num of apps with this SDK	8,548	1,332	8,043	7,078	3,033	2,124	7,359	\

Additionally, we check the status of fraudulent apps identified by AlfScan on May 25, 2023 in Sec. 6. It is observed that **79.22%** of these apps have been removed by app markets, while the remaining apps are still publicly available. We are actively engaging with relevant ad networks to facilitate responsible disclosure of our findings.

Efficiency: We conduct further evaluations to assess the efficiency of AlfScan using a randomly selected subset of 100 unique apps from our dataset. For the static module, the average examination time for each unique app is 14.76 seconds. While there is no existing tool that precisely fulfills the same purpose, we chose the state-of-the-art Android backward slicing tool ClickScanner[68], which has a similar goal to us, to compare the performance of AlfScan 's static module. The results indicate that ClickScanner takes an average of 21.74 seconds per app on the same task, making it 47% slower than our static module.

Moving on to the comparison of our dynamic module with Monkey [51] and Time Machine [19], we employ Frida instrumentation to enable Monkey and Time Machine to fuzz apps for collecting ad hit information and AppIDs. Monkey randomly simulates user interactions with the test apps, aiming to trigger ads. On the other hand, Time Machine takes random snapshots of the test app when specific screens are reached, allowing for recovery to a non-hanging state for further exploration. In contrast, our approach only takes snapshots when the current exploration of the test app aligns with the targets based on the control graph (CG).

The results, illustrated in Fig. 6, depict the number of apps in which the targets (*i.e.*, triggering ads) are successfully hit over time. Within a span of 60 minutes, Monkey and Time Machine reach the targets in 24 apps and 60 apps, whereas AlfScan successfully processes all 100 apps. Due to the requirement of a sequence of complex user interactions to trigger most targets, Monkey's performance is sub-par as it is unable to navigate to deep states. Although Time Machine manages to explore deeper states, real-world apps often possess a vast number of states, and randomly exploring each state makes it challenging to reach the target efficiently.

6 LARGE-SCALE ANALYSIS

In this section, we provide the first large-scale analysis of the *ALF* in the wild and present our findings. AlfScan successfully identifies 4,515 unique fraudulent apps out of the total 91,006 apps, accounting for 4.96% of the dataset. Two or more fraudulent apps will drive the ad traffic for the same AppID to form a cluster. AlfScan identified 1,483 fraudulent clusters. Note that we optimize the accuracy in the large-scale analysis by utilizing the dynamic module to recheck apps identified as potentially fraudulent due to multiple hard-coded AppIDs in static analysis. In the following, we present the findings of our large-scale analysis. This comprehensive list categorizes fraudulent clusters involving high-download fraudulent apps by ad SDK, AppIDs, and developers. And it has been shared with relevant ad networks for further action.

6.1 Characterizing *ALF*

Finding 1: *ALF can amplify ad traffic with a considerable factor, ranging from* 2.43 *times to* 33.33 *times.*

Our research aims to assess the extent to which the *ALF* activity can amplify ad traffic and quantify the loss of advertisers based on real-world bidding logs from the *Baidu Network* mentioned in Sec. 5. Each ad bidding log is marked with the App's name and its AppID. Since each AppID is officially assigned to one app, we define the *ad traffic amplification factor* (ATAF) given a fraudulent cluster as the ratio of the total traffic generated by the cluster to the traffic generated by the app (denoted as *legit-app*) that actually own the corresponding AppID. Due to the lack of one-to-one mappings between AppIDs and *legit-apps* (which are regarded as sensitive and private by ad networks), we make various assumptions on the mappings and estimate the traffic amplification factors accordingly. As shown in Table 2, the traffic amplification factor ranges from 2.43× (under the most conservative assumption) to 33.33× (under the most aggressive assumption).

Estimation E1: 2.43×, estimated by assuming that the *legit-app* is the one that generates the maximal traffic in the cluster. Assuming that the *legit-app*[1] is the one with maximal traffic in each fraudulent cluster would result in the most conservative estimation of the ATAF. In detail, we introduce Equation 1, $ATAF_{E1}$ specifically tailored for scenario Estimation E1. Here, n signifies the total number of apps, while T_i denotes the traffic generated by the i-th app ($i = 1 \dots n$); m represents the total number of clusters, with $C_j \subset \{1, \dots, n\}$ representing the set of indices of apps within the j-th cluster ($j = 1 \dots m$). The numerator represents the total traffic generated by all apps, while the denominator accounts for the sum of the highest traffic volume within each cluster, emphasizing the average amplification factor in all fraudulent clusters.

$$\mathrm{ATAF}_{E1} = \frac{\sum_{i=1}^{n} T_i}{\sum_{j=1}^{m} \max_{i \in C_j} T_i}. \tag{1}$$

We start the estimation with a concrete case shown in Fig. 7. We discovered 6 overlapped fraudulent clusters with 17 apps with a total of over 77 million downloads searched on [45]. These apps share 6 AppIDs, with the width of each stream indicating the number of real-world bidding logs generated by these apps on *Baidu Ad Network* with the corresponding AppID within one day. This fraudulent activity aims to generate excessive ad traffic for specific AppIDs, increasing advertising revenue beyond what these AppIDs

[1]If an app has the highest ad traffic in multiple clusters, we only consider it as the *legit-app* in the cluster where it has the largest absolute value of traffic. For other clusters, we consider the apps with the second highest traffic as the *legit-app*. This approach is applied consistently to the other assumptions.

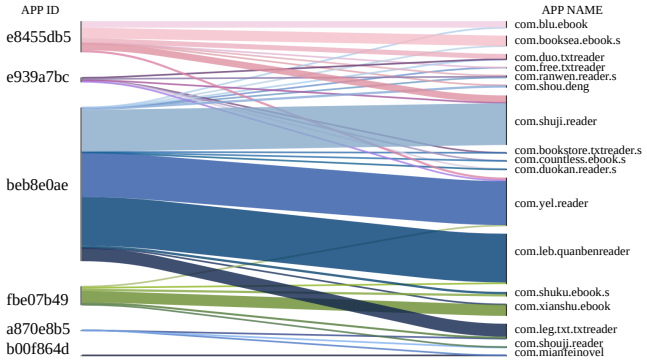

Fig. 7: 17 fraudulent apps and 6 collusion-based AppIDs with their real-world ad bidding logs. It is estimated that over 65% of the ad bidding logs are suspected to be fraudulent under the most conservative assumption.

Table 2: Estimation of ad traffic amplification factors (ATAF) by *ALF* under different assumptions.

Estimation with real-world bidding logs	
Assumption	**ATAF**
E1: the *legit-app* is the one that generates the maximal traffic	2.43×
E2: the *legit-app* is the one that generates the minimal traffic	33.33×

deserve since one AppID should only accept advertising traffic from one unique app. Our most conservative estimate suggests that over 65% of the real-world bidding logs in Fig. 7 are likely fraudulent. This implies that at least 65% of the advertising revenue invested in these apps is at risk of fraud or may not yield the expected results. These clusters potentially amply ad traffic by at least 2.86×.

Based on the bidding log data we had, expanding our analysis to all fraudulent clusters identified by AlfScan involving the *Baidu Ad Network* SDK (*i.e.*, corresponding to 148 fraudulent apps), we observe that 59% of the real-world bidding logs generated by these clusters (*i.e.*, 5.34 million logs) in one day are suspected to be fraudulent. Fraudsters can at least amply ad traffic by 2.43×. Calculated from the average bid value of $3.55 CPM[65] (USD 3.55 per 1000 impressions) in the online advertising, the potential loss attributable to these fraudulent clusters, which only involves 148 fraudulent apps, could amount to a staggering **$18,950** in one day! This highlights the ability of the *ALF* to greatly amplify fraudulent ad traffic and defraud ad revenue.

Estimation E2: 33.33×, estimated by assuming that the *legit-app* is the one that generates the minimal traffic in the cluster. The intuition for this extreme scenario is that fraudsters strategically employ disposable apps to minimize the risk of detection and shift the potential of being banned to these disposable apps, while concealing the fraudulent activities of their *legit-apps*. This tactic spreads *ALF* activities across numerous disposable apps, reducing detection risk on *legit-apps*, and consequently generating larger ad traffic on disposable apps. This deceptive strategy will be thoroughly discussed in Sec. 7 with a concrete case. Similar to the previous formula, we illustrate Equation 2 under this scenario as below:

$$\text{ATAF}_{E2} = \frac{\sum_{i=1}^{n} T_i}{\sum_{j=1}^{m} \min_{i \in C_j} T_i}. \quad (2)$$

Based on the aggressive assumption above, we uncovered a shocking 33.33× amplification in ad traffic, suggesting that up to 97% of bidding logs from these clusters could be fraudulent under this extreme assumption. This illustrates the alarming extent to which *ALF* activities can undermine the integrity of the mobile advertising ecosystem.

6.2 Patterns in Fraudulent Clusters

Finding 2: *Over 74% fraudulent clusters are composed of same-certificate apps or apps with certificate validity start date intervals of less than one month, which could enhance AlfScan's detection efficiency.*

We retrieve public certificates from fraudulent apps and get their SHA1 values to identify developers. Android apps use unique private keys, each linked to a public certificate for source verification. This correlation helps us identify apps with the same developer.

We find that on average, each AppID is shared by 4.29 apps, but is shared by only 2.56 app certificates. In detail, as shown in Fig. 8, AlfScan identified 1,483 fraudulent clusters. For 777 (52.4%) of them, all apps in each cluster use the same certificate, indicating that they are developed by the same developer/group. Moreover, there are 101 fraudulent clusters (6.8%) where at least two or more apps in each cluster use the same certificate. This suggests that fraudulent developers tend to engage in *ALF* individually or in collaboration with a limited number.

In the remaining 605 clusters (40.8%), all apps in each cluster use different certificates. Further investigation revealed that while the certificates of these apps are different, the certificate validity start date intervals (CVSDI) between the certificates within 54% of the clusters are less than one month. The certificate validity start date is the specific date and time at which a digital certificate becomes valid and can be used for encryption and authentication. CVSDI refers to the time differences between the starting dates of digital certificates. In 61% of the clusters, the CVSDI between the certificates is less than three months, and in 84% of the clusters, it is less than one year. This suggests that fraudsters may intentionally create or acquire new certificates and create new collaborative apps signed by the certificates above within these clusters after a certain period to conceal their deceptive activities. Alternatively, they might spend a period of time searching for collaborators to carry out the *ALF*.

While collusion among fraudulent apps in *ALF* makes finding fraudulent apps' accomplices difficult and complicates detection, fraudulent apps' certificates offer valuable insights to enhance AlfScan's detection efficiency. We can use the certificates' information as a heuristic to scan apps with the same certificate or shorter CVSDI first, in the hope of discovering collaborative apps sooner than a random scan order. We conducted experiments to validate this approach: Experiment EXP_{random} scanned all apps randomly, while Experiment $EXP_{\text{heuristic}}$ scanned apps sorted by certificate information. After scanning 25% (50%, 75%) of the apps, EXP_{random} and $EXP_{\text{heuristic}}$ reported 947 vs 1784 (2771 vs 3492, 3431 vs 4241)

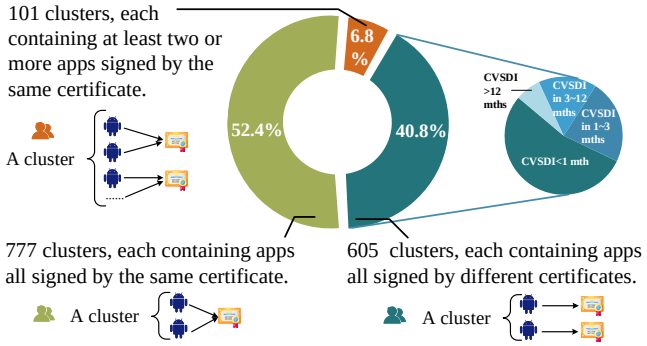

Fig. 8: Certificate overlap and certificate validity start date intervals (CVSDI) among apps in clusters indicate that 52.4% of fraudulent clusters are composed of apps from the same source, while in the remaining clusters, the majority of apps have CVSDI within 1 month.

fraudulent apps, respectively. The sooner we can identify the fraudulent apps, the smaller the financial impact on the advertisers.

6.3 Reliability Issues of Third-Party App Development Frameworks

Finding 3: 293 *fraudulent apps (corresponding to 6.5% of all fraudulent apps) share only four AppIDs, which are generated by four third-party frameworks.*

We discover another type of the AppIDs which are default AppIDs generated by third-party frameworks. These frameworks aid developers in app development. They insert default AppIDs during the initial app creation process if developers try to integrate the ad in their apps, potentially leading to fraudulent clusters. AlfScan identifies 293 unique apps (6.5% of all fraudulent apps) developed by different developers that share 4 default AppIDs generated by 4 third-party frameworks, resulting in 4 clusters. We conduct further analysis on these default AppIDs to gain deeper insights into the implications of them in *ALF*.

We begin with apps created by Kodular [32], a free platform that transforms developers' ideas into Android apps. Upon investigation, AlfScan detects an AppID (*i.e.*, 3140736) provided by the *Unity Ad Network*, which is shared by 141 apps. Analyzing the decompiled codes of these apps shown in Fig. 9(a), we discover that all of them were generated by Kodular. If developers fail to specify their own Unity AppID at the beginning, Kodular inserts a default Unity ad AppID (*i.e.*, 3140736), assigned with the variable named "UNITY_ADS_GAME_ID", into the generated apps for ad loading. In cases where developers neglect to replace these default AppIDs before releasing the apps, their apps are unknowingly utilized to drive ad traffic for these default AppIDs. Although we cannot definitively determine if apps containing such AppIDs are intentionally or unintentionally fraudulent, payments made to accounts associated with these AppIDs do undermine the interests of advertisers, who could consider such behaviors fraudulent. *ALF* through third-party frameworks can lead to substantial financial losses for advertisers, as third-party app development platforms typically involve a vast number of apps.

```
public class KodularAdsUtil {
1. public static final String UNITY_ADS_GAME_ID = "3140736";
......
}
```

(a) A potentially fraudulent third-party platform *"Kodular"* inserts a default Unity ad AppID *"3140736"* in the apps generated by it if the developers didn't specify their own Unity AppID at the beginning.

```
public class UnityAdsProperties  {
1. public static String UNITY_ADS_GAME_ID = null;
......
}
```

(b) The code snippet provided by Unity which helps developers develop apps predefines the Unity ad AppID as *"null"* at the beginning.

Fig. 9: The code snippets in Finding 3.

Subsequently, we contact the Unity Ad Support team and report our findings. They confirm that AppID 3140736 have been involved in fraudulent activities. The total download count of the detected apps involved in the fraudulent cluster formed by Kodular surpasses 8, 000, 000, resulting in significant losses for advertisers.

Typical third-party development frameworks usually provide a placeholder like "YOUR_OWN_APPID" or a null string, allowing developers to replace it with their own AppID. As depicted in Fig. 9(b), this practice is from Unity's third-party auxiliary development code aimed at facilitating the integration of Unity ads. It serves as a good example for third-party app development platforms to consider when regarding the use of default AppIDs.

7 CASE STUDIES

In this section, we investigate more sophisticated *ALF* schemes. We dive into three representative cases and share the insights.

Case 1: Remote configurations facilitate changes of dynamically loaded AppID in *ALF*. Some fraudulent apps predefine a local AppID in their code. They dynamically retrieve the configuration of another AppID from a remote server and dynamically change the AppID according to the configuration to conduct the *ALF*.

In our investigation, as shown in Fig. 10, AlfScan identified an app named *com.koodroid.chicken* with over 40 million downloads that is conducting the *ALF*. The app initially sets a predefined AppID in an *AdInstanceID* object. It first loads a configuration from a specific URL[15] and gets the configuration file at Line 1. Then it retrieves the dynamically loaded AppID from the file at Line 2. If the dynamically loaded AppID is null, the app resorts to displaying the ad using the local AppID (Line 4 and 6). Otherwise, the app displays the ad using the dynamically loaded AppID (Line 2 and 6). This approach allows fraudsters to dynamically change the AppID according to different situations, evading detection and further exploiting the advertising ecosystem.

Case 2: Disposable apps shield high-profile apps from market bans. In order to protect their high-profile apps from being banned by app markets, fraudsters employ a tactic where they create disposable apps to covertly drive ad traffic for their high-profile AppIDs. They buy or create less popular apps at a low cost to carry out large-scale *ALF* activities. By doing so, they shift the risk of being banned onto these disposable apps. The purpose of this strategy is to minimize the *ALF* activities on their high-profile apps and avoid

```
public static boolean AddQQSplashAd(Activity arg12) {
1. SharedPreferences file = arg12.getSharedPreferences(arg12.getPackageName(),0);
2. String appid  = file.getString("qq_app2_key", "");
3. if(appid.isEmpty()) {
     //The default AppId is stored locally in the AdInstanceID.
4.   appid = AdInstanceID.getQQKey();
5. }
6. new SplashAD(appid); //Load ads using final AppId.
}
```

Fig. 10: The code snippet from case 1. The app receives the AppID sent from the remote server.

```
public WebNewsFrg() {
1. String appid;
2. if(adCondition()){
3.   appid = "d7d3402a";
4.   String adUrl = "http://cpu.baidu.com/1021/"+appid+"a?chk=1";
5.   WebNewsFrg adView = new com.duoduo.oldboy.ui.view.frg.WebNewsFrg(adurl);
6. }else{
7.   appid = "c9e0f226";
8.   BaiduNative.setAppSid(appid);
9. }
}
```

Fig. 11: The simplified code snippet from the case 3. Fraudulent apps load the ad with the same AppID by abusing the interface of the ad network to avoid SDK's verification.

detection or banning by app markets. These less popular apps serve as "bagmen" for the ad traffic for specific AppIDs.

For instance, AlfScan has identified two fraudulent apps, namely *com.chillyroom.happy* and *com.hj.tlhg*. Upon further analysis, we discover that the high-profile app *com.chillyroom.happy* is a game application available on Google Play with over 1 million downloads. On the other hand, the disposable app *com.hj.tlhg* is a game application that is not commonly found on mainstream app markets. Both apps generate ad traffic using the same AppID distributed by Google Admob. The Google Admob stated that "an app ID is a unique ID number assigned to apps"[24]. However, these two apps break this rule. Furthermore, upon uploading them to VirusTotal[54], the high-profile app passes all engines' detection, while the low-quality app is flagged as a Trojan. This implies that the advertising traffic generated by the low-quality app, which could be originally barred from passing app market and advertising network reviews, and unable to obtain a legal AppID for advertising activities, is now erroneously attributed to the high-profile app. Consequently, the low-quality ad traffic generated by the disposable app can be monetized through the high-profile app's AppID. Fraudsters typically distribute these disposable apps through unofficial channels, such as forum posts with download links or non-mainstream app markets.

Moreover, we observe that both of the apps above share the same developer certificate, indicating that they are developed by the same individual or group. However, while we find information about the high-profile app *com.chillyroom.happy* on the developer's website and in their app-ads.txt file, there is no mention of the disposable app *com.hj.tlhg* anywhere. This suggests that fraudsters intentionally conceal the association with disposable apps and avoid leaving traces of them to make detection difficult.

The damage caused by this strategy is substantial due to the low cost associated with creating disposable apps. Fraudsters have no intention of updating these disposable apps in the future and are not concerned about their banishment by app markets.

Case 3: Fraudsters abuse the interface of the ad networks to evade SDKs' verification and conduct *ALF*. Fraudsters employ various tactics to evade data collection and verification by ad networks, making it easier to conduct *ALF*. Ad networks typically provide APIs through their official ad SDKs for developers to load ads. These SDKs often collect device information for verification purposes. However, AlfScan discovers that some fraudulent apps load ads with the AppID assigned by the *Baidu Ad Network* without utilizing its official SDK, resulting in the poor verification of apps.

Upon further analysis in Fig. 11, we find that these apps construct a URL containing the fraudulent AppID *"d7d3402a"* (Line 4) under certain conditions (Line 2) and load the ad using a WebView (Line 5). The variable *appid* (Line 1) used to build the URL is also employed by the SDK API (Line 7 and 8) to load ads with another legitimate AppID *"c9e0f226"* under specific conditions. Consequently, the DDG generated by AlfScan contains both values of the variable. We inspect the URL and speculate that it corresponds to the URL of the *Baidu Ad Network* advertising content pool with the ad distribution service. Moreover, we find real-world bidding requests of the fraudulent AppID from the aforementioned apps in our bidding log dataset, indicating that exploiting the vulnerability in the permission management of the ad network's interface and utilizing this URL can drive traffic to the fraudulent AppIDs effectively.

By abusing the interface of the ad network, fraudsters can conveniently circumvent the verification and information collection performed by SDKs. This scheme simplifies the execution of *ALF*. We recommend that ad networks implement stricter permission management for similar interfaces to prevent abuse by fraudsters.

8 DISCUSSION

8.1 Limitations of AlfScan

Multiple AppIDs in dead codes. To enhance runtime efficiency, AlfScan directly regards apps with multiple AppIDs, flagged by the static module, as potentially fraud. However, this approach may yield false positives. For example, some developers own multiple legitimate apps, each app with a different AppID. For convenience, developers may employ a shared code snippet that encompasses all their legitimate AppIDs for advertisements across all their apps. The static module may erroneously consider all AppIDs in the code snippet as active, although this code can only return one AppID associated with a specific app based on different app names. Other AppIDs in the snippet are relegated to dead code conditions and will never be active. To mitigate this, the dynamic module can double-check apps with multiple hard-coded AppIDs, though it is slower. Fortunately, this issue is rare, affecting only 8 benign apps in our ground-truth dataset.

Encrypted or packed AppIDs. Sly fraudsters may use encrypted AppIDs or packed apps to evade AlfScan detection, leading to false negatives. Encrypted AppIDs can't be identified without decryption, and packed apps can hide DEX files from analysis. To address these limitations, future improvements can include unpacking tools like Happer [61] for packed apps and analyzing ad traffic between fraudulent apps and ad networks to reveal the true AppIDs.

Incomplete fraudulent clusters. Despite our dataset containing over 91,000 apps, fraudulent apps may still share the same AppID with uninvolved apps outside our dataset. AlfScan's performance can be enhanced by expanding the dataset with additional sources.

8.2 Responsible Disclosure

In March 2023, we reported our findings to 15 ad network companies. Unity ad team replied within one day and confirmed that the flagged AppIDs reported by us, involving 344 apps embedding their ad SDK, were flagged as fraudulent by them, validating our detection results. Three mainstream ad companies (*i.e.*, Vungle, Baidu, Tencent) replied to us within one week, expressing a strong interest in our detection results and the tool's details. We will continue to follow up with other ad network companies for feedback. Additionally, we have also reported our results to the IAB Tech Lab [27], a global nonprofit organization that develops technical standards and solutions to enhance security, interoperability, and innovation in the digital advertising industry. We are working together with China Advertising Association to promote the establishment of new ad fraud detection standards to defeat *ALF* in ad networks.

9 RELATED WORK

Previous studies primarily focus on addressing mobile ad fraud within individual apps, overlooking collaborative ad fraud across multiple apps. Other studies focusing on app collusion attacks often assume the involvement of at least one entity experiencing malicious events or there were communication issues between them. The legitimacy of ad events and attribution procedures on individual apps in our newly identified *ALF* presents challenges for existing methods, highlighting the need for detection across multiple apps to identify shared AppIDs.

Detection of fake ad events. Existing works, designed to identify deceptive or counterfeit ad events within a single entity, fall short in effectively addressing *ALF*. Crussell et al. [17], Cao et al. [13], Christin et al. [14] and Zhu et al. [68] dissect the click fraud. Kim et al. [30] deploy FraudDetective to identify fraudulent clicks via reasoning based on observed suspicious behaviors without user interactions. Lee et al. [34] propose AdCube and identify four new impression frauds and click frauds on VR. Dong et al. [18] and Crussell et al. [17] analyze impression fraud. Liu et al. [37] design DECAF to detect placement fraud. Sun et al. [49] and Xu et al. [60] propose EvilHunter and FeatNet to detect device farms. Zeng et al. [66], Zarras et al. [63], Pochat et al. [33], Zeng et al. [64], Walls et al [55] and Akgul et al. [2] do great work on the potentially problematic content of the ads. Pearce et al. [41] have undertaken a detailed examination of the activity of one of the largest click fraud botnets in operation. Stone-Gross et al. [48] present a detailed view of how one of the largest ad exchanges operates and the associated security issues. However, *ALF* entails genuine ad events and user interactions, involving collusion in ad attribution across multiple entities and adhering to standard mobile ad display processes which makes it difficult to detect.

Detection of false attribution. Current false ad attribution detections typically focus on website-based advertising or single-entity attack scenarios. Kline et al [31] detect ad misrepresentation fraud on the websites. Vekaria et al. [53] find that dark pooling allows misinformation sites to deceptively sell their ad inventory to reputable brands. However, both of them are confined to website-based advertising and fail to tackle the distinctive challenges specific to the mobile advertising ecosystem. This includes the critical task addressed in this paper—extracting the identities of the apps and cross-checking them with the AppIDs. In the industry, the IABtechlab [27] has prioritized addressing app misrepresentation fraud [25] by a whitelist of the authorized ad inventory, while AppsFlyer [7] has concentrated on installation hijacking [8]. These efforts aim to combat the manipulation of attribution data to falsely claim credit for legitimate ad events. However, their scope is limited to scenarios only involving a single entity launching the attack.

Detection of app collusion attack. Existing app collusion attacks typically involve at least one entity with malicious events or issues with communication between them. Traditional methods detect collusive behavior in apps by identifying suspicious activities or communication within individual apps or between apps. Li et al. [36] find cross-app WebView infection across different apps. Zhang et al. [67] detect a runtime information-gathering collusion attack through side-channel information, such as thread names. Bosu et al. [10] identify a collusive data leak attack that mostly uses implicit intents. Elish et al. [21] analyze the inter-app communication flow to identify various collusion attacks. Wang et al. [56] identify collusion attacks that can compromise user privacy by examining inter-app communication channels. Liu et al. [38] construct a large-scale inter-component communication (ICC) graph to assess ICC vulnerabilities. However, all ad events and the communication between apps and networks within *ALF* appear normal which makes these methods ineffective for *ALF* detection.

Besides, we have not found any app market that claims to ensure the AppID-to-app relationships, though AppID-to-app one-to-one mapping has been officially defined by IAB[28], MRC[40], and industry players such as Google[24, 23]. Given the unique challenges of detecting *ALF*, particularly in identifying and extracting mobile application identities (i.e., AppID), we are not aware of any existing tool available for app markets to detect *ALF*.

10 CONCLUSION

In this paper, we identified a new form of ad fraud, named *ALF*. It is based on collusion among multiple seemingly legitimate apps, which challenges existing detection mechanisms designed for a single fraudulent entity. We introduced AlfScan, the first automatic tool to detect *ALF*. AlfScan analyzes the apps using both static and dynamic techniques with high accuracy and efficiency. We conducted a large-scale analysis on 91,006 real-world apps, AlfScan flags 4,515 as fraudulent and identified 1,483 fraudulent clusters. After reporting our results to 15 ad networks, we received positive feedback from 4 teams. In particular, our submitted flagged AppIDs with their associated 344 fraudulent apps have been confirmed by the Unity ad team, which highlights AlfScan's effectiveness.

ACKNOWLEDGEMENTS

We thank the anonymous shepherd and reviewers for their insightful comments. The authors from Shanghai Jiao Tong University were partially supported by the National Natural Science Foundation of China (No. 62325207, 62332013, 62302298). We would like to express our gratitude to the ad verification company RTBAsia (contact: Qiuhua Fan) and the China Advertising Association for their invaluable assistance in analyzing the fraudulent examples.

REFERENCES

[1] Adcolony. 2023. Adcolony ad setup. https://support.adcolony.com/helpdesk/adcolony-android-sdk/. (2023).

[2] Omer Akgul, Richard Roberts, Moses Namara, Dave Levin, and Michelle L. Mazurek. 2022. Investigating influencer VPN ads on youtube. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 876–892. DOI: 10.1109/SP46214.2022.9833633.

[3] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (MSR '16). ACM, Austin, Texas, 468–471. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2903508.

[4] AppBrain. 2023. Android ad network statistics and market share. https://www.appbrain.com/stats/libraries/ad-networks. (2023).

[5] AppInChina. 2023. How can you monetize your app or game in china with ads? https://www.appinchina.co/services/monetization/ad-monetization/. (2023).

[6] Applovin. 2023. Applovin ad setup. https://dash.applovin.com/documentation/mediation/android/getting-started/integration. (2023).

[7] AppsFlyer. 2023. Appsflyer - appsflyer ad analytics. https://www.appsflyer.com. (2023).

[8] AppsFlyer. 2023. Attribution fraud. https://www.appsflyer.com/glossary/attribution-fraud/. (2023).

[9] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, (Aug. 2022), 3971–3988. ISBN: 978-1-939133-31-1. https://www.usenix.org/conference/usenixsecurity22/presentation/arp.

[10] Amiangshu Bosu, Fang Liu, Danfeng Yao, and Gang Wang. 2017. Collusive data leak and more: large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 71–85.

[11] Buildfire. 2023. Top mobile ad networks (2023). https://buildfire.com/mobile-ad-networks/. (2023).

[12] Haipeng Cai, Na Meng, Barbara Ryder, and Daphne Yao. 2019. Droidcat: effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security*, 14, 6, 1455–1470. DOI: 10.1109/TIFS.2018.2879302.

[13] Chenhong Cao, Yi Gao, Yang Luo, Mingyuan Xia, Wei Dong, Chun Chen, and Xue Liu. 2021. Adsherlock: efficient and deployable click fraud detection for mobile applications. *IEEE Trans. Mob. Comput.*, 20, 4, 1285–1297. DOI: 10.1109/TMC.2020.2966991.

[14] Nicolas Christin, Sally S. Yanagihara, and Keisuke Kamataki. 2010. Dissecting one click frauds. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (CCS '10). Association for Computing Machinery, Chicago, Illinois, USA, 15–26. ISBN: 9781450302456. DOI: 10.1145/1866307.1866310.

[15] Fraudulent App com.koodroid.chicken. 2022. The configurations downloaded from the remote server. https://www.product.koodroid.com/download/check.php?pr=chicken&mt=5&new=true. (2022).

[16] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the clones: detecting cloned applications on android markets. In *Computer Security – ESORICS 2012*. Sara Foresti, Moti Yung, and Fabio Martinelli, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 37–54. ISBN: 978-3-642-33167-1.

[17] 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* (MobiSys '14). Association for Computing Machinery, Bretton Woods, New Hampshire, USA, 123–134. ISBN: 9781450327930. DOI: 10.1145/2594368.2594391.

[18] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F. Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. 2018. Frauddroid: automated ad fraud detection for android apps. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE 2018). Association for Computing Machinery, Lake Buena Vista, FL, USA, 257–268. ISBN: 9781450355735. DOI: 10.1145/3236024.3236045.

[19] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of android apps. In *Proceedings of the 42nd International Conference on Software Engineering* (ICSE '20), 1–12.

[20] Yingtong Dou, Weijian Li, Zhirong Liu, Zhenhua Dong, Jiebo Luo, and Philip S. Yu. 2020. Uncovering download fraud activities in mobile app markets. In *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining* (ASONAM '19). Association for Computing Machinery, Vancouver, British Columbia, Canada, 671–678. ISBN: 9781450368681. DOI: 10.1145/3341161.3345306.

[21] Karim O Elish, Haipeng Cai, Daniel Barton, Danfeng Yao, and Barbara G Ryder. 2018. Identifying mobile inter-app communication risks. *IEEE Transactions on Mobile Computing*, 19, 1, 90–102.

[22] Facebook. 2023. Facebook ad setup. https://developers.facebook.com/docs/audience-network/setting-up/ad-setup. (2023).

[23] GOOGLE. 2023. Ad fraud. https://support.google.com/googleplay/android-developer/answer/9969955?hl=en&ref_topic=9969691&sjid=10118054247325294537-AP#zippy=%2Cexamples-of-common-violations. (2023).

[24] GOOGLE. 2023. Find your app ids & ad unit ids. https://support.google.com/admob/answer/7356431?hl=en. (2023).

[25] OpenRTB Working Group. 2019. Iab tech lab authorized sellers for apps (app-ads.txt) version 1.0. iab tech lab. https://iabtechlab.com/wp-content/uploads/2019/03/appads.txt-v1.0-final-.pdf. (2019).

[26] Huawei. [n. d.] Huawei ad setup. https://developer.huawei.com/consumer/en/doc/development/HMSCore-Guides/publisher-service-banner-0000001050066915. ().

[27] IAB. 2023. IAB Tech Lab. https://iabtechlab.com. (2023).

[28] IAB. 2020. The iab europe guide to ad fraud. https://iabeurope.eu/wp-content/uploads/2020/12/IAB-Europe-Guide-to-Ad-Fraud-1.pdf. (2020).

[29] Inmobi. 2023. Inmobi ad setup. https://www.inmobi.com/sdk. (2023).

[30] Joongyum Kim, Junghwan Park, and Sooel Son. 2021. The abuser inside apps: finding the culprit committing mobile ad fraud. In *NDSS*.

[31] Jeffery Kline, Aaron Cahn, and Paul Barford. 2022. Placement laundering and the complexities of attribution in online advertising. *arXiv preprint arXiv:2208.07310*.

[32] Kodular. 2023. Kodular home. https://www.kodular.io/. (2023).

[33] Victor Le Pochat, Laura Edelson, Tom Van Goethem, Wouter Joosen, Damon McCoy, and Tobias Lauinger. 2022. An audit of facebook's political ad policy enforcement. In *31st USENIX Security Symposium (USENIX Security 22)*, 607–624.

[34] Hyunjoo Lee, Jiyeon Lee, Daejun Kim, Suman Jana, Insik Shin, and Sooel Son. 2021. AdCube: WebVR ad fraud and practical confinement of Third-Party ads. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, (Aug. 2021), 2543–2560. ISBN: 978-1-939133-24-3. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-hyunjoo.

[35] Deqiang Li and Qianmu Li. 2020. Adversarial deep ensemble: evasion attacks and defenses for malware detection. *IEEE Transactions on Information Forensics and Security*, 15, 3886–3900. DOI: 10.1109/TIFS.2020.3003571.

[36] Tongxin Li, Xueqiang Wang, Mingming Zha, Kai Chen, XiaoFeng Wang, Luyi Xing, Xiaolong Bai, Nan Zhang, and Xinhui Han. 2017. Unleashing the walking dead: understanding cross-app remote infections on mobile webviews. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 829–844.

[37] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. 2014. DECAF: detecting and characterizing ad fraud in mobile apps. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, (Apr. 2014), 57–70. ISBN: 978-1-931971-09-6. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/liu%5C_bin.

[38] Fang Liu, Haipeng Cai, Gang Wang, Danfeng Yao, Karim O Elish, and Barbara G Ryder. 2017. Mr-droid: a scalable and prioritized analysis of inter-app communication risks. in 2017 ieee security and privacy workshops (spw). (2017).

[39] Mintegral. 2023. Mintegral ad setup. https://www.mintegral.com/. (2023).

[40] MRC. 2020. Invalid traffic detection and filtration standards addendum. https://mediaratingcouncil.org/sites/default/files/Standards/IVT%20Addendum%20Update%20062520.pdf. (2020).

[41] Paul Pearce, Vacha Dave, Chris Grier, Kirill Levchenko, Saikat Guha, Damon McCoy, Vern Paxson, Stefan Savage, and Geoffrey M. Voelker. 2014. Characterizing large-scale click fraud in zeroaccess. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (CCS '14). Association for Computing Machinery, Scottsdale, Arizona, USA, 141–152. ISBN: 9781450329576. DOI: 10.1145/2660267.2660369.

[42] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE Symposium on Security and Privacy (SP)*, 1332–1349. DOI: 10.1109/SP40000.2020.00073.

[43] Pixalate. 2023. Blocking using data feeds. https://www.pixalate.com/knowledgebase/high-risk-app-lists. (2023).

[44] ProGuard. 2023. How it works? https://www.guardsquare.com/manual/home. (2023).

[45] QIMAI. 2023. Qimai. https://www.qimai.cn/. (2023).

[46] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making malory behave maliciously: targeted fuzzing of android execution environments. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, (Eds.) IEEE / ACM, 300–311. DOI: 10.1109/ICSE.2017.35.

[47] ronsource. 2023. Ronsource ad setup. https://developers.is.com/ironsource-mobile/android/sdk-change-log/. (2023).

[48] Brett Stone-Gross, Ryan Stevens, Apostolis Zarras, Richard Kemmerer, Chris Kruegel, and Giovanni Vigna. 2011. Understanding fraudulent activities in online ad exchanges. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (IMC '11). Association for Computing Machinery, Berlin, Germany, 279–294. ISBN: 9781450310130. DOI: 10.1145/2068816.2068843.

[49] Suibin Sun, Le Yu, Xiaokuan Zhang, Minhui Xue, Ren Zhou, Haojin Zhu, Shuang Hao, and Xiaodong Lin. 2021. Understanding and detecting mobile ad fraud through the lens of invalid traffic. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (CCS '21). Association for Computing Machinery, Virtual Event, Republic of Korea, 287–303. ISBN: 9781450384544. DOI: 10.1145/3460120.3484547.

[50] Technavio. 2023. Advertising services market size to grow by usd 188.92 billion from 2021 to 2026, driven by the growth in in-app advertising - technavio. https://www.prnewswire.com/news-releases/advertising-services-market-size-to-grow-by-usd-188-92-billion-from-2021-to-2026–driven-by-the-growth-in-in-app-advertising—technavio-301788330.html. (2023).

[51] 2023. UI/Application Exerciser Monkey. en. https://developer.android.com/studio/test/other-testing-tools/monkey. (2023). Retrieved Apr. 8, 2023 from.

[52] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*. Stephen A. MacKay and J. Howard Johnson, (Eds.) IBM, 13. https://dl.acm.org/citation.cfm?id=782008.

[53] Yash Vekaria, Rishab Nithyanand, and Zubair Shafiq. 2022. The inventory is dark and full of misinformation: understanding the abuse of ad inventory pooling in the ad-tech supply chain. *arXiv preprint arXiv:2210.06654*.

[54] VirusTotal. 2020. Virustotal. https://www.virustotal.com/gui/. (2020).

[55] Robert J. Walls, Eric D. Kilmer, Nathaniel Lageman, and Patrick D. McDaniel. 2015. Measuring the impact and perception of acceptable advertisements. In *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*. Kenjiro Cho, Kensuke Fukuda, Vivek S. Pai, and Neil Spring, (Eds.) ACM, 107–120. DOI: 10.1145/2815675.2815703.

[56] Bin Wang, Chao Yang, and Jianfeng Ma. 2023. Iafdroid: demystifying collusion attacks in android ecosystem via precise inter-app analysis. *IEEE Transactions on Information Forensics and Security*.

[57] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2018. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Trans. Priv. Secur.*, 21, 3, Article 14, 32 pages. DOI: 10.1145/3183575.

[58] Michelle Y. Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. en. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. ISBN: 9781891562419. DOI: 10.14722/ndss.2016.23118.

[59] Wong, Michelle Y. and Lie, David. 2022. Driving execution of target paths in android applications with (a) car. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (ASIA CCS '22). Association for Computing Machinery, Nagasaki, Japan, 888–902. ISBN: 9781450391405. DOI: 10.1145/3488932.3497765.

[60] Chao Xu, Zhentan Feng, Yizheng Chen, Minghua Wang, and Tao Wei. 2018. Featnet: large-scale fraud device detection by network representation learning with rich features. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security* (AISec '18). Association for Computing Machinery, Toronto, Canada, 57–63. ISBN: 9781450360043. DOI: 10.1145/3270101.3270109.

[61] Lei Xue, Hao Zhou, Xiapu Luo, Yajin Zhou, Yang Shi, Guofei Gu, Fengwei Zhang, and Man Ho Au. 2021. Happer: unpacking android apps via a hardware-assisted approach. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1641–1658. DOI: 10.1109/SP40001.2021.00105.

[62] Tianjun Yao, Qing Li, Shangsong Liang, and Yadong Zhu. 2020. Botspot: a hybrid learning framework to uncover bot install fraud in mobile advertising. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management* (CIKM '20). Association for Computing Machinery, Virtual Event, Ireland, 2901–2908. ISBN: 9781450368599. DOI: 10.1145/3340531.3412690.

[63] Apostolis Zarras, Alexandros Kapravelos, Gianluca Stringhini, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. 2014. The dark alleys of madison avenue: understanding malicious advertisements. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (IMC '14). Association for Computing Machinery, Vancouver, BC, Canada, 373–380. ISBN: 9781450332132. DOI: 10.1145/2663716.2663719.

[64] Eric Zeng, Tadayoshi Kohno, and Franziska Roesner. 2021. What makes a "bad" ad? user perceptions of problematic online advertising. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (CHI '21) Article 361. Association for Computing Machinery, Yokohama, Japan, 24 pages. ISBN: 9781450380966. DOI: 10.1145/3411764.3445459.

[65] Eric Zeng, Rachel McAmis, Tadayoshi Kohno, and Franziska Roesner. 2022. What factors affect targeting and bids in online advertising? a field measurement study. In *Proceedings of the 22nd ACM Internet Measurement Conference* (IMC '22). Association for Computing Machinery, Nice, France, 210–229. ISBN: 9781450392594. DOI: 10.1145/3517745.3561460.

[66] Eric Zeng, Miranda Wei, Theo Gregersen, Tadayoshi Kohno, and Franziska Roesner. 2021. Polls, clickbait, and commemorative $2 bills: problematic political advertising on news and media websites around the 2020 u.s. elections. In *Proceedings of the 21st ACM Internet Measurement Conference* (IMC '21). Association for Computing Machinery, Virtual Event, 507–525. ISBN: 9781450391290. DOI: 10.1145/3487552.3487850.

[67] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiaoyong Zhou, and XiaoFeng Wang. 2015. Leave me alone: app-level protection against runtime information gathering on android. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 915–930.

[68] Tong Zhu, Yan Meng, Haotian Hu, Xiaokuan Zhang, Minhui Xue, and Haojin Zhu. 2021. Dissecting click fraud autonomy in the wild. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (CCS '21). Association for Computing Machinery, Virtual Event, Republic of Korea, 271–286. ISBN: 9781450384544. DOI: 10.1145/3460120.3484546.

[69] Yadong Zhu, Xiliang Wang, Qing Li, Tianjun Yao, and Shangsong Liang. 2021. Botspot++: a hierarchical deep ensemble model for bots install fraud detection in mobile advertising. *ACM Trans. Inf. Syst.*, 40, 3, Article 50, 28 pages. DOI: 10.1145/3476107.

A AUTHORITATIVE GUIDELINES ON *ALF*

There are some **industry authoritative guidelines** about *ALF*: (1) The Media Rating Council (MRC), which oversees audits for audience measurement services that media buyers and sellers in the advertising community rely on, explicitly lists the traffic generated by *ALF* as sophisticated invalid traffic[40]: "*... The second category, herein referred to as "Sophisticated Invalid Traffic" or SIVT, consists of more difficult to detect situations that require advanced analytics, multi-point corroboration/coordination, significant human intervention, etc., to analyze and identify. Key examples are: ... App misrepresentation: App ID spoofing ...*". (2) The Interactive Advertising Bureau (IAB), which leads the interactive advertising association and represents companies responsible for selling over 75% of online advertising in the United States, regards ad content generated by *ALF* as falsely represented content[28]: " *The aim of advertising is to deliver the right message to the right person in the right environment. Fraudsters use various techniques to compromise all of these three core values across various platforms and devices, resulting in wasted advertiser spend and damaged reputations for susceptible publishers. Below are the top 10 most common types of fraud detected. ... APP NAME SPOOFING: Similar to domain spoofing in display, apps can submit a false app identifier to the bidding platform. This interferes with detection of apps utilising background services to load ads, as well as brand safety and contextual targeting ...* ". (3) Google has also stated that *ALF* is a typical example of common violation[23]: "*Examples of common violations: ... False representations of the ad inventory by an app, for example ... an app that misrepresents the package name that is being monetized ...*".