The Abuser Inside Apps: Finding the Culprit Committing Mobile Ad Fraud

Joongyum Kim*
School of Computing, KAIST
kjkpoi@kaist.ac.kr

Jung-hwan Park*
School of Computing, KAIST
ahnmo@kaist.ac.kr

Sooel Son
School of Computing, KAIST
sl.son@kaist.ac.kr

***Abstract*—Mobile ad fraud is a significant threat that izes app publishers and their users, thereby undermini ecosystem of app markets. Prior works on detecting mol fraud have focused on constructing predefined test scenari preclude user involvement in identifying ad fraud. Howev to their dependence on contextual testing environments, works have neglected to track which app modules and user interactions are responsible for observed ad fraud.**

To address these shortcomings, this paper presents the and implementation of FraudDetective, a dynamic testing work that identifies ad fraud activities. FraudDetective focu identifying fraudulent activities that originate without an interactions. FraudDetective computes a full stack trace an observed ad fraud activity to a user event by conn fragmented multiple stack traces, thus generating the relationships between user inputs and the observed frau activity. We revised an Android Open Source Project (to emit detected ad fraud activities along with their full traces, which help pinpoint the app modules responsible f observed fraud activities. We evaluate FraudDetective on 48,172 apps from Google Play Store. FraudDetective reports that 74 apps are responsible for 34,453 ad fraud activities and find that 98.6% of the fraudulent behaviors originate from embedded third-party ad libraries. Our evaluation demonstrates that FraudDetective is capable of accurately identifying ad fraud via reasoning based on observed suspicious behaviors without user interactions. The experimental results also yield the new insight that abusive ad service providers harness their ad libraries to actively engage in committing ad fraud.

I. Introduction

The mobile ad has been a compelling motivator that drives app publishers to develop innovative apps. Nowadays, mobile ads have become pervasive. Recent research shows that the mobile ad market has expanded to reach 187 billion USD in 2020, which comprises 30.5% of the global ad market budget [49].

Whereas the mobile ad ecosystem facilitates the virtuous cycle of bringing innovation to mobile computing, ad fraud has become a major security threat. Previous studies [47], [52], [73] have demonstrated that the total losses due to ad fraud amount to approximately 9%-20% of the annual market budget for global mobile advertising. For instance, ZeroAccess operates the world's largest botnet, making $100,000 daily via ad fraud [64].

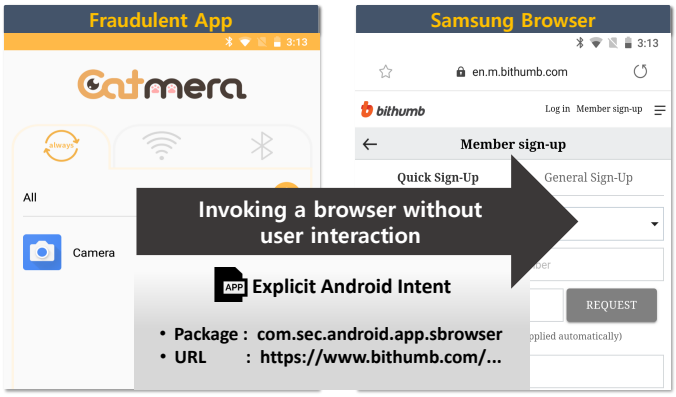

Fig. 1: Ad fraud example: The invocation of a different app via Android explicit Intent, which is triggered without user interaction.

To maintain the sanity of Google Play Store, protecting users from mobile ad fraud, Google has published a developer policy regarding mobile ads [30] that denounces abusive ad libraries for their excessive monetization. Also, Google Android security has actively identified apps with abusive behaviors that result in ad fraud [54].

Previous studies proposed novel dynamic testing frameworks, MAdFraud [25] and MAdLife [21], designed to identify ad fraud in Android apps. Whereas they demonstrate their efficacy in finding apps that commit ad fraud, these frameworks lose sight of which user interactions and app modules cause the commission of ad fraud. MAdFraud conducts dynamic testing of apps without any interaction, emulating an environment involving no user interaction. MAdLife focuses on identifying full-screen ads that pop up in the foreground immediately after an app starts. However, depending on these types of contextual testing environments inevitably limits the testing of target app functionalities.

Our contributions. We design and implement FraudDetective, a dynamic testing framework for identifying mobile ad fraud initiated via Android apps. We define an *ad fraud activity* as (1) a click URL request submission for which the targeted ad service counts user clicks or (2) an invocation of a different app

*Both authors contributed equally to the paper

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23161
www.ndss-symposium.org

in the foreground without any explicit user interaction. Figure 1 is an example of the latter type of ad fraud activity, which instantiates the Samsung Android browser with a Bithumb sign-up page without having any user input. By definition, identifying an ad fraud activity requires computing the causality between that fraudulent activity and explicit user interaction, such as a touch or a drag event. For instance, it is benign for a mobile app to invoke the Chrome browser with a promotional page when a user touches a banner ad for Nike sneakers. On the other hand, when this mobile app invokes the browser with the same page but without user touch, it becomes an ad fraud activity.

Prior approaches [21], [25] to the runtime detection of ad fraud activities have suffered from two types of problems: 1) they have neglected to model the causality between observed fraud activities and user interactions precisely, resorting to leverage restricted testing environments having no user input; and 2) their methods have been unable to track which app modules conduct an observed ad fraud activity.

To address these issues, we propose to compute a *full stack trace* to capture such causal relationships. A full stack trace is a sequence of callees that lie within the calling context from an execution entry to a statement committing an ad fraud activity. This stack trace contributes to identifying the existence of explicit user input and determines which source classes invoked the ad fraud activity, revealing the culprit module within an app committing ad fraud.

However, a stack trace for a fraud activity is often fragmented due to the usage of multiple threads, message queues connecting event generators and their handlers, and external Chromium WebView instances separated from apps. To compute the non-fragmented full stack trace for an observed fraud activity, we revise an Android Open Source Project (AOSP) that corresponds to Android Oreo [5] and leverage this AOSP to emit execution logs of ad fraud activities and their stack traces. FraudDetective harnesses this AOSP for its dynamic testing of apps, connects fragmented stack traces via collecting execution logs, and identifies ad fraud activities.

We evaluate FraudDetective on 48,172 apps that FraudDetective crawled from Google Play Store. Of 48,172 apps, FraudDetective identifies 74 apps that commit 34,453 ad fraud activities. We further analyze whether these observed ad fraud cases originate in the apps themselves or in ad libraries embedded in these apps. We find that 98.6% of the observed activities originate from ad libraries, which account for 73 of the 74 apps identified. This observation yields the new insight that abusive ad library providers play a key role in the commission of ad fraud by exploiting actual user devices. That is, an ad service provider invites various app developers to embed their ad libraries and commits ad fraud by victimizing mobile users with mobile apps from these developers. Consequently, the victims contribute to increasing ad traffic by visiting certain promotional websites, as orchestrated by these ad library providers.

In addition, we observed that one Google Play Store app has been used to forcibly invoke YouTube and Naver [42] mobile apps in the foreground and redirect their users to web pages or videos promoting specific products and services. Although YouTube and Naver do not conduct excessive promotions, ad fraud activities from this app may lead to users blaming YouTube and Naver because their apps with promotional content are brought into the foreground without explicit user interaction. Thus, users have left related negative feedback about YouTube and Naver apps at Google Play Store. However, FraudDetective finds, through the analysis of full stack traces of observed ad fraud activities, that the culprits are abusive ad libraries that have redirected users by invoking these apps via cross-app Android Intents.

By computing full stack traces of ad fraud activities, we improve the precision of ad fraud detection compared to prior work. We believe that this capability is an important requisite for dynamically identifying ad fraud behaviors, which is applicable to vetting mobile apps in many app stores, thereby protecting users from ad fraud campaigns.

II. Background

A. Mobile ad ecosystem

Serving mobile ads is a prevalent method of monetizing mobile apps. App developers integrate their mobile apps with a mobile ad library, and the embedded library conducts the rendering of ads at the screen estates of its hosting app. The usage of such mobile ad libraries is quite prevalent; approximately 56% of Android apps in Google Play Store include AdMob, an ad library managed by Google [16].

There are three key participants in the mobile advertising ecosystem: publisher, advertiser, and ad service provider. (1) A publisher is an app developer who monetizes their app by integrating an ad library managed by an ad service provider. (2) An advertiser or their agency designs an ad campaign for their target audiences and requests the launch of such ad campaigns to an ad service provider. (3) An ad service provider connects the advertisers' need for greater exposure to their ads with the publishers' offer to serve ads. This ad service provider also offers an ad library for publishers to include. When a publisher embeds this ad library, it fetches ads from the ad service provider and then renders the ads at user devices. Each rendered ad is called an ad impression, which typically refers to an image or a video rendered one time.

Ad service providers offer various ways of charging advertisers for their services. In general, there are three representative methods of billing advertisers: cost-per-mile (CPM), cost-per-click (CPC), and cost-per-install (CPI). CPM, CPC, and CPI involve charging an advertiser for rendered ad impressions, user clicks, and app installs, respectively. For instance, when an ad service provider asks $5 CPM for a given campaign, an advertiser is required to pay 5 USD when this campaign delivers 1,000 ad impressions to audiences.

B. Mobile ad fraud

Mobile ad fraud refers to an operation that generates unwanted ad traffic involving ad impressions, clicks, or conversions, thus generating fraudulent revenues. This paper focuses on two types of ad fraud: click fraud and impression fraud.

Click fraud. Click fraud refers to a fraudulent operation generating illegitimate clicks that consume the marketing budget of a victim advertiser [46], [79]. A click fraud adversary could be an abusive publisher, an ad service provider, or a competitor of a targeted advertiser. The motivation for publishers and ad

service providers to engage in click fraud is to inflate their CPC prices by promoting a fraudulently high CPC [36], [82]. An advertising competitor may recruit a botnet network to click ad impressions of a targeted advertiser, thus depleting their ad budget.

A successful click fraud campaign is highly dependent on generating click URL requests that a target ad service provider accepts and counts toward the billing of a target advertiser. A typical click URL points to an ad server and redirects users to a landing page. Consider the click URL example below:

This click URL points to the Cauly ad network and contains information that indicates an application identifier (aRU5Bq1u), an application package name (kr.kbac3k.ktv), and a publisher (466158) that initiated the request.

Therefore, there exist two ways of implementing click fraud: 1) the attacker sends a vast volume of click URL requests that a target ad service provider accepts by leveraging her own botnet networks [26], [28], [64]; 2) alternatively, the attacker deceives users into actually clicking ad impressions, thus generating admissible click URL requests from users' devices [82]. The former method requires an understanding of how the ad service provider generates an admissible click URL request. On the other hand, the latter requires no such understanding but necessitates deceiving a large number of users into clicking unwanted ad impressions [82].

Impression fraud. One key requirement of mobile ads is to render ad impressions; advertisers are often charged by the number of rendered impressions. An attacker is able either to hide ads underneath other visible elements on the screen or create invisible ads by making them small [32], [56]. In either case, no ad impressions are exposed to users, but the users' devices still send ad impression requests, which results in advertisers being charged.

III. Motivation

Preserving the sanity of app markets (e.g., Google Play Store) is an essential task for protecting the security and privacy of market users, thus establishing long-term success. Therefore, Google has been operating an Android app analysis framework, known as Bouncer, to find malware [39], apps with known security vulnerabilities [67], and ad abusing apps [54].

Considering that the number of Android apps in Google Play Store surpassed 2.8 million as of October 2019 [17], analyzing mobile apps in an automatic way is paramount to the scalable detection of malicious or abusive applications. To this end, previous research has proposed novel dynamic testing frameworks designed to identify Android apps committing ad fraud [21], [25].

MAdFraud is designed to identify fraudulent click URL requests without authentic user clicks [25]. It observes outgoing HTTP URL requests and their responses while not interacting with an app under testing, which simulates an execution environment with no user intervention. MAdLife is another dynamic analysis framework that detects abusive full-screen ad impressions rendered without any user interactions. It compares the pre-click and post-click log data and screenshots of a target app and classifies it as abusive when two data points are equivalent, denoting that the app has already rendered the full-screen ad impression event before the framework conducts an actual click [21].

We argue that these previous frameworks suffer from four limitations as follows. (1) Their approaches are unable to manifest causal relationships between user interactions and fraudulent activities, such as an automatic submission of a click URL request triggered without user touch. MAdFraud classifies all observed HTTP requests with click URLs as abusive when a target app is in the foreground or background. It creates a specific testing environment in which the target app cannot obtain legitimate user inputs. Thus, when this app requires user interaction before committing ad fraud, MAdFraud inevitably produces a false negative. MAdLife identifies an ad fraud activity when a target app shows an ad landing page in the foreground before clicking WebView ads. It only focuses on identifying click fraud that involves clicking a WebView instance and misses computing causal relationships of fraudulent activities not involving WebView.

(2) The previous strategies cannot pinpoint which app module conducts ad fraud. Because both approaches only take external behaviors into account, they lose sight of internal app logic and are thus unable to determine the culprit committing observed ad fraud. The offender might be a target app under testing or one of the embedded libraries in the hosting app. We observed that 73 apps conducted ad fraud by means of their embedded third-party libraries (§VI-B). Note that pinpointing abusive modules helps app developers patch their apps, especially when they inadvertently conduct click fraud by including fraudulent ad libraries. Auditors also benefit from being able to penalize the identified ad services in order to stop their fraud campaigns, instead of needing to track the patch of each identified app.

(3) Neither system interacts with a target app, leading them to cover only a small portion of the functionalities. By design, MAdFraud should not have user interactions during its testing. This limitation brings with it the inevitable shortcoming of limited testing coverage. When a target app requires app-specific permission consents or the touching of user controls to initiate abusive behaviors, MAdFraud will produce false negatives.

(4) Both systems leverage emulators to conduct dynamic testing. Thus, the systems may not observe ad fraud activities that only appear at real mobile devices [51], [57].

Note that the aforementioned four limitations also become technical challenges that a next-generation dynamic testing framework should address for the accurate detection of mobile ad fraud. In this paper, we define an ad fraud activity as a click URL request submission or an invocation of other apps via cross-app Intents without genuine user interaction. For the accurate detection of these ad fraud activities, we propose a dynamic ad fraud detection framework, FraudDetective.

To address the first and second technical challenges, we revised an Android operating system so that it produces a full stack trace from an execution entry to a sink method that sends a click URL request or invokes a cross-app Intent. FraudDetective is able to identify whether each full stack trace

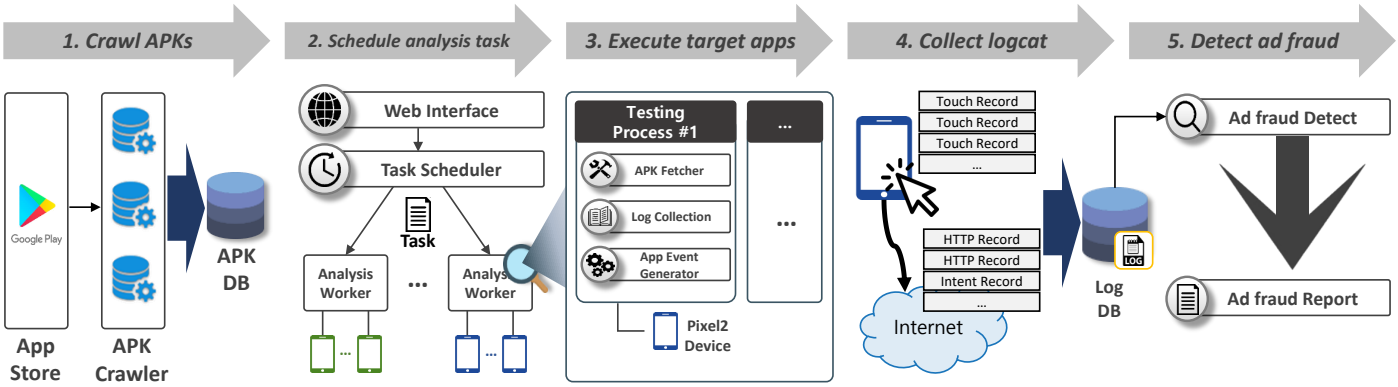

Fig. 2: FraudDetective architecture: A workflow overview of dynamic Android app testing.

originates from actual user interactions, thereby addressing the first challenge. Each stack trace consists of executed callees with their classes, denoting which class modules are responsible for implementing each callee. Note that this stack trace helps auditors pinpoint in-app modules that explicitly conduct ad fraud, thereby addressing the second challenge.

FraudDetective is capable of supporting various user interactions to increase its testing coverage. Unlike MAdFraud and MAdLife, full stack traces untangle the testing strategy with user interaction from ad fraud detection policies, thereby addressing the third challenge. In response to the final challenge, FraudDetective uses actual Android mobile devices with a modified AOSP image to minimize possible false negatives.

IV. FraudDetective Overview

FraudDetective is an automated dynamic testing framework that (1) crawls Android apps from a given app market, (2) schedules an analysis task that specifies a list of apps to vet, (3) executes apps in accordance with a specified testing strategy on real mobile devices with a revised Android Open Source Project (AOSP) framework, (4) collects tagged execution logs via ADB Logcat [8], and (5) identifies ad fraud activities from the collected logs.

Figure 2 depicts the architecture of FraudDetective and provides a workflow overview of identifying ad fraud cases. It starts by crawling Android Package Kits (APKs) from Google Play Store. We implement an APK crawler that harvests APKs and writes them into the APK storage.

FraudDetective employs the Task Scheduler module with a web administrator page. A FraudDetective user schedules an analysis task via this web interface. This analysis task specifies a list of mobile apps stored in the APK storage and a dynamic testing duration. Then, the Task Scheduler dispatches a specified task to an available analysis worker in which dynamic testing actually occurs. This producer-consumer architecture is implemented via a message queue framework, RabbitMQ, which distributes tasks among different hosts via network channels.

An analysis worker refers to a host machine connected to Android mobile devices or Android emulators. For each connected mobile device or emulator, an analysis worker instantiates a testing process that (1) fetches APKs in the analysis task, (2) executes APKs and interacts with them via generating user interactions, and (3) collects tagged execution Logcat logs [8]. Note that each Android device runs a revised AOSP, which leaves specified tags for ad fraud-related execution logs. These marks play an important role in pruning unnecessary logs for storage at the log database. The revised AOSP logic leaves a tag when a target app invokes HTTP(S) outgoing request APIs or Intents. Furthermore, the revised AOSP framework also leaves logs of actual parameter values and stack traces, which enable FraudDetective to compute full stack traces for observed suspicious behaviors. Each full stack trace holds causality information, indicating whether a user initiates each observed behavior. That is, this revised AOSP plays the role of emitting Android internal information to an analysis worker, which helps identify which user actions and libraries have committed ad fraud.

FraudDetective supports analyzing apps in parallel to facilitate detection. The Task Scheduler is able to control multiple analysis workers, and each worker is able to control heterogeneous Android mobile devices. For the prototype of FraudDetective, we used eight Android Pixel 2 devices.

Finally, the ad fraud detector module verifies the stored full stack traces for each task and reports identified ad fraud activities as well as the in-app modules responsible for invoking the identified fraud activities.

V. Design

We define an ad fraud activity that FraudDetective aims to detect (§V-A) and describe how FraudDetective computes a full stack trace (FST) for each observed fraud activity, a sequential list of multiple stack traces leading to the ad fraud activity (§V-B). The section then describes how we augmented an AOSP to compute such $FSTs$ (§V-C) and how FraudDetective precisely detects ad fraud, given an FST and its corresponding ad fraud activity (§V-D). Lastly, we describe the testing policy employed by FraudDetective to trigger ad fraud activities (§V-E).

A. Ad fraud activity

FraudDetective is designed to detect an *ad fraud activity*, which is a sensitive Android API invocation with a parameter invoking ad fraud without involving user interaction. Consider the invocation of `new URL(adClickUrl)`

Sink method name
android.app.Activity.startActivity()
android.app.ContextImpl.startActivity()
android.app.Fragment.startActivity()
android.content.ContextWrapper.startActivity()
java.net.HttpURLConnection()
org.apache.http.client.methods.HttpRequestBase.setURI()
android.webkit.Webview.loadUrl()
android.webkit.Webview.reload()
android.webkit.Webview.goForward()
android.webkit.Webview.pageUp()
android.webkit.Webview.pageDown()
com.android.webview.chromium.WebViewContentsClientAdapter.onLoadResource()

TABLE I: Sensitive Android APIs that invoke ad fraud activities.

`.openConnection()` triggered without user interaction. Because it sends a click URL request to an ad service without explicit user interaction, FraudDetective considers this invocation to be an ad fraud activity. Similarly, a non-user-initiated invocation of the `startActivity` [7] API with an Intent value that invokes other apps with URLs is also considered to be an ad fraud activity.

To precisely detect ad fraud activities, it is crucial to check for the existence of user interaction that causes the ad fraud activities. For this, we designed FraudDetective to emit *ad fraud candidates* for sensitive API invocations and to compute a full stack trace (FST) for each candidate, which helps determine the existence of user interaction.

Formally, an *ad fraud candidate* is an executed invocation statement that calls one of the predefined sensitive Android APIs with an actual parameter indicating ad fraud. Note that this candidate does not model whether a genuine user input triggers this invocation. In this paper, we abbreviate an *ad fraud candidate* and a confirmed *ad fraud activity* as an FC and an F, respectively.

By definition, the computation of an FC requires a list of sensitive Android APIs and actual parameters for each API invoking ad fraud. In this paper, we focus on identifying ad fraud that involves transmissions of click URL requests and invocations of other apps via cross-app Intents.

Sensitive Android APIs. Table I shows a list of 12 sensitive Android APIs that FraudDetective monitors. Four of them involve Android Intent invocations, while six of them instantiate WebView instances. To compile the list, we investigated Android API references [3], [4] as well as 15 mature apps and checked which APIs have been used for sending click URL HTTP(S) requests to ad services.

Argument patterns. FraudDetective requires the specification of the click URL patterns used in the invocations of the sensitive Android APIs in Table I. We conducted a preliminary study to generalize common click URL patterns. We first searched for ad SDKs of which SDK descriptions and source code are available from the Internet, thereby collecting 20 ad SDKs. Among them, we identified click URL patterns from the descriptions of five ad SDKs, namely, Adjust, AppsFlyer, Kochava, Tune, and LinkMine [2], [55]. For seven other ad SDKs, namely, AppLovin [18], Facebook [35], Unity Ads [76], AdMob [40], MoPub [60], TNKFactory [75], and Cauly [20], we integrated each ad library with our testing app and observed click URL patterns by clicking banner and full-screen ads.

From the collected click URLs of these 12 ad networks, we devised click URL patterns. We implemented regular expressions that (1) check for the existence of at least one /click, /clk, or /aclk token in a given URL path and (2) check whether the number of URL parameters is over eight. Furthermore, it checks whether the domain of a given URL is among the ad networks listed in EasyList [34] or NoTracking [62].

We acknowledge that our approach is a heuristic based on a limited number of ad SDKs. However, note that FraudDetective is able to accept arbitrary URL patterns that auditors want to monitor. Once a URL pattern is secured, FraudDetective is able to report an app module that sends URLs matching the given pattern. One possible way of identifying unknown click URLs is to leverage a machine learning classifier, which MAdFraud deploys when identifying click URLs. However, this approach suffers from false positives and negatives depending on the collected training instances. Instead, we focus on devising a regular expression that gives no false positives.

We further specified FCs to be invocations of cross-app Intents that call other apps. Instead of focusing on a limited number of popular apps, including YouTube, Google Play Store, and major browsers, we designed FraudDetective to detect apps that invoke other apps besides themselves.

Note that FraudDetective leverages a revised AOSP (§V-C) to leave an FC in Logcat execution logs. From each FC, FraudDetective also extracts (1) the source class of an invoked statement FC as well as (2) the FST that shows all the methods used (and their classes) to reach the FC from a program entry. This additional information helps FraudDetective determine (1) whether an FC originates from a hosting app or one of its libraries and (2) what user interactions trigger the FC, thus pruning false positives of ad fraud (§V-D).

B. Full stack trace

For each FC, there exists an FST from a program entry to the invocation of a sensitive Android API. We model a full stack trace FST as the sequence of all the preceding transitive callers of its FC from a program entry. Therefore, each FST contains all the callees to reach its FC from its entry point, and the last element of the FST is the invocation statement calling a sensitive Android API with an actual parameter indicating ad fraud.

Note that an FST can be one stack trace (ST) reaching an FC or merged multiple STs to reach an FC from a program entry point. For instance, when a target app uses a new thread to invoke an FC, the stack trace at the invocation of the FC does not contain an original program entry. Thus, FraudDetective should connect such fragmented STs to compute a complete FST. Section V-C provides further relevant details.

By design, an FST captures how a user input invokes its corresponding event handler. For example, the left-hand side of Figure 3 shows an FST that is a single ST from the program entry, `com.android.internals.os.ZygoteInit.ma`

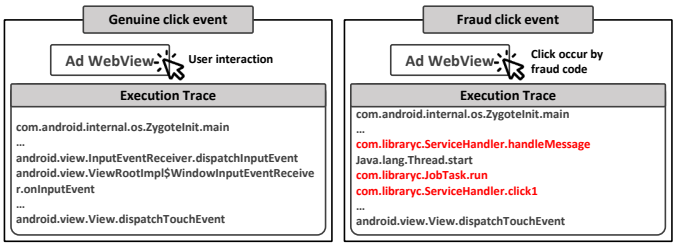

Fig. 3: Examples of two stack traces: One triggered by a genuine user touch and the other triggered by a forged touch event.

in, to the touch screen motion event handler, `android.view.View.dispatchTouchEvent`. Because all the invoked methods that precede `dispatchTouchEvent` belong to Android system classes, this FST is triggered by an authentic user touch. On the other hand, the FST in the right-hand figure shows several classes belonging to Library C, a third-party ad library that precedes the invocation of `dispatchTouchEvent`. This means that the touch event handler is forcibly invoked by `com.libraryc.ServiceHandler.click1` in this ad library, which denotes an *ad fraud full stack trace (AFST)*.

When a touch event occurs, the event is delivered from an Android Activity to a View instance by `dispatchTouchEvent`. This `dispatchTouchEvent` is a controller that decides how to route touch events [11], [12], [72]. Specifically, the `dispatchTouchEvent` processes 16 types of user action events, including touch, drag, move, button press, and scroll. Note that the Android OS always invokes this `dispatchTouchEvent` when the aforementioned user actions occur in any Android Activity windows [72]. Therefore, unforged stack traces leading to this `dispatchTouchEvent` invocation hold only Android internal classes, not developer-defined classes.

We leveraged this hierarchical call pattern to identify genuine user touch events. For each FC and its FST, FraudDetective checks for the presence of user- defined code in the FST that precedes the invocation of a `dispatchTouchEvent`. If found, FraudDetective labels this FST as an ($AFST$), and its FC becomes an F. The presence of an $AFST$ indicates the occurrence of an F.

When FraudDetective computes an FST, this FST contains all the STs in which the FST involves a user interaction, such as touch and drag. If the FST does not involve any user input, the FST only captures the last ST—the one closest to the FC. That is, FraudDetective focuses on tracking $FSTs$ that involve user interaction via using `dispatchTouchEvent`.

Note that an $AFST$ is a key technical component that contributes to FraudDetective identifying the causal relationships between user inputs and ad fraud activities as well as pinpointing the source classes of observed ad fraud activities, which MAdLife and MAdFraud [21], [25] did not address.

C. Augmenting an AOSP

The objective of augmenting an AOSP is three-fold: (1) to leave Logcat execution logs indicating FCs, (2) to connect fragmented STs generating a complete FST, and (3) to identify source classes in which FCs occur. To this end, we augmented the AOSP 8.1 (Android Oreo) and revised 793 LoC in the original AOSP as well as the Chromium Android WebView library.

Ad fraud candidates. As Table I shows, we revised eight Android APIs that send HTTP(S) requests, thus leaving tagged execution logs that list called methods with destination URL addresses. We also revised four Intent methods to leave an actual Intent parameter upon their invocation. Thus, an Android device with the revised AOSP leaves an FC in Logcat logs. For each FC, the revised AOSP also emits its corresponding FST. This FST is a sequential list of fragmented STs, each of which is a stack trace of callees. Thus, the last callee of a previous stack trace is responsible for invoking the first callee of the next stack trace in this FST. The ad fraud detector module harnesses these computed $FSTs$ to prune FCs originating from genuine user inputs.

Connecting fragmented STs. We conducted a preliminary study to identify which cases fragment a complete FST into multiple STs. We investigated three ad libraries, namely, MoPub [60], Cauly [20], and AppLovin [18]. We observed four cases that require connecting fragmented STs: (1) an FC invoked in a new thread separated from a main app thread; (2) an FC invoked in a thread in the thread pool managed by Android concurrent queues; (3) an FC invoked by an `android.os.Handler` class exchanging a message among threads; and (4) an FC invoked in a Chromium WebView instance of which the logic is implemented in a separate external library and not a part of the AOSP.

Figure 4 depicts the first three cases of simplified example code required to link fragmented STs. In the first case, we observed that two ad libraries create a new thread upon a user interaction and send ad requests within this new thread. For instance, an `OnClick` event handler invokes a new thread, and this new thread executes an FC that invokes a sensitive Android API. At this point, the ST of the FC via `new Exception().getStackTrace()` does not capture the `OnClick` event handler accepting a user touch event. To compute a non-fragmented FST, we revised the `java.lang.Thread` class in the AOSP. The revised AOSP remembers a ST that instantiates a new thread, thus enabling the mapping of each new thread to its parent ST in which this thread was created. We also implemented a global hash table that maps a given thread ID (TID) to its parent ST. We revised `java.lang.Thread.start()` so that when a new thread starts, its parent thread propagates the current ST to this child thread, and the child thread stores the delivered ST with its current TID in the hash table before it starts. Therefore, FraudDetective connects the ST of an FC in a new thread to the ST of its parent thread.

We also observed that several ad libraries recycle an existing thread in a thread pool instead of creating a new thread. This engineering practice makes the previous approach ineffective because of this type of augmented system method, such as `java.lang.Thread.start()`, cannot link a parent thread in which a user event happens to another recycled thread in which the FC occurs. However, these libraries do use an Android concurrent queue, such as `java.util.concurrent.SynchronousQueue`, to wake inactive threads in a thread pool. Thus, we revised

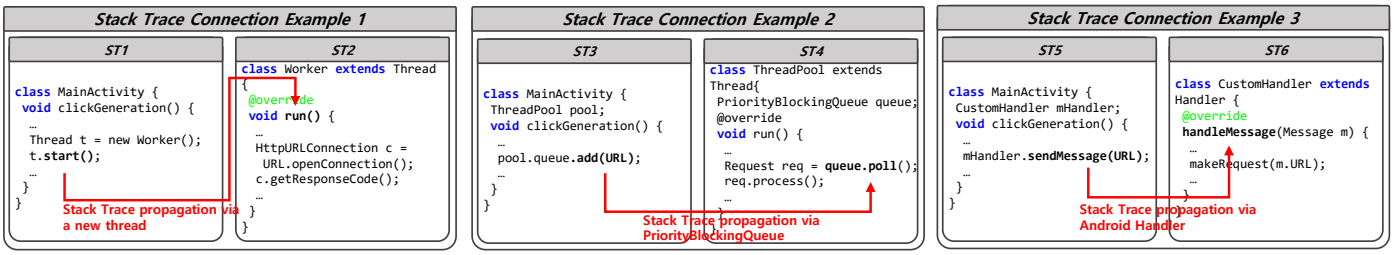

Fig. 4: Pseudo-code examples of ST connections via AOSP modification.

concurrent queue classes to propagate the ST of a parent thread that sends a wake-up signal to a recipient thread that accepts this signal. We created a wrapper class for elements in the concurrent queues. This wrapper class has an element field that references the original element and another field that stores the ST information. The functions that move and change queue elements, such as dequeue, enqueue, comparator, heap, indexing, and remove, have been revised to use the wrapper class instead of that of the original element; this process enables each element to correspond to the ST of the thread that enqueues the element. Therefore, when a thread that performs the enqueuing of an element is different from the thread that conducts the dequeuing of the element, the ST is delivered to the dequeuing thread. Therefore, FraudDetective is able to connect multiple STs fragmented over different threads that share one thread pool. The middle example in Figure 4 corresponds to this case. The parent $ST3$ is linked to $ST4$ because the thread is invoked upon receiving a URL via `queue`, a shared priority queue.

The third case involves using the `android.os.Handler` class. Developers usually use this class to make a scheduled runnable thread. This runnable thread is later invoked upon the delivery of messages over an `android.os.MessageQueue` class instance. Because an ST in this type of invoked runnable thread is separate from the ST of the parent thread sending a wake-up signal, we revised the AOSP code of `android.os.MessageQueue` and `android.os.Message` as follows. We modified `android.os.Message` to store the ST of the thread that enqueues a message to an `android.os.MessageQueue` instance. When the message is dequeued, the ST in this message is also delivered to the recipient thread that dequeued the message. For the above process, we modified the enqueue and dequeue functions, `android.os.MessageQueue.enqueueMessage` and `android.os.MessageQueue.next`. The right side of Figure 4 represents this last case, which links $ST5$ and $ST6$ via `android.os.Handler`.

The last case involves a Chromium WebView instance, which is not a part of the AOSP. Android apps often instantiate a Chromium WebView instance via the `android.webkit.WebView` interface and load a web page that involves loading various sub-resources, including JavaScript files, web pages within iframe instances, images, and others. Unfortunately, the augmenting AOSP cannot capture these sub-resources loading within this given web page because such loading occurs in Chromium WebView. To connect the ST of a thread that loads a web page to an FC that fetches sub-resources in Chromium WebView, we revised the Android Chromium and WebView client source code as follows. We

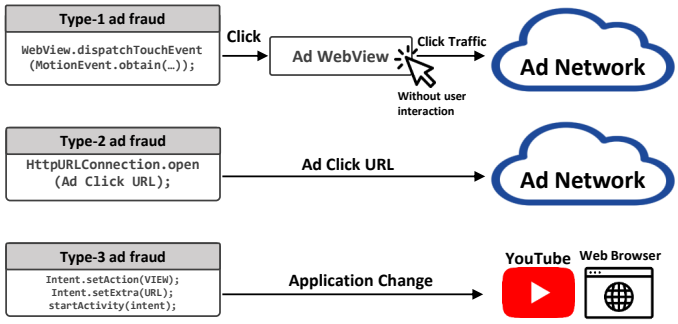

Fig. 5: FraudDetective considers three types of ad fraud, each of which occurs without user interaction. Type-1 ad fraud programmatically generates a forged click and lets the app send click URL requests. Type-2 ad fraud occurs when the fraud code calls `HttpURLConnection` to generate an ad click URL request without any user interactions. Type-3 ad fraud forcibly causes the victim to visit a target URL using YouTube or web browsers via an Android intent.

revised the `OnLoadResource` function in both Chromium and Android WebView, which is always invoked when loading resources, to collect the URL of each loaded resource. We also revised the `loadUrl`, `goBack`, `reload`, and `postUrl` methods of the `android.webkit.WebView` AOSP class to capture the ST of a thread in which the app loads a web page. In order to map this ST to a sub-resource loaded in Chromium, we created an interface that sends and receives data between `android.webkit.WebViewClient` in AOSP and `chromium.WebViewContentsClientAdapter` in Chromium. We have FraudDetective to send the ST for each sub-resource to the AOSP side and then connect this ST to the ST that invoked a web page loading via a Chrome API invocation.

D. Detecting ad fraud activities

For a given pair of an FC and its FST, FraudDetective identifies an F. To this end, the ad fraud detector checks whether the FST originates from a forged interaction or a genuine user interaction, such as user touch or drag. In this paper, a forged interaction refers to an artificial interaction via programmatic `dispatchTouchEvent` invocations that mimic a genuine user touch.

We designed the ad fraud detector to identify three types of ad fraud activities, as shown in Figure 5. A Type-1 F refers to a click URL transmission triggered by a forged user click on a WebView instance rendering ad impressions. A Type-2 F

is a click URL transmission triggered without user interaction. Lastly, a Type-3 F is an invocation of other mobile apps via a cross-app Intent, which brings them into the foreground without user interaction.

To identify a Type-1 F, FraudDetective checks all the methods in $ST \in FST$ to see whether there exists a `dispatchTouchEvent` invocation. If so, FraudDetective then checks all the source classes of the method calling this invocation. When any of these classes is not an Android internal class, FraudDetective determines that the FST has been triggered by a mimicked interaction because either the app itself or one of its libraries has forged this touch interaction.

For a given pair of an FC and its FST, FraudDetective classifies the FST as a Type-2 or Type-3 $AFST$ when there is no user interaction event handler in the FST. This means that this FST does not originate from any user interaction. When an F occurs via sending a click URL request, FraudDetective labels this FST as a Type-2 $AFST$. A Type-2 $AFST$ contains only the last ST. When an FST is to send a cross-app Intent and the argument of this Intent invocation calls other apps, FraudDetective labels this FST as a Type-3 $AFST$. FraudDetective only collects the last ST in the FST when there is no user interaction.

Interestingly, we observed several ad libraries that invoke not only mobile browsers, including the Chrome and Samsung mobile browsers, but also a mobile app associated with a popular portal website, Naver [61]. The goal of these ad libraries is to cause users to search using certain keywords in the Naver mobile app, thus manipulating portal search rankings by exploiting victims' mobile devices (§VII-B).

Modules responsible for detected ad fraud. Remind that FraudDetective computes an $AFST$ for each detected F. From this $AFST$, FraudDetective classifies whether the app or its embedded third-party module is responsible for executing the F. FraudDetective first checks whether the app itself invokes the F. It first computes the longest common prefix between the source class that invoked the F and the app package name. When the namespace nesting of this common prefix is two or deeper, such as `com.musicpackage`, FraudDetective classifies that the given F occurs due to the app itself. Otherwise, FraudDetective deems that the F is due to a third-party library. The employed heuristic can produce false reports when a hosting app or third-party libraries obfuscate their class names via ProGuard [45].

To validate the efficacy of this simple policy, we manually confirmed whether third-party modules were indeed responsible for executing the fraud detected by FraudDetective in the 73 apps that FraudDetective detected (§VI-B). However, it is possible for malicious ad-serving web pages to exploit such third-party modules to initiate ad fraud (§VII-A). In this case, FraudDetective is still able to attribute an exploited module within the app to the observed ad fraud, thereby helping auditors to investigate root causes external to the app itself.

For each $AFST$, we extracted three classes within the $AFST$ that appear ahead of triggering the F. We then checked whether these identified classes were from third-party libraries via the following three methods: 1) we decompiled each identified class and manually checked whether each matched open-sourced ad SDKs; 2) we checked whether the decompiled code of each identified class appeared across the 74 apps that FraudDetective detected; and 3) we confirmed that each identified class did not reference any classes that belonged to the app itself.

When matching classes to confirm their equivalence, we compute signatures for these classes based on the argument and return types of member functions in the classes. For a given class, we extract the argument and return the value types of all the member functions and then concatenate these types for its signature. Note that Bakes *et al.* [19] suggested the employed signature-based method for identifying third-party libraries and claimed its resiliency to common code obfuscations, including ProGuard.

E. Task scheduling and dynamic testing

The Task Scheduler assigns an analysis task to each analysis worker. An analysis worker conducts dynamic testing as the task describes.

Task scheduling. In the default setting, a task specifies five Android APKs to test with a testing duration of 1,500 seconds. To reflect app usage patterns of real users, all workers rotate apps in the foreground every 15 seconds. This means that one in five applications is always running in the foreground while the other applications run in the background. Thus, each application has 20 chances to run in the foreground, where it runs for 300 seconds. This round-robin testing strategy helps cover realistic usage scenarios, such as the execution of apps in the background and multiple executions of the same app.

Analysis worker. A worker manages an ADB USB connection with each mobile device. Because this ADB connection can be disconnected due to the long testing time, the worker monitors its USB connection status and reconnects when a disconnection occurs.

A worker also conducts dynamic testing while executing a testing process. FraudDetective leverages an Android UI Automator [9] to perform user interactions. FraudDetective also uses ADB commands in order to trigger certain device events, such as battery status changes. The testing process performs the following six actions in random order.

- Turn off the screen, wait for two seconds, and then turn on the screen.
- Press the home/back button, wait for one second, and then press the recent button to go back to the application [6].
- Press the volume down button once, wait for one second, and then press the volume up button once. Repeat this procedure five times.
- Change the battery charging status (charge off and on,) and change the value to 50%, 15%, and 5% at two seconds intervals.
- Open the Android notification bar and close it in one second.
- Rotate the screen 90 degrees left, 90 degrees right, and to the up-side-down landscape position at two second intervals.

Note that Android malware checks whether the underlying Android OS is on an emulator, and it often reveals its ill intent behaviors when it is highly likely that authentic users are using their devices [1], [51], [77]. Thus, we mimicked

# of download	∼5K	5K ∼ 100K	100K ∼ 500K	500K ∼ 1M	1M ∼ 5M	5M ∼ 10M	10M ∼ 50M	50M∼	Total
# of apps	7,368	10,828	8,573	5,311	8,793	3,877	2,426	996	48,172

TABLE II: Number of collected Android apps with download numbers.

normal use cases using real Pixel 2 devices and invoked daily system events, such as volume down/up and battery charging alerts. Also, Shirazi *et al.* demonstrated that device users often watch their smartphones in mobile and trigger the landscape mode [68]. Thus, we also included such behaviors for testing.

Furthermore, to increase dynamic testing coverage, FraudDetective automatically passes any system/custom consent or full-screen consent panels by pushing the consent or close buttons in the current activity. Because a system consent (Android permission consent) or full-screen consent panel often blocks the execution of a target app, FraudDetective uses the UI Automator to parse the current screen's UI information and push buttons with "consent," "yes," "ok," "agree," "confirm," "go," "continue," "start" and ohter messages to pass pop-up dialog panels. Note that this ability to conduct dynamic testing stems from FraudDetective computing of $FSTs$. Each FST carries its user interaction source, which enables FraudDetective to identify the F.

VI. Evaluation

A. Experimental setup

We conducted experiments on two host machines running 64-bit Ubuntu 18.04 LTS with Intel i7 8700 (3.2GHz) CPUs and 16GB of main memory. One host implemented the Task Scheduler, which distributes analysis tasks. The other one was an analysis worker that detects ad fraud activities by testing apps specified in assigned tasks. This worker was connected with eight Pixel 2 devices.

Crawled mobile apps. We collected 48,172 Android apps from Google Play Store. We collected these apps via two crawling methods. The first method is to use the Google Play unofficial Python API [31], which enables crawling the top 100 ranked apps from each of the 35 Google Play categories. From April 2019 to September 2020, we collected 10,024 apps. To cover less popular apps, we also randomly sampled additional 38,148 apps from APK mirror sites [13], [14], [15]. Note that we deliberately selected these mirror sites because they only mirror authentic apps from Google Play Store. Table II shows the popularity of the crawled mobile apps.

B. Ad fraud

FraudDetective classified each identified F as one of the three ad fraud types (§V-D). To test a total of 48,172 apps, FraudDetective took approximately 36 days with eight Pixel 2 devices. Table III shows the number of detected mobile apps that commit ad fraud with their fraud types. Specifically, FraudDetective found 34,232 Type-2 fraud activities, corresponding to the existence of 34,232 click URL requests triggered without user interaction. From each app that committed ad fraud, FraudDetective detected an average of 497 ad fraud requests per app.

Type	# of records (# of apps)	Responsible module		
		Module	# of apps	Ratio
Type-1	0 (0)	App	0	0%
		Library	0	0%
Type-2	34,232 (66)	App	1	1.5%
		Library	65	98.5%
Type-3	221 (8)	App	0	0%
		Library	8	100%

TABLE III: Number of detected apps of each fraud type and responsible modules.

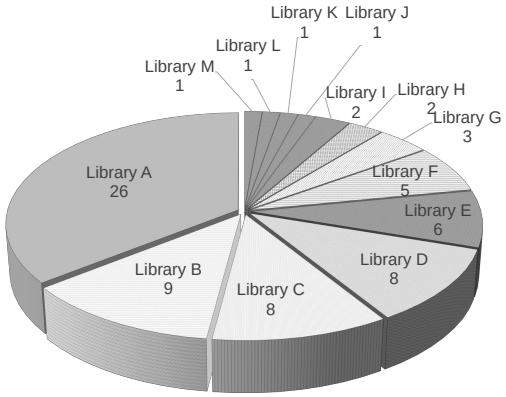

Fig. 6: Fraudulent app distribution by ad libraries committing ad fraud.

Table III also presents the number and percentage of libraries that are responsible for each type of identified ad fraud. A responsible module refers to a class within an app committing observed ad fraud activities (§V-D). Of 66 apps committing Type-2 ad fraud, FraudDetective reported that 98.5% of the observed fraud activities stemmed from third-party libraries. Figure 6 shows the distribution of third-party libraries that intentionally or unintentionally committed ad fraud in the 73 apps, which include eight apps that committed Type-3 ad fraud, while the others committed Type-2 ad fraud.

FraudDetective reported that click URL requests originated from Library A without user interaction in 26 apps. Considering that Library A is a popular ad service provider with a large user audience, we further analyzed the reported $AFSTs$ in these 26 apps and found a new fraud case that abuses Library A. FraudDetective reported that click URL requests originated web pages from Library A WebView instances without user interaction. The web pages included `<link>` tags that generated more than 40 ad click URLs. That is, the abuser designed an ad campaign that serves ad impressions that generate click requests, thus committing ad fraud without user interaction. We describe this in greater detail in Section VII-A.

Meanwhile, Library B and C intentionally committed ad fraud by creating an ad WebView instance that does not display on the main screen. This invisible WebView instance loads click URLs, thereby sending click requests to various ad services without user interaction. Section VII-B provides further details. Libraries D, G, J, and K also intentionally committed ad fraud by generating ad click requests using `java.net.HttpURLConnection`. FraudDetective also re-

# of download	∼1K	1K∼ 50K	50K∼ 500K	500K∼ 5M	5M∼ 100M	100M∼	Total
# of apps	6	8	16	15	22	7	74

TABLE IV: Number of detected fraudulent apps with download numbers.

ported that Libraries F and H generated click URL requests using the WebView `loadURL` function.

Table IV shows the number of downloads for the detected fraudulent apps, which illustrates the impact of the detected ad fraud. There exist 29 apps with more than five million installs, thus demonstrating that today's ad fraud detection systems require further improvement to prevent abusers from victimizing benign app users. Furthermore, the new abuse case in Library A motivates a thorough examination of ad impressions and their outgoing requests.

FraudDetective also observed eight Type-3 ad fraud activities that involve cross-app Intents from eight apps. These apps invoke other apps without any user interactions.

One app using Library B used cross-app Intents to invoke a browser app with a specific page or the YouTube app with a specific video in the foreground. The app redirects users to several webpages promoting Bitcoin websites and a cosmetics sales blog. Section VII-B further describes its fraudulent behaviors. Two other apps also created a cross-app Intent. The Intent action is `android.intent.action.VIEW`, which invokes the default browser with a URL. The URLs of the Intent are all the same: `http://www.fofy.com/red.php?utm_source=1`. When we visited this URL, the final landing page was from a subdomain of `http://www.fofy.com`, which rendered Google display ads. This means the above Intent action is able to generate ad impression revenue. Note that `http://www.fofy.com` has been variously reported by the spyware and malware reporting community as forcing users to visit their webpage [53], [58]. FraudDetective concluded that Library E committed this ad fraud behavior. The remaining five apps created an Intent that invokes Play Store, which promoted mobile apps without any interactions.

These experimental results yield the new insight that several ad service providers actively commit ad fraud to promote specific websites or products in which users have not expressed interest. That is, these ad service providers victimize their publishers as well as the users of apps from these publishers to increase ad click and impression traffic. We also emphasize that FraudDetective contributes to pinpointing fraudulent third-party modules inside these identified apps, which can help app auditors not only to understand observed ad fraud but also to propose that these app developers change their ad libraries.

Responsible disclosure. We reported all of our findings to Google as well as identified ad library vendors to address the identified fraud behaviors.

Ad fraud confirmation. We investigated the current status of 74 fraudulent apps that FraudDetective reported. As of September 15, 2020, among the 74 apps, 19 apps had been removed from Play Store, and 49 apps have been updated. We further analyzed whether the latest versions of these 49 updated apps have removed their ad libraries committing ad fraud. We confirmed that 22 out of the 49 updated apps removed the identified ad libraries, demonstrating the correctness of our identification results.

# of Apps	MAdFraud	MAdLife	FraudDetective
# of Test Apps	165K†	143K‡	48K‡
# of Fraud Apps	21	38	74

† Malware + Third-party Stores
‡ Google Play Store

TABLE V: MAdFraud [25] vs. MAdLife [21] vs. FraudDetective (click fraud).

C. Comparison with previous studies and false negatives

We compared our experimental results with those of MAdLife [21] and MAdFraud [25] in Table V. FraudDetective analyzed fewer applications than the previous tools did but found more fraudulent apps. Note that the numbers in the first and second columns are from their findings [21], [25].

Because we were unable to obtain the source code of MAdFraud and MAdLife, we analyzed how many of the 74 apps that FraudDetective detected would have been missed by these tools. Note that MAdLife mainly finds abusive clicks on WebView instances. However, FraudDetective is able to find click fraud that does not involve any user click but still sends click URL requests. Both MAdFraud and MAdLife use Android emulators for dynamic analysis. On the other hand, FraudDetective leverages real devices and triggers various events, which helps increase dynamic testing coverage.

MAdFraud. In order to find 36 out of the 74 fraudulent apps, it was necessary for FraudDetective to pass permission or consent windows when the apps started. Because MAdFraud does not interact with apps, it is unable to find these 36 apps. Note that MAdFraud was built upon the Android emulator in 2014 of which the version is below 6.0 and which does not support dynamic system permission. Therefore, MAdFraud did not need to pass any of these system permission windows for further execution. However, we observed that 28 apps presented their own custom consent windows and start-up ads that block execution, which necessitates explicit user interaction for further execution. MAdFraud is also unable to detect two apps committing ad fraud involving cross-app Intents without user interaction because it only monitors outgoing click URL HTTP requests. Thus, it misses requests for promotional web pages from other mobile apps, including browsers and YouTube.

MAdLife. We contacted the authors of MAdLife and received the names of packages and their app versions for 38 apps in which they found click fraud behaviors in their previous study [21]. Of the 38 apps, we checked 30 apps with FraudDetective, excluding two apps for which APKs are not available on the Internet and six apps in which we did not observe any fraudulent behaviors due to their deprecated services.

FraudDetective successfully reported 30 fraudulent apps with no false negatives. Of the 30 apps, FraudDetective identified 29 fraudulent apps with the same fraud behavior; they invoked a cross-app Intent with the `android.intent.action.VIEW` action, which invokes a default browser with *Fofy* and *Leadbolt* ad network URLs without involving user interaction. Unlike the 74 fraudulent apps that FraudDetective identified from our dataset (§VI-B), the

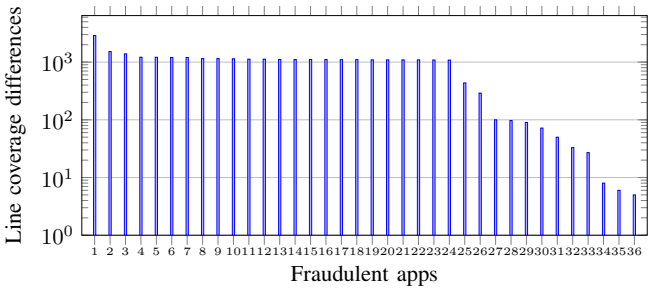

Fig. 7: Line coverage differences between the environments with and without user interaction. The Y-axis represents a coverage difference: *line coverage with interaction while testing the app − line coverage without interaction while testing the app.*

observed $AFSTs$ originated from Library E and another third-party library. FraudDetective also identified that the remaining app fired a cross-app Intent that invoked a default browser with the website of the app itself. This website served the same service as the app, thus making this report a false positive.

The experimental results demonstrate that FraudDetective is able to detect all of the ad fraud cases that MAdLife found in its previous study. Furthermore, FraudDetective reported that these fraudulent behaviors originated from the apps themselves.

False negatives. To assess false negatives, we further collected apps with known click fraud behaviors. We searched through various sources, including recent press releases about identified fraud campaigns [23], [50], [63]. However, we were only able to collect six fraudulent apps, each of which was downloaded over 10M times from Google Play Store. Note that it is often difficult to know the specific versions of fraudulent apps, and they are often unavailable from public app stores, including APK mirror websites. Lastly, many known fraudulent apps were found to no longer exhibit fraudulent behaviors due to the deprecation of their C&C servers.

Among the six known fraudulent apps, FraudDetective detected all of the apps that commit Type-2 fraud, producing no false negatives. FraudDetective also reported that the module responsible was Library D, which generated 194 click URL requests from these six fraudulent apps. When merging these six apps with the 30 apps identified by MAdLife, FraudDetective identified all of these fraudulent apps, reporting no false negatives.

D. Efficacy of user interaction

We evaluated the degree to which conducting user interactions in FraudDetective contributes to increasing testing coverage and improving the identification of fraud activities.

Line coverage. We measured the line coverage of a given APK using ACVTool [66]. We tested a given app for five minutes with and without the user interactions described in §V-E. Since ACVTool does not support APKs using Multidex, we tested 36 apps among the 112 fraudulent apps that FraudDetective and MAdLife found.

Figure 7 shows the difference from covered lines with user interactions to those without user interactions for each app. The differences vary from 5 to 2,885 lines; user interactions contributed to increased coverage of 853.39 lines on average.

Operation	Avg. Execution Overhead
Resource loading in `WebView`	0.916 ms
HTTP request in `HttpURLConnection`	0.224 ms
Activity change using `Intent`	0.231 ms
Touch event	0.392 ms
ST propagation via a new `Thread`	0.731 ms
ST propagation via `PriorityBlockingQueue`	1.145 ms
ST propagation via `Handler`	0.883 ms

TABLE VI: Execution overhead of FraudDetective for each operation that we modified in the AOSP framework.

Fraud activities. We checked whether user interactions contributed to finding fraud activities. Among the 112 fraudulent apps consisting of 74 apps that FraudDetective found and 38 apps reported by MAdLife, we were able to reproduce fraud activities in 73 apps. At the time of conducting this evaluation, 25 apps were deprecated, and click fraud campaigns became dormant in the other 14 apps.

Of the 73 apps, FraudDetective required no user interactions to find 23 apps. Among the remaining 50 apps, 36 apps required bypassing startup ads or custom consent windows, seven required consent to Android permission system windows, five required bypassing both of permission system and custom consent windows, and two required bypassing all three of the aforementioned windows.

We observed that the transition of a testing app from the background to the foreground helped close startup splash ads and contributed most to increasing line coverage and finding more fraudulent apps. We further describe the interesting case study of Library B, which attempts to avoid fraud detection in §VII-C.

E. Finding ad fraud with Android emulators

To test the efficacy of FraudDetective in detecting ad fraud using Android emulators, instead of real devices, we set up FraudDetective to test the 74 fraudulent apps identified using the Android Virtual Device (AVD) emulator [10]. Out of the 50 apps, we were unable to install 19 apps in an AVD device due to their usage of ARM native libraries. Furthermore, we observed that the AVD emulator environment drastically slows down the execution of these 19 apps. Each fraudulent app took more than 20 minutes to instantiate, thus rendering infeasible the execution of the 19 apps in the AVD emulator.

Considering that MAdLife used the Genymotion Android emulator [37] for their dynamic testing, we also tested the Genymotion Android emulator with Android 8.1 Google Pixel 2 for FraudDetective. However, we were again unable to install the 19 fraudulent apps using ARM native libraries. The experimental results demonstrate that leveraging real mobile devices contributes to expanding the testing coverage of FraudDetective by supporting diverse apps developed in various execution environments.

F. Performance overhead

Table VI presents the execution overhead for each operation that we modified in the AOSP framework. We randomly sampled a total of 170 apps and tested each app for five minutes using FraudDetective. In this experiment, we triggered a click

Listing 1: An ad HTML page of Library A that generates click URL requests.

```
<html>
...
<body>
  <a href="" target="_blank"
      style="text-decoration: none;">
    <img src="" />
  </a>

  <link rel="stylesheet"
      href="https://tracking.appxigo.com/click
  /9840/18?ref_id=05EB1711-1EDF-44DC-9488-803D4D
  0547198f7eaaf1a6d14d6bb58b1cd2e164c6e0&sub_pub=
  7991f1375594407b84f8160e&app_name=ticketmaster&
  custom1=[click_id]8f7eaaf1a6d14d6bb58b1cd2e164
  c6e0-!-5e8bfaee-76015-e8bf-ee76-1c354534371
  -!-1588176638.180842-!-366e0279-1ee0-4433-
  9ca6-22b603745f81" />

  ...
```

event five times for each app testing instance because we modified the event handling logic in the AOSP framework. As the table shows, the performance overhead is negligible; all executions were completed within 1.2 ms on average.

FraudDetective generated an average of 1032.6 Logcat messages in five minutes, which took up approximately 2.29 MB. For analyzing a total of 48,172 apps, FraudDetective required a disk space of 224 GB.

VII. Case Studies

We present two representative ad fraud cases that are notable by the extent of their abuse and explain newly obtained insights into how attackers commit mobile ad fraud.

A. Case 1: Click fraud abuse of Library A ad impressions

FraudDetective reported 17 apps with Library A for sending click URL requests without user interaction; it also reported that the observed requests originated from the embedded Library A.

We investigated each identified FST and its F; each F was invoked from the WebView instances that Library A instantiates and uses for rendering ad impressions. Every observed click URL request stemmed from the same web page: `https://cpi-offers.com/fantastic.html`. Listing 1 above shows part of the web page content. It contained approximately 40 `<link>` tags, each of which embedded a different click URL, thus generating click requests when a WebView instance rendered the page. These click URL requests targeted six ad network services, namely, Appxigo Media, Ad4Game, g2afse, AppsFlyer, and Xapads.

In this case, the adversary abuses Library A to generate click URL requests without involving any user interactions. She designs an ad campaign that embeds payloads and leverages the ad network of Library A to victimize the app publishers of Library A as well as their users to generate a large volume of click traffic. The novelty of this observed attack is that the attacker directly exploits WebView instances in which ad impressions are usually rendered without any user interaction.

Package Name	# of Intents
com.android.chrome	34
com.naver.whale	8
mobi.mgeek.TunnyBrowser	8
net.daum.android.daum	8
com.nhn.android.search	28
com.cloudmosa.puffinFree	8
com.google.android.youtube	12
com.sec.android.app.sbrowser	8
org.mozilla.firefox	8
Play Store	11
default browser	88
Total	221

TABLE VII: Apps that commit impression fraud invoked via Intents.

In summary, the adversary exploited vulnerable Library A WebView instances to commit click fraud, and FraudDetective reported this in-app Library A WebView instance from which click URL requests are originated with exploiting web pages.

B. Case 2: Impression fraud via invoking third-party apps

FraudDetective identified two apps connected to Type-3 fraud activities. It reported that eight fraudulent apps generated 133 explicit Intents and 88 implicit Intents, respectively.

Table VII collates the target apps invoked via Intents initiated by ad fraud campaigns. `com.gmail.heagoo.appdm.adv` and `video.editor.no.watermark` invoked a default browser via the URL of `http://www.fofy.com/red.php?utm_source=1` without user interaction. FraudDetective reported that the identified $AFST$ is perpetrated by Library E.

On the other hand, `com.camera.catmera` fires explicit Intents that invoke prevalent apps, including Chrome, Puffin [43], Naver [61], Firefox, YouTube, and Play Store, thus bringing up one of the target apps without any user interactions. FraudDetective also reported one ad library responsible for the 133 ad fraud activities observed.

We believe that the motivation of this ad library is to increase incoming traffic toward designated web pages with the aforementioned mobile apps. When analyzing the URLs specified in these observed Intents, it appears that this ad library conducts a keyword search for specific restaurants by invoking the Naver browser. The keywords used include restaurant names and locations, which may increase the search rankings of the restaurant names in the portal website operated by Naver. This ad library also redirects users to the sign-up page of a Bitcoin trading site, a blog selling cosmetic products, and a YouTube video site promoting video games and golf lessons. We believe that these websites belong to advertisers who launched ad campaigns with the ad service provider using this ad library.

The ad fraud outlined above entails various side-effects. Although this ad library commits ad fraud and invokes other apps without user interactions, users blame the invoked apps, including Naver and YouTube, for promoting certain websites and videos. YouTube and Naver app pages in Google Play Store have negative comments from users complaining about

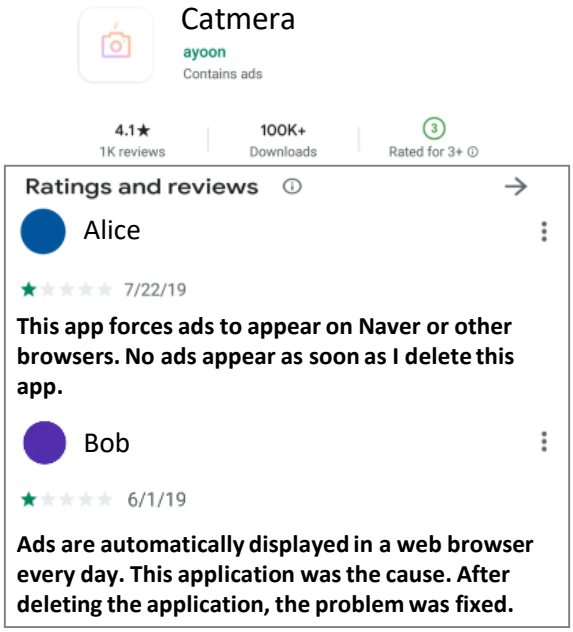

Fig. 8: Negative reviews of the `com.camera.catmera` app in Google Play Store: There were more than 30 complaints about advertising pop-ups occurring in a browser without user interaction.

the forcible switch into these apps without any user interactions. It is natural that when users are compelled to see unwanted promotional videos or messages from the Naver and YouTube apps, it adversely affects those brands. Figure 8 also shows comments from victims to the effect that these abrupt pop-ups of Naver and YouTube apps disappeared when they removed `com.camera.catmera`.

We reported the identified Library B, to Google and Naver. Law enforcement is currently conducting an investigation of this ad network vendor.

C. Case 3: Ad library avoiding fraud detection

We discovered code for avoiding fraud detection in Library B. We analyzed the source code by using JADX [71] to decompile apps that commit ad fraud. Library B commits ad fraud when all of the following three conditions are satisfied: (1) a device is not in the battery charging status; (2) a device does not have any of the specific apps listed in Table VIII; and (3) a device is in Wi-Fi connection status.

The first condition is related to detection in a device under testing that is connected to any USB port. Note that a USB connection is needed to control devices using ADB for automated testing. However, FraudDetective is able to find this app because it changes the battery status while conducting dynamic testing.

The second condition is to avoid devices with traffic monitoring apps. According to our manual analysis of the decompiled library, it does not show abusive behaviors when the apps in Table VIII exist in the device. Note that the five apps capture network traffic originating from the device. Thus, we concluded that these ad libraries hide themselves when there is a risk of being monitored by other security monitoring apps.

Application name	Purpose
Packet Capture	HTTP traffic capture
IP Tools	Network status monitor
Network Scanner	Network status monitor
Network Utilities	Network traffic monitor
tPacketCapture	HTTP traffic capture

TABLE VIII: Apps that an abusive ad library detects to conceal their abusive behaviors.

Package Name	# of Requests
`com.sisunapp.wisesaying`	1,192
`com.wtwoo.girlsinger.worldcup`	883
`com.somansa.factory_kyh`	393
`com.appsnine.compass`	386
`com.camera.catmera`	379
`com.pump.noraebang`	226
`kr.yncompany.myrecipes`	42
`com.serendipper16.chattinganalysis`	32
Total	3,533

TABLE IX: Apps that commit click fraud attributed to the same publisher due to one embedded library.

The last condition is related to preventing users from noticing the massive volume of cellular data usage caused by abusive apps. Because these apps generate ad click traffic even in the background, their data usage is obviously high, which could motivate users to remove them. Therefore, these apps limit themselves to conducting click fraud when the devices use Wi-Fi, which is relatively less restrictive in generating ad traffic.

D. Case 4: Click fraud traffic with common publisher identifiers

FraudDetective further analyzed the 34,232 Type-2 ad fraud activities (§VI-B) that send click URL requests to various ad services. Among the observed 34,232 click URL requests, we noted that eight different apps share the same app identifier in their click URL requests to the "LinkMine" ad network.

Table IX shows the eight identified apps and the number of observed ad requests committing click fraud with the same app identifier. All the requests in Table IX were issued from the same third-party ad library.

According to its ad SDK description [55], this app identifier refers to a publisher identifier that attributes user clicks to an app publisher. That is, this identified ad library leverages other apps to send click URL requests and allows all these requests to be attributed to a specific app publisher.

Considering that this app publisher gets paid more due to all the aggregated clicks from other apps, we believe that this app publisher is responsible for committing click fraud and has a strong connection with the third-party ad library committing Type-2 ad fraud activities.

VIII. Limitations

FraudDetective is designed to report in-app modules for

identified ad fraud, which helps app publishers patch their apps. Furthermore, this capability also helps auditors provide actionable notices to benign publishers using fraudulent ad libraries. FraudDetective enables this attribution of observed ad fraud via computing $FSTs$. However, FraudDetective has a limitation in its implementation regarding connecting fragmented STs. Note that we have modified `SynchronousQueue` and `PriorityBlockingQueue` to propagate information from one thread to another that share the same thread pool (§V-C). When a developer uses an unmodified concurrent queue among the five remaining concurrent queues for their app, FraudDetective will produce an incomplete FST, thus producing false positives. However, among a total of 28,160 ad fraud activities from the 74 apps, we observed no false positive. We believe that this implementational limitation is fixable with more engineering time and effort.

Another limitation of FraudDetective stems from its dependence on the click URL patterns with which it is supplied. We generalized click URL patterns from the manual investigations of seven major ad libraries and five major ad networks (§V-A). However, there still exist different patterns of click URLs that we did not capture, thus producing false negatives. One mitigation is to leverage a trained machine-learning classifier, similar to the MAdFraud method for classifying click requests. However, this approach also requires a training dataset that represents diverse click URL patterns to increase accuracy. Therefore, we leveraged a coarse regular expression that captures HTTP requests with many parameters having at least one "click" word. Then, via a source code audit, we double-checked whether the reported FCs were indeed ad fraud activities. We confirmed that all the reported 74 apps committed ad fraud via sending click URL requests without involving user consent.

Another limitation of FraudDetective is that it requires manual ad library identification. In the case of an ad SDK with open-source code, ad library identification can be performed automatically by comparing the source code with the decompiled APK source code. According to previous studies, it is possible to distinguish known third-party libraries with high accuracy [19], [22], [38]. This approach requires having the published code of ad SDKs against which a given code can be compared. In our evaluation, we observed five ad libraries of which the code and libraries were unavailable on the Internet. Therefore, for those five ad SDKs, we leveraged the names of classes responsible for ad fraud activities to deduce ad service names. When these class names are obfuscated, we find matching classes from other apps without obfuscation, thereby deducing ad service names from the matching classes without obfuscation.

IX. Related Works

A. Mobile ad click fraud

Previous studies of identifying click fraud in mobile advertising focus on developing dynamic testing frameworks. MAdFraud [25] ran Android apps with an Android emulator for 60 seconds each in the background and foreground while emulating no user interaction. It then found ad click traffic which occurred under the testing environment involving no user interaction. MAdLife [21] found that 37 Android apps always navigated to an ad's landing page without user interaction. This behavior of forcing users to go to an ad landing page by launching an Android app was found through the Genymotion Android emulator [37]. These testing tools played an important role in revealing the occurrence of mobile click fraud. However, they did not compute the causal relationship between the occurrence and the cause of ad click fraud.

Cho *et al.* investigated how effectively mobile ad networks responded to click fraud [24]. They developed ClickDroid, a mobile ad click bot, which clicks mobile ads periodically. ClickDroid attempted to avoid the detection of mobile click ad fraud in ad networks by modifying a device identifier each time it clicked on a mobile ad. A total of 100 clicks were performed through each of eight major mobile ad networks, and only two mobile ad networks detected traffic abnormalities, demonstrating ad networks' incapability of identifying click fraud.

B. Web ad click fraud

Ad network services have strived to detect ad click fraud by analyzing click fraud traffic patterns using ad fraud filters [27], [28], [41], [81]. Many filters have been studied, such as identifying a high click ratio on a specific website [28] or checking duplicate clicks on the same ad [81].

Ad fraudsters often make fraud profit using botnets that infect victims' hosts [64], [80]. ZeroAccess infected approximately 1.9 million host machines [59], generating approximately $2.7 million in monthly revenue, primarily through ad click fraud. The Federal Bureau of Investigation (FBI), European Cybercrime Centre (EC3), and ad network vendors, such as Microsoft, worked together to eradicate the ZeroAccess botnet and took legal actions [59].

Clickjacking has been reported an effective way of conducting click fraud [33], [48], [82]. Furthermore, Zhang *et al.* demonstrated that abusive third-party JS scripts have modified click URLs in their hosting websites (e.g., '`<a href="Click URL">`'), thus hijacking authentic users' clicks [82].

C. Analyzing ad libraries

Previous studies have investigated mobile ad libraries in their excessive permissions usages, aggressive collections of private information, and inherent vulnerabilities leading to private information leakage [29], [44], [74], [78]. To this end, researchers have proposed systems that restrict permission usages by ad libraries or separate ad library modules from its hosting app via isolating them in different processes [65], [69], [70]. Moreover, prior approaches proposed detection methods of identifying specific third-party libraries, which abuse their hosting apps and permissions [19], [22], [83].

X. Conclusion

In this paper, we design, implement, and evaluate FraudDetective, a dynamic testing framework for uncovering ad fraud. We compute the causal relationship between a user interaction event and an ad fraud activity and model it into a full stack trace. To compute these full stack traces in dynamic testing, we revise Android system classes and let a target app under

dynamic testing execute the revised system code, thus leaving execution logs. FraudDetective leverages these execution logs to determine whether observed full stack traces actually commit ad fraud without genuine user interactions.

FraudDetective found 34,453 observed ad fraud activities perpetrated by 74 apps, clearly demonstrating its efficacy in discovering ad abuse. It also reports that 98.6% of apps commit ad fraud by means of their ad libraries. This new insight suggests that app publishers and their users have become victims of ad fraud and invites further research on practical defenses to prevent these ad libraries from committing ad fraud.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their concrete feedback. We sincerely appreciate Seongil Wi for his support in polishing the paper. This work was supported by the Naver Corporation and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT), No.2020-0-00209.

REFERENCES

[1] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Emulator vs real phone: Android malware detection using machine learning," in *Proceedings of the ACM on International Workshop on Security and Privacy Analytics*, 2017, pp. 65–72.

[2] Amazon advertising, "Amazon advertising - urls for app advertising," https://advertising.amazon.com/resources/ad-policy/en/mmp-measurement-urls, 2020, accessed: 2020-05-01.

[3] Android, "Android api reference," https://developer.android.com/reference, 2020, accessed: 2020-05-04.

[4] Android, "Android code search," https://cs.android.com/, 2020, accessed: 2020-05-04.

[5] Android, "Android Open Source Project," https://source.android.com/, 2020, accessed: 2020-05-01.

[6] Android, "Recent screen," https://developer.android.com/guide/components/activities/recents, 2020, accessed: 2020-05-01.

[7] Android developer, "Android api documentation - intent," https://developer.android.com/reference/android/content/Intent, 2020, accessed: 2020-05-01.

[8] Android Developer, "Logcat command-line tool," https://developer.android.com/studio/command-line/logcat, 2020, accessed: 2020-05-01.

[9] Android Developer, "UI Automator," https://developer.android.com/training/testing/ui-automator, 2020, accessed: 2020-05-01.

[10] Android developers, "Android Virtual Device (AVD)," https://developer.android.com/studio/run/managing-avds, 2020, accessed: 2020-05-22.

[11] Android Documentation, "Input events overview," https://developer.android.com/guide/topics/ui/ui-events.html, 2020, accessed: 2020-05-22.

[12] Android Documentation, "Manage touch events in a viewgroup," https://developer.android.com/training/gestures/viewgroup.html, 2020, accessed: 2020-05-22.

[13] APKMirror, "Apkmirror," https://www.apkmirror.com/, 2020, accessed: 2020-09-15.

[14] APKMonk, "Apkmonk," https://www.apkmonk.com/, 2020, accessed: 2020-09-15.

[15] APKPure, "Apkpure," https://apkpure.com/, 2020, accessed: 2020-09-15.

[16] AppBrain, "Android ad network statistics and market share," https://www.appbrain.com/stats/libraries/ad-networks, 2020, accessed: 2020-05-01.

[17] AppBrain, "Number of android applications," https://www.appbrain.com/stats, 2020, accessed: 2020-05-01.

[18] AppLovin, "Android-sdk-demo," https://github.com/AppLovin/Android-SDK-Demo, 2020, accessed: 2020-05-01.

[19] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 356–367.

[20] Cauly, "Android-sdk," https://github.com/cauly/Android-SDK, 2020, accessed: 2020-05-01.

[21] G. Chen, W. Meng, and J. Copeland, "Revisiting mobile advertising threats with MAdLife," in *Proceedings of the World Wide Web Conference*, 2019, pp. 207–217.

[22] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou, "Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios," in *IEEE Symposium on Security and Privacy*. IEEE, 2016, pp. 357–376.

[23] Chen Yu, "Sophisticated Android clickfraud apps," https://news.sophos.com/en-us/2018/12/06/android-clickfraud-fake-iphone/, 2018, accessed: 2020-09-15.

[24] G. Cho, J. Cho, Y. Song, and H. Kim, "An empirical study of click fraud in mobile advertising networks," in *Proceedings of the International Conference on Availability, Reliability and Security*, 2015, pp. 382–388.

[25] J. Crussell, R. Stevens, and H. Chen, "MAdFraud: Investigating ad fraud in android applications," in *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services*, 2014, pp. 123–134.

[26] N. Daswani and M. Stoppelman, "The anatomy of clickbot.a," in *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets*, 2007, pp. 11–11.

[27] V. Dave, S. Guha, and Y. Zhang, "Measuring and fingerprinting click-spam in ad networks," in *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2012, pp. 175–186.

[28] Dave, Vacha and Guha, Saikat and Zhang, Yin, "Viceroi: Catching click-spam in search ad networks," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2013, pp. 765–776.

[29] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2187–2200.

[30] Developer Policy Center, "Monetization and ads," https://play.google.com/about/monetization-ads/ads/, 2020, accessed: 2020-05-01.

[31] dflower, "Google play python api," https://github.com/dflower/google-play-crawler, 2014, accessed: 2020-05-01.

[32] F. Dong, H. Wang, L. Li, Y. Guo, T. F. Bissyandé, T. Liu, G. Xu, and J. Klein, "FraudDroid: Automated ad fraud detection for android apps," in *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 257–268.

[33] X. Dong, M. Tran, Z. Liang, and X. Jiang, "Adsentry: comprehensive and flexible confinement of javascript-based advertisements," in *Proceedings of the Annual Computer Security Applications Conference*, 2011, pp. 297–306.

[34] EasyList, "Easylist - filter lists are sets of rules for adblock," https://easylist.to/, 2020, accessed: 2020-05-224.

[35] Facebook, "Android facebook audience network," https://developers.facebook.com/docs/audience-network/android/, 2020, accessed: 2020-05-01.

[36] M. Feily, A. Shahrestani, and S. Ramadass, "A survey of botnet and botnet detection," in *Proceedings of the International Conference on Emerging Security Information, Systems and Technologies*, 2009, pp. 268–273.

[37] Genymotion, "Genymotion android emulator," https://www.genymotion.com/, 2020, accessed: 2020-05-01.

[38] L. Glanz, S. Amann, M. Eichberg, M. Reif, B. Hermann, J. Lerch, and M. Mezini, "Codematch: obfuscation won't conceal your repackaged app," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 2017, pp. 638–648.

[39] Google, "Android and security," http://googlemobile.blogspot.com/2012/02/android-and-security.html, 2012, accessed: 2020-05-01.

[40] Google AdMob, "Get started," https://developers.google.com/admob/android/quick-start, 2020, accessed: 2020-05-01.

[41] Google Ads, "How does google stop invalid activity?" https://www.google.com/ads/adtrafficquality/, 2020, accessed: 2020-05-01.

[42] Google Play, "Naver - google play app," https://play.google.com/store/apps/details?id=com.nhn.android.search, 2020, accessed: 2020-05-01.

[43] Google Play, "Puffin Web Browser," https://play.google.com/store/apps/details?id=com.cloudmosa.puffinFree, 2020, accessed: 2020-05-01.

[44] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the ACM conference on Security and Privacy in Wireless and Mobile Networks*, 2012, pp. 101–112.

[45] GUARDSQUARE, "Proguard," https://www.guardsquare.com/, 2020, accessed: 2020-05-04.

[46] H. Haddadi, "Fighting online click-fraud using bluff ads," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 2, pp. 21–25, 2010.

[47] S. Harries, "How much are fraudsters costing advertisers?" https://www.adjust.com/blog/ad-fraud-roundup-how-much-are-fraudsters-costing-advertisers/, 2019, accessed: 2020-05-01.

[48] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and defenses," in *Proceedings of the USENIX Security Symposium*, 2012, pp. 413–428.

[49] Insight, "Mobile share of advertising market to exceed 30% in 2020," https://www.zenithmedia.com/insights/global-intelligence-issue-06-2018/mobile-share-of-advertising-market-to-exceed-30-in-2020/, 2018, accessed: 2020-05-01.

[50] Israel Wernik, Danil Golubenko , Aviran Hazum, "Tekya Clicker Hides in 24 Children's Games and 32 Utility Apps," https://research.checkpoint.com/2020/google-play-store-played-again-tekya-clicker-hides-in-24-childrens-games-and-32-utility-apps/, 2020, accessed: 2020-09-15.

[51] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu, "Morpheus: automatically generating heuristics to detect android emulators," in *Proceedings of the Annual Computer Security Applications Conference*, 2014, pp. 216–225.

[52] A. Karge, "Mobile ad fraud in 2019: Combating the different types," https://www.smaato.com/blog/mobile-ad-fraud-in-2019-tools-and-tactics-used-by-fraudsters/, 2019, accessed: 2020-05-01.

[53] L. Kiguolis, "2spywar, fofy.com - virus removal instruction," https://www.2-spyware.com/remove-fofy-com.html, 2020, accessed: 2020-05-01.

[54] D. Kleidermache, "Tackling ads abuse in apps and sdks," https://security.googleblog.com/2018/12/tackling-ads-abuse-in-apps-and-sdks.html, 2018, accessed: 2020-05-01.

[55] Linkmine, "Linkmine api," https://docs.google.com/document/d/1nR2WZLg-G3wuO44rujhgLzVaj0m9webFgm0oq5oR-XA/, 2019, accessed: 2019-11-07.

[56] B. Liu, S. Nath, R. Govindan, and J. Liu, "Decaf: Detecting and characterizing ad fraud in mobile apps," in *NSDI*, 2014, pp. 57–70.

[57] L. Liu, Y. Gu, Q. Li, and P. Su, "Realdroid: Large-scale evasive malware detection on" real devices"," in *Proceedings of the International Conference on Computer Communication and Networks*, 2017, pp. 1–8.

[58] MG, "How to remove fofy.com from computer," https://malware-guide.com/blog/how-to-remove-fofy-com-from-computer, 2020, accessed: 2020-05-01.

[59] Microsoft News Center, "Microsoft, the fbi, europol and industry partners disrupt the notorious zeroaccess botnet," https://news.microsoft.com/2013/12/05/microsoft-the-fbi-europol-and-industry-partners-disrupt-the-notorious-zeroaccess-botnet/, 2020, accessed: 2020-05-01.

[60] MoPub, "mopub-android-sdk," https://github.com/mopub/mopub-android-sdk, 2020, accessed: 2020-05-01.

[61] Naver, "Naver portal," https://www.naver.com/, 2020, accessed: 2020-05-01.

[62] NoTracking, "Notracking - block lists," https://github.com/notracking/hosts-blocklists/tree/master/adblock, 2020, accessed: 2020-05-22.

[63] Ohad Mana, Israel Wernik, Bogdan Melnykov, Aviran Hazum, "Haken Clicker and Joker Premium Dialer," https://research.checkpoint.com/2020/android-app-fraud-haken-clicker-and-joker-premium-dialer/, 2018, accessed: 2020-09-15.

[64] P. Pearce, V. Dave, C. Grier, K. Levchenko, S. Guha, D. McCoy, V. Paxson, S. Savage, and G. M. Voelker, "Characterizing large-scale click fraud in zeroaccess," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2014, pp. 141–152.

[65] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2012, pp. 71–72.

[66] A. Pilgun, O. Gadyatskaya, Y. Zhauniarovich, S. Dashevskyi, A. Kushniarou, and S. Mauw, "Fine-grained code coverage measurement in automated black-box android testing," in *ACM Transactions on Software Engineering and Methodology*, 2020, pp. 1–35.

[67] E. PROTALINSKI, "Google has helped 300,000 android developers fix security vulnerabilities in over 1 million apps," https://venturebeat.com/2019/02/28/google-has-helped-300000-android-developers-fix-security-vulnerabilities-in-over-1-million-apps/, 2019, accessed: 2020-05-01.

[68] A. Sahami Shirazi, N. Henze, T. Dingler, K. Kunze, and A. Schmidt, "Upright or sideways? analysis of smartphone postures in the wild," in *Proceedings of the international conference on Human-computer interaction with mobile devices and services*, 2013, pp. 362–371.

[69] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, "Flexdroid: Enforcing in-app privilege separation in android." in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[70] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications," in *Presented as part of the USENIX Security Symposium*, 2012, pp. 553–567.

[71] skylot, "Jadx - dex to java decompiler," https://github.com/skylot/jadx, 2020, accessed: 2020-05-01.

[72] D. Smith, "Mastering the android touch system," https://speakerdeck.com/newcircle/dave-smith-mastering-the-android-touch-system, 2020, accessed: 2020-05-22.

[73] S. Smith, "Ad fraud to cost advertisers," https://www.juniperresearch.com/press/press-releases/ad-fraud-to-cost-advertisers-19-billion-in-2018, 2017, accessed: 2020-05-01.

[74] S. Son, D. Kim, and V. Shmatikov, "What mobile ads know about mobile users." in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[75] TnkFactory, "Tnkad sdk," http://docs.tnkad.net/tnk-ad-sdk, 2020, accessed: 2020-05-01.

[76] Unity Technologies, "unity-ads-android," https://github.com/Unity-Technologies/unity-ads-android, 2020, accessed: 2020-05-01.

[77] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the ACM symposium on Information, computer and communications security*, 2014, pp. 447–458.

[78] T. Watanabe, M. Akiyama, F. Kanei, E. Shioji, Y. Takata, B. Sun, Y. Ishi, T. Shibahara, T. Yagi, and T. Mori, "Understanding the origins of mobile app vulnerabilities: A large-scale measurement study of free and paid apps," in *IEEE/ACM International Conference on Mining Software Repositories*, 2017, pp. 14–24.

[79] K. C. Wilbur and Y. Zhu, "Click fraud," *Marketing Science*, vol. 28, no. 2, pp. 293–308, 2009.

[80] J. Wyke, "The zeroaccess botnet–mining and fraud for massive financial gain," *Sophos Technical Paper*, 2012.

[81] L. Zhang and Y. Guan, "Detecting click fraud in pay-per-click streams of online advertising networks," in *Proceedings of the International Conference on Distributed Computing Systems*, 2008, pp. 77–84.

[82] M. Zhang, W. Meng, S. Lee, B. Lee, and X. Xing, "All your clicks belong to me: Investigating click interception on the web," in *Proceedings of the USENIX Security Symposium*, 2019, pp. 941–957.

[83] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting third-party libraries in android applications with high precision and recall," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 141–152.