Dissecting Click Fraud Autonomy in the Wild

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

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

Haotian Hu
hht971026@sjtu.edu.cn
Shanghai Jiao Tong University

Xiaokuan Zhang
Zhang.5840@osu.edu
The Ohio State University

Minhui Xue
jason.xue@adelaide.edu.au
The University of Adelaide

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

ABSTRACT

Although the use of pay-per-click mechanisms stimulates the prosperity of the mobile advertisement network, fraudulent ad clicks result in huge financial losses for advertisers. Extensive studies identify click fraud according to click/traffic patterns based on dynamic analysis. However, in this study, we identify a novel click fraud, named humanoid attack, which can circumvent existing detection schemes by generating fraudulent clicks with similar patterns to normal clicks. We implement the first tool ClickScanner to detect humanoid attacks on Android apps based on static analysis and variational AutoEncoders (VAEs) with limited knowledge of fraudulent examples. We define novel features to characterize the patterns of humanoid attacks in the apps' bytecode level. ClickScanner builds a data dependency graph (DDG) based on static analysis to extract these key features and form a feature vector. We then propose a classification model only trained on benign datasets to overcome the limited knowledge of humanoid attacks.

We leverage ClickScanner to conduct the first large-scale measurement on app markets (i.e., 120,000 apps from Google Play and Huawei AppGallery) and reveal several unprecedented phenomena. First, even for the top-rated 20,000 apps, ClickScanner still identifies 157 apps as fraudulent, which shows the prevalence of humanoid attacks. Second, it is observed that the ad SDK-based attack (i.e., the fraudulent codes are in the third-party ad SDKs) is now a dominant attack approach. Third, the manner of attack is notably different across apps of various categories and popularities. Finally, we notice there are several existing variants of the humanoid attack. Additionally, our measurements demonstrate the proposed ClickScanner is accurate and time-efficient (i.e., the detection overhead is only 15.35% of those of existing schemes).

CCS CONCEPTS

• Security and privacy → Software security engineering;

---

*Haojin Zhu (zhu-hj@sjtu.edu.cn) is the corresponding author.

KEYWORDS

Click Fraud; Static Analysis; Variational AutoEncoders; Humanoid Attack

1 INTRODUCTION

The mobile advertisement (ad) market has grown rapidly over the past decades with the unprecedented popularity of smartphones. To motivate the app developer to embed the advertisers' ads in their apps, the pay-per-click (PPC) mechanism is widely deployed, in which the advertiser pays the developer according to the number of times the embedded ads have been clicked by users [17, 48].

However, the PPC mechanism also encounters the increasing threat of *click fraud* [10]. By adopting the strategy of click fraud, the unscrupulous developer generates "fake" ad click events that do not originate from real users to obtain extra payment from the ad network. For instance, an attacker can embed malicious code on fraudulent apps or third-party SDKs leveraged by other unsuspecting app developers to trigger the ad clicks automatically in the background without any human involvement. It is estimated that advertisers have lost 42 billion USD of ad budget globally in 2019 due to fraudulent activities committed via online, mobile, and in-app advertising [43].

To defend against click fraud, both academia and industry have proposed a series of *dynamic analysis* based approaches to distinguish fraudulent clicks from the legitimate clicks. These approaches fall into the following two categories: *user-side* [8, 9, 19, 20, 28, 41] and *ad network-side* approaches [11, 14, 32, 47, 49, 50]. (1) The user-side approaches rely on installing an additional patch or ad SDK on the user's device. The legitimacy of ad clicks is determined by checking whether the click pattern meets a certain rule. (2) The ad network-side schemes determine whether an app performs fraudulent clicks based on statistical information (*e.g.*, timing patterns) of the clicks through traffic analysis. These existing detection schemes either require users to install patches on their smartphones, which is not user-friendly, or require the ad network to collect traffic data from thousands of apps, which is less scalable. Moreover, both approaches use dynamic analysis, which is not complete since they do not cover all feasible program paths. Furthermore, the effectiveness of these dynamic analysis based approaches relies on the

assumption that fraudulent click patterns are distinguishable from those of real users. Therefore, it is natural to raise the following question: *Is there a smart attacker who can simulate a real human's clicks patterns and bypass existing click fraud detection?*

In this study, we answer the above question by identifying emerging automated click fraud, named `humanoid attack`. In this paper, we define `humanoid attack` as a kind of click fraud that has almost the same click and traffic patterns as normal clicks. Specifically, the fraudulent applications could randomize the click coordinates/time interval, or even follow the legitimate actions of a real user to generate the clicking traffic, rendering the fake click sequences to be indistinguishable from legitimate ones even if the ad traffic is monitored. Some fraudulent applications will also receive the fake click's configuration from a remote server and avoid detection adaptively and locally. To date, the detection of `humanoid attacks` via large-scale *static analysis* has received little attention. Therefore it is crucial to understand and mitigate `humanoid attacks`.

A large-scale static analysis of `humanoid attacks` imposes the following technical challenges. 1) How can we capture the fraudulent behavior patterns at the bytecode level by defining a set of novel features to distinguish the codes triggering false clicks from the codes generating legitimate clicks? 2) Based on the proposed features, how can we build a novel system that can automatically extract these features and accurately identify the fraudulent apps while considering very few positive samples in practice?

To address these challenges, we propose `ClickScanner`, a light-weight and effective static analysis framework to automatically detect `humanoid attacks`. *First*, our work starts from a preliminary study that aims to investigate what features can be adopted to identify `humanoid attacks`. To achieve this, we build a simple prototype based on Soot [45] to investigate the working logic behind the suspicious fraudulent apps, which likely manipulate the `MotionEvent` object to generate fake, yet indistinguishable click sequences. *Second*, through the preliminary vetting results of prototypes and careful manual checking of suspicious apps' working behaviors and bytecodes, we identify 50 apps conducting legitimate clicks and 50 apps conducting `humanoid attacks` as the seed apps for accuracy tests and feature definition.[1] Our study reveals that the `humanoid attack` mainly utilizes the combination of the following four strategies to obfuscate its fake clicks and avoid detection: *1) simulating the human clicks by randomizing the coordinates; 2) making the trigger condition of the fake clicks unpredictable by randomizing the triggering time; 3) generating the fake clicks by following the legitimate actions of real people; 4) predefining fake click's execution logic in code, receiving the click's coordinates and trigger condition from a remote server, and avoiding the detection adaptively and locally. Third*, after characterizing the working logic of `humanoid attacks`, to achieve light-weight detection, we propose a novel data dependency graph (DDG) to extract key features related to the `humanoid attack`. From the generated graph, a light-weight feature vector with 7 dimensions is obtained. *Finally*, to overcome the issue of the lack of positive examples of `humanoid`

attacks, we exploit variational AutoEncoders (VAEs) to build a robust classifier to perform one-class classification, which flags the fraudulent apps by the reconstruction error between the input and output with limited knowledge of positive examples.

We utilize `ClickScanner` to conduct the first large-scale measurement on the `humanoid attack`. The main results and contributions of our measurements are shown as follows.

- **Designing `ClickScanner` to dissect the `humanoid attack`.** We identify an novel pattern of automated click fraud, named `humanoid attack`, and design and implement the first tool to detect such an attack based on static analysis and VAEs with limited knowledge of fraudulent examples.
- **Effectiveness of `ClickScanner`.** We apply `ClickScanner` in the wild on 20,000 top-rated apps from Google Play and Huawei AppGallery to demonstrate that it can indeed scale to markets. We identify a total of 157 fraudulent apps out of the 20,000 apps with a high precision rate of 94.6%. Some of them are popular, with billions of downloads. In terms of time overhead, the average detection time of `ClickScanner` is 18.42 seconds, which is only 15.35% of the best case within four popular dynamic analysis based schemes (*i.e.*, FraudDetective [20], FraudDroid [14], MadFraud [11], DECAF [28], and AdSherlock [8]). We compare the performance of `ClickScanner` with 65 existing detection engines (*e.g.*, Kaspersky [24], McAfee [29]) from VirusTotal [46]. We show that 115 fraudulent apps out of the detected 157 fraudulent apps can bypass all employed engines, which demonstrates that our `ClickScanner` outperforms existing detection engines. We further apply `ClickScanner` on 100,000 apps randomly selected from Google Play.[2] In total, 584 apps are marked as fraudulent. We also find the difference in the behavior of `humanoid attacks` between popular and niche apps as shown in Section 5.4. Overall, the experimental results demonstrate that `ClickScanner` is effective and efficient in detecting the `humanoid attack`.
- **Novel findings are identified by `ClickScanner`.** A measurement study demonstrates the following interesting findings: 1) The `humanoid attack` distribution among app categories are notably different across different app markets (*i.e.*, Google Play and Huawei AppGallery), indicating attackers and users in different regions have different biases towards mobile ads. 2) Instead of changing the local codes of apps, the proportion of ad SDK-based attacks (*i.e.*, the fraudulent codes are in the third-party ad SDKs) has increased from 14% in June 2018 to 83% in August 2020, indicating that the SDK based attack is now dominant. 3) The ad SDK-based attacks undergo a decrease after July 2020, which is possibly due to the strict security policies of app markets as shown in Section 5.3.2. 4) More sophisticated click fraud other than coordinated or timing randomization attacks is identified by `ClickScanner`, and the details are shown in Section 6.

2 PRELIMINARIES

2.1 Mobile Advertising Ecosystem

A typical mobile advertising ecosystem consists of four components: the advertiser, user, ad network, and developer. As shown in Fig. 1, the ad network serves as the intermediary among the

---

[1] In order to add as many benign examples as possible to the dataset for training, we not only collect legitimate clicks on the view with the ad but also with other content. If there is no special emphasis in the latter part, the benign datasets will include legitimate clicks on views of other content. The fraudulent datasets are all fake clicks on the ad view.

[2] Due to the lower number of available apps, we skip this measurement on the Huawei App Gallery.
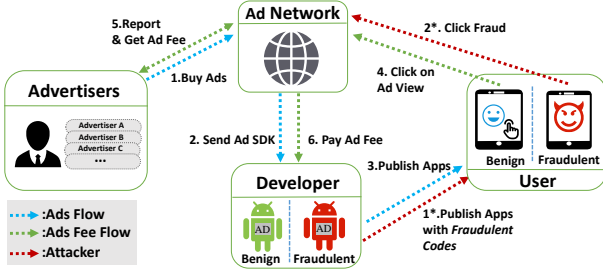
Figure 1: Overview of the mobile advertising ecosystem.

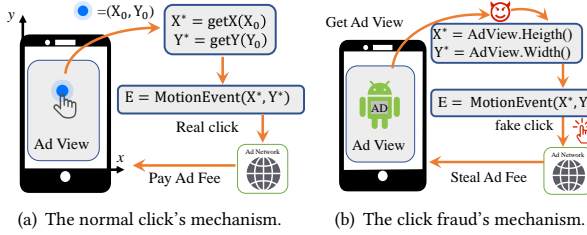(a) The normal click's mechanism.　(b) The click fraud's mechanism.

Figure 2: The click event generation mechanism in Android and how attackers use it to commit click fraud.

other components. The advertisers publish ads in the ad network that are then embedded in the apps developed by developers. Then, developers publish apps to the users and receive the advertising fee paid by advertisers through the ad network when users click on the ads. Currently, one of the most popular payment mechanism in the ad network is pay-per-click (PPC), in which the revenue received by developers is related to the number of clicks. However, these mechanisms are vulnerable to click fraud, in which attackers generate fake clicks to cheat both advertisers and users. For instance, researchers point out that around 10% to 15% of ads in Pay-Per-Click online advertising systems are not authentic traffic [12, 31, 34, 38, 39]. A report published by *Juniper Research* [40] reveals that the advertiser's loss caused by click fraud reached $42 billion in 2019.

2.2 Click Event Generation Mechanisms in Android

Since click fraud causes huge losses in the mobile ad ecosystem, it is important to figure out how click events are generated. The click mechanisms in the Android platform are shown below:

- **Normal click generation.** As shown in Fig. 2(a), when the smart phone's screen is touched by the user, the click properties, such as time, type, and coordinates, are included in a *MotionEvent* object and dispatched by the function dispatchTouchEvent to the targeted view. Then, the click information is delivered to the ad network, thereby an ad click is finished and counted by the ad network.
- **Click fraud generation.** As shown in Fig. 2(b), in a click fraud scenario, the attacker could inject malicious code snippets into apps to generate fake clicks without any user interaction. Different from normal clicks, the attacker creates a *MotionEvent*

object filled with a subset of motion values (*e.g.*, click coordinates (X^*, Y^*)) which are carefully fabricated. Since the *MotionEvent* object can be constructed arbitrarily by the attackers, from the view of the ad network, the fake click has the same format as a normal one.

2.3 Existing Click Fraud Detection Schemes

Extensive click fraud detection schemes could be divided into two categories and their insights and limitations are shown below:

- **User-side detection.** These schemes install an additional patch or SDK on users' devices to check the click pattern generated on users' devices. One of the most recent works is AdSherlock [8] which is based on the insight that: 1) "bots-driven fraudulent clicks" can be detected because the properties are inconsistent between human clicks while remaining the same for bots-driven clicks, and 2) the "in-app fraudulent clicks" can be detected because the in-app clicks do not generate any motion events. However, there are many click fraud apps that can generate motion events that simulate the properties of a human's click through the MotionEvent.obtain() method [4], and AdSherlock failed to consider this kind of click fraud. Another recent work is Fraud-Detective [20], which generates the causal relationships between user inputs and observed fraudulent activity. However, FraudDetective requires a large time overhead and cannot cover all of the app's functionalities, which makes it difficult for FraudDetective to trigger and identify the humanoid attack discovered in this paper.
- **Ad network-side detection.** These schemes analyze the ad requests at the ad network server. The most recent work is Clicktok [32], which argues that unusual click-stream traffic is often simple reuse of legitimate data traffic. Thus, they try to detect click fraud by recognizing patterns that repeat themselves in the same click-stream of ads. However, to date, a large amount of click fraud does not rely on legitimate click data streams, and attackers can also carefully construct data streams similar to the pattern of legitimate click data streams to fool detectors as shown in Section 3.

Furthermore, most of the above schemes are based on dynamic analysis or traffic analysis, and therefore incur limitations. These tools cannot cover all feasible program paths, and are thus not effective and impractical to deploy in the app market. These schemes also rely on the hypothesis that the patterns generated by click fraud and real clicks are distinctly different, which may not hold true when facing humanoid attacks.

3 MOTIVATING EXAMPLE AND INSIGHT

3.1 Preliminary Study on Humanoid Attack

While the community struggles to properly address traditional click fraud based on dynamic and traffic analysis, deception techniques used by attackers continue to evolve. To characterize humanoid attacks, we conduct a preliminary study to collect several fraudulent apps towards further building ClickScanner. We took a straw-man strategy that the first humanoid attack event was spotted and discussed on a security panel inside a company. As researchers in collaboration with the company, we tried to explore

more events by building up ClickScanner-Beta characterizing the app's activity to scale up the detection in the wild. Note that, when a touch event occurs, the `dispatchTouchEvent` [2] delivers the event from an Android Activity down to the target view. Therefore, we build ClickScanner-Beta based on Soot [45] to monitor the `MotionEvent.obtain` invocation, which generates and delivers the *MotionEvent* object — an object used to report movement events [3] to the `dispatchTouchEvent`. We optimized ClickScanner-Beta iteratively by first filtering out the seed apps through manual verification. When a feature is established, we update ClickScanner-Beta to reduce the overhead of manual verification. *We highlight that the difference between the* `humanoid attack` *and random clicks is four-fold: The* **basic attack** *randomizes the click properties (and sometimes these properties may follow a certain distribution such as Gaussian distribution) in local codes, such as: 1) simulating the human clicks by randomizing the coordinates; 2) making the trigger condition of the fake clicks unpredictable by randomizing the triggering time. The* **advanced attack** *receives click properties from the cloud-server (see Section 6.2) or generates click properties by imposing random disturbances on user actions, such as 3) generating the fake clicks by following the legitimate actions of real people; 4) predefining fake click's execution logic in code, receiving the click's coordinates and trigger condition from a remote server, and avoiding the detection adaptively and locally.* Through the preliminary vetting of our prototype and careful manual verification of the apps' working behaviors and decompiled codes, we identify 50 apps conducting legitimate clicks and 50 apps conducting `humanoid attacks` as the SEED APPS. We elaborate on one of the most representative fraudulent apps in this section as a motivating example.

3.2 Analysis of Our Motivating Example

We use Monkey [13] to randomly click on the motivating example and an app with no click fraud. We then show the coordinate distribution and timing pattern of the click event on ad banners on both apps to illustrate some of the key challenges addressed by our work. To illustrate the advantages of our work, we also reproduce traditional click frauds including *fixed clicking* and *replay clicking* in [8, 32] and compare their coordinate distributions and timing patterns with `humanoid attacks` we found. The click event records are shown in Fig. 3. Note that for simplicity, we filter out the coordinate distributions and timing patterns generated by each kind of attack when presenting the results.

The *fixed clicking* belongs to the traditional click fraud where the coordinates of the generated touch events (shown in Fig. 3(a) and 3(b)) are the same, which is easy to detect through traditional rule-based or threshold-based approaches. The *replay clicking* is another traditional click fraud which replays organic clickstreams on ad banners. This can be detected by [32] because their timing patterns are similar to traditional timing patterns as shown in Fig. 3(c). However, we discovered that the `humanoid attack` is more sophisticated and cannot be easily detected using the above approaches, since attackers simulate human clicks to camouflage their false clicks. The coordinate distribution and timing pattern of the `humanoid attack` is generated as if the user clicks. For instance, the distribution of coordinates in *X* axis of `humanoid attack` resembles normal clicking, whereas the ad traffic generated

(a) Visual distribution of the click points.

(b) Cumulative distribution of the click points along the X axes.

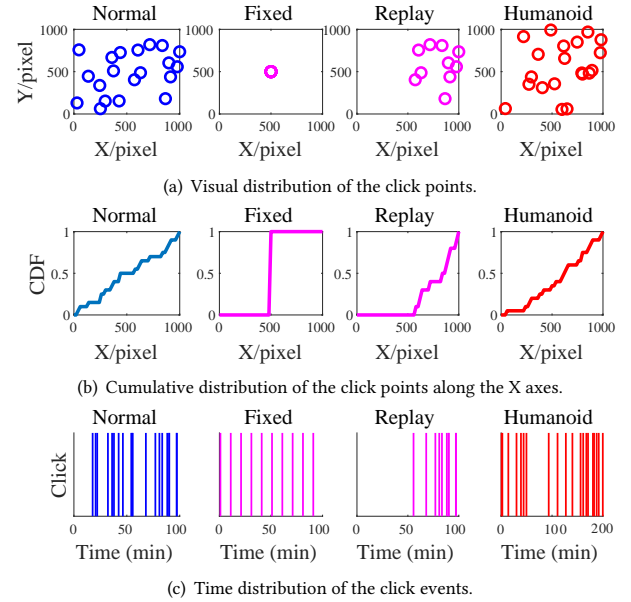(c) Time distribution of the click events.

Figure 3: Illustration of ad clicks between normal, traditional (fixed and replay) fraud, and humanoid attack.

by the fraudulent app is nearly 0.5 times more than that of the normal app, which can be easily passed off as traffic from legitimate users that are interested in the ad.

To reveal how the attacker achieves this, we next analyze the decompiled codes of the above `humanoid attack` app. As shown in Fig. 4, the code snippet example simplified from the motivating example exhibits the click fraud following legitimate human actions. In general, the ad view in the fraudulent app is randomly clicked again in a random time period after the real person clicks on the ad. The fake clicks will never be triggered if the real user does not click the ad. To do this, attackers insert the function `dispatchTouchEvent` at lines 5 and 6 which generates the fake clicks in the body of the function `onClick()`. At lines 1 and 2 in Fig. 4, the attacker also tries to fool detectors by making the trigger condition of the click event unpredictable and by randomizing the coordinate and trigger time of the fake click, impersonating a human's click pattern.

This makes it hard for traditional ad network-side fraud detection approaches to detect it because the click in `humanoid attack` is very likely to be triggered by a real person due to the uncertainty of click patterns. Additionally, user-side approaches are ineffective in

```
public void onClick(View AdView) {
1.    float rand = getRandomNumber();
2.    long v1 = SystemClock.uptimeMillis() + (long)rand;
3.    long v2 = v1 - 500 - (((long)new Random().nextInt(500)));
4.    if(rand > 0f && rand > new Random().nextFloat()) {
5.        float x = ((float)(AdView.getWidth()* (new Random().nextFloat())));
6.        float y = ((float)(AdView.getHeight()* (new Random().nextFloat())));
7.        (ViewGroup)AdView.getChildAt(0).dispatchTouchEvent(MotionEvent.obtain(v2, v1, 0, x, y, 0));
8.        (ViewGroup)AdView.getChildAt(0).dispatchTouchEvent(MotionEvent.obtain(v2, v1, 1, x, y, 0));
    }
}
```

Figure 4: The code snippet simplified from the fraudulent app in the motivating example.
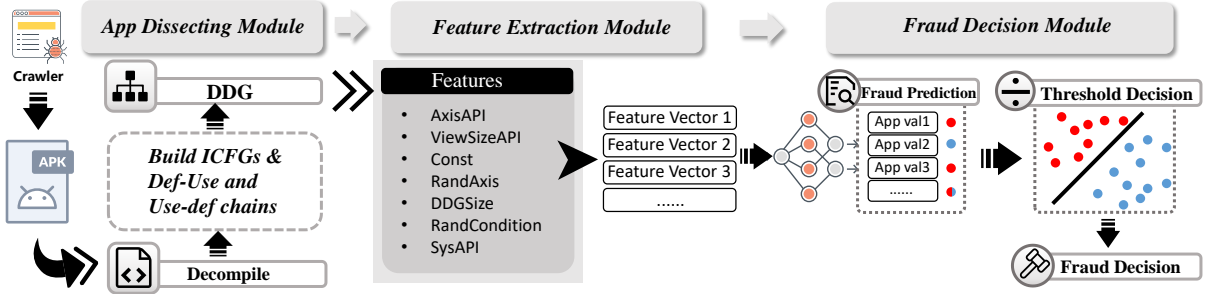
Figure 5: The workflow of `ClickScanner`.

detecting humanoid attacks, because the click patterns and click effects on the user-side generated by humanoid attack are almost the same as those of real clicks since the attackers are assumed to be allowed to arbitrarily construct the *MotionEvent* object instead of just using the same *MotionEvent* [8]. Further, it is also a challenge for detectors to trigger humanoid attacks due to its random trigger condition. Therefore, there is a pressing need to address the fake clicks that stem from the humanoid attacks. To this end, this paper presents `ClickScanner`, a scalable, efficient, and automated static analysis system to identify the humanoid attack.

3.3 Insight of `ClickScanner`

Section 3.2 demonstrates that the humanoid attack can manipulate ad clicks with similar pattern to that of a normal click scenario, thus causing existing detection schemes based on dynamic analysis futile. In this study, to successfully detect the humanoid attack, the key insight is that although the click pattern is camouflaged as legitimate, at the bytecode code level the difference of the ad click trigger condition and generation process between legitimate and fraudulent apps are notably significant, which can be characterized as detection features.

As illustrated in Fig. 4, it is observed that when generating a fraudulent click, the app must exploit methods `getHeight` and `getWidth` to obtain the height and width of the ad view. Furthermore, the click trigger condition defined by the method `Random` is utilized to disguise fraudulent clicks. By checking the *parameters* and *trigger conditions* within the bytecode of `MotionEvent`, it is feasible to detect this humanoid attack case. Therefore, we propose a static analysis based detection scheme `ClickScanner`, and break down `ClickScanner` in Section 4.

4 SYSTEM DESIGN OF CLICKSCANNER

As shown in Fig. 5, `ClickScanner` mainly consists of three components (*i.e.*, App Dissecting Module, Feature Extraction Module, Fraud Decision Module) to automatically detect humanoid attacks. In this section, we break down `ClickScanner` into each component.

4.1 App Dissecting Module

For a given APK, `ClickScanner` first determines whether it is associated with mobile ads, and then converts the properties of the click event targeted at an ad view to a data dependency graph (DDG) for further feature extractions.

4.1.1 Preprocessing of App Dissecting. When detecting humanoid attacks in apps from the app market, it is crucial for `ClickScanner` to only focus on apps involving mobile ads. To achieve this goal, `ClickScanner` has the following three steps. First, `ClickScanner` checks apps' permissions and filters out those with no permissions such as INTERNET and ACCESS_NETWORK_STATE [26]. Second, for the remaining apps, `ClickScanner` leverages LibRadar [30], a popular and obfuscation-resilient tool to detect third-party libraries on those apps and discards the apps without ad libraries. Third, `ClickScanner` needs to remove views that do not contain ad contents to avoid unnecessary analysis. Since there are no explicit labels that would allow us to easily distinguish ad views from other views, in this study, `ClickScanner` uses the relevant ad features, such as string, type, and placement features, to determine ad views, followed by prior research [14]. In summary, only the apps that successfully pass the above three analysis steps would undergo the static analysis of `ClickScanner`.

4.1.2 Extracting Click Event Properties through Static Analysis. After `ClickScanner` selects those click events targeted at ad views, `ClickScanner` performs static analysis on them and extracts their properties and trigger conditions. As mentioned in Section 2.2, attackers typically use the `MotionEvent.obtain` function to create a new *MotionEvent* object by obtaining the properties of a click event as its parameters, and then attackers deliver it to the `dispatchTouchEvent` function to perform the humanoid attack. Therefore, for a given app (APK), `ClickScanner` first utilizes the static analysis tools Soot [45] and Flowdroid [6] to build inter-procedural control flow graphs (ICFGs), Def-Use (DU), and Use-Def (UD) chains of it. However, separately deploying the above ICFG, UD, and DU chains cannot represent the parameters assignment process of the `MotionEvent.obtain` function and the trigger condition formation process of the `dispatchTouchEvent`. Therefore, to overcome these issues, we propose a novel data dependency graph (DDG) to show the overall *properties* and *trigger conditions* of the click event for further feature extraction, and the details of DDG building are introduced as follows.

The initialization of DDG. We propose a novel data dependency graph (DDG) to show the overall properties and trigger conditions of the click event based on ICFGs, DU, and UD chains for further feature extraction. DDG can include all the data that make up properties and trigger conditions of the click event in a graph, where each node represents the statement, and each edge represents the

Algorithm 1 DDGTool

Input: ICFG; UD chain; DU chain; root;
Output: DDG;
1: DDG = emptyset
2: DDG.setRoot(*root*)
3: **while** DDG is changing **do**
4:    **for** every i in DDG **do**
5:       **if** i is const **then**
6:          const_def = getDefSite(*i, UDChain*)
7:          DDG += New DDGNode(*const_val*)
8:       **else if** i is func **then**
9:          **if** !isSysAPI(*i*) **then**
10:             func_def = getDefSite(*ICFGs, i*)
11:             DDG += getSubGraph(*func_def*)
12:          **else**
13:             DDG += New DDGNode(*i*)
14:          **end if**
15:       **else if** i is var **then**
16:          DDG += New DDGNode(*i*)
17:          DDG += getPre(*ICFG, UD&DUChain, i*)
18:       **else**
19:          para_caller = getCaller(*ICFGs, i*)
20:          DDG += getPara(*para_caller*)
21:       **end if**
22:    **end for**
23: **end while**
24: **return** DDG

---

dependency relation between the two statements. We can find out what data have been used to form the properties and trigger conditions of the click event and what the relationship is between them by the DDG. After obtaining the ICFG, DU chains, and UD chains, we develop backward program slicing in Algorithm 1 to build the DDG. Fig. 6 shows a DDG generated by `ClickScanner` corresponding to Fig. 4. The red arrows are the routes of backward program slicing and nodes are the statements. The inputs of the algorithm are the ICFG, DU and UD chains, and the *root* which are those items in the condition expression of `dispatchTouchEvent` and the parameters of `MotionEvent.obtain` as mentioned in Section 3. In particular, the roots of this DDG in Fig. 6 are "x coordinate" node and "y coordinate" node. The algorithm starts with the empty set *data dependency graph (DDG)* and aims at finding the assignment process of those items and parameters. It is observed that both the values of the items in the condition expressions and the parameters of `MotionEvent.obtain` representing the properties of a click event are usually composed of four types of data. They are *constants*, *variables*, *return value* of a method and *parameters* of the function which calls the `dispatchTouchEvent`. One or more of these four types of data are combined through arithmetic operations to form the final result.

The expansion of DDG. For each item in the DDG generation, as shown at line 3 to line 22 in Algorithm 1, `ClickScanner` handles the above four types of data by repeating the following steps. 1) If it is a constant, `ClickScanner` will find the definition site of it by UD chain and directly add it to the DDG. 2) If it is a return value of a certain system method that usually has a fixed meaning,
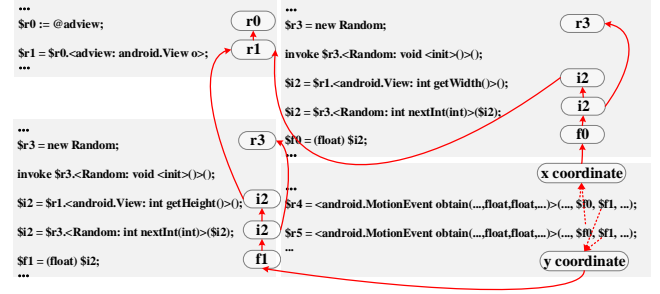

Figure 6: A simplified illustrative figure of the data dependence graph (DDG) of the motivating example constructed by `ClickScanner` for the code in Fig. 4. The red arrows are the routes of backward program slicing.

`ClickScanner` directly adds the return values of it to the DDG. However, if it is a return value of the developer-defined method without fixed meaning, `ClickScanner` finds the method's definition site and identifies what processing has been performed in its method body. Then, `ClickScanner` converts all the nodes in their method bodies into subgraphs of the DDG. 3) If it is a variable that is usually formed by different types of data, to figure out the variable's meaning, `ClickScanner` needs to find the variable's assignment process. Therefore, `ClickScanner` finds its predecessors from the definition site of the variable based on the ICFG and UD chains and adds it to the DDG. 4) If it is a parameter of the method in which the `dispatchTouchEvent` is called, `ClickScanner` first gets the caller of the method based on ICFG and then finds the parameter values of the method.

As shown in Fig. 6, `ClickScanner` successfully finds exactly where the `humanoid attack` occurs in codes. The process of generating the DDG of the trigger condition is similar to the above description, so we omit it due to space limitations.

4.2 Feature Extraction Module

Once a DDG is built, `ClickScanner` can find all necessary features in it for verifying whether the `humanoid attack` takes place in the given app. Android systems typically have 7 different constructors for the `MotionEvent.obtain` and all of them have the following common parameters: *eventTime, actionType, axisValue and metaState*, among which we choose the axis values (*i.e.*, touch position (AXIS_X, AXIS_Y)) for further study. Because the benign app receives the `MotionEvent` object from the system, sometimes it needs to record the coordinates of the click, and then dispatches it. However, for fraudulent apps (*i.e.*, Fig. 4 at lines 5 and 6), they first obtain the height and width of the ad view and construct the fake click's coordinates that follow a random distribution, which mimics the benign app behaviors. As a result, even if the traditional click fraud detection approaches can obtain click traffic, they cannot distinguish between a `humanoid attack` and a normal click since properties such as the coordinates are similar.

This shows that instead of analyzing the pattern of generated coordinates, the `humanoid attack` can be identified by checking the process of coordinate generating. For instance, the "illegal operations" including obtaining height/width and exploiting Random()

method in Fig. 4 may be used to detect its fraudulent behaviors. In a real-world scenario, we further characterize the axis into five features:

(1) The number of APIs for getting the actual click coordinates generated by users (AxisAPI). As shown in Fig. 4, the fraudulent app involves no APIs to get the coordinates of real users' click (*e.g.*, getX() and getY()). Instead, it constructs the coordinates by itself. Intuitively, the existence of APIs, which are used to get the actual click coordinates generated by users, can be indicative of whether the app is a fraudulent application. When an app contains a *MotionEvent* whose coordinate parameters do not involve the system APIs above, we take it as a potentially fraudulent application. As shown in Table 1, the F-score of AxisAPI is 0.81 when identifying `humanoid attack` instances over our ground truth dataset (SEED APPS).

(2) The number of APIs for getting the size of ad view (ViewSizeAPI). Many fraudulent apps obtain the size of the ad view in order to place the coordinates of the fake click inside the ad view. Although some benign apps will also get the view size, the proportion is much lower than that in fraudulent apps based on the observation of benign samples in our dataset.

(3) The number of the constants (Const). Since some fraudulent apps try to click on the area around a fixed point in the ad view, such as the download and install button, they will obtain the size of a view and calculate it with a constant to get a specific point coordinate. The F-score for this feature is shown in Table 1.

(4) The number of API for getting random numbers (RandAxis). To better mimic human clicks, the fraudulent apps often use APIs that generate random numbers (*e.g.*, `random.nextGaussian`) when constructing click coordinates or the time distribution of clicks, as shown in the motivating example. In doing so, attackers can disguise the traffic generated by fake clicks as traffic generated by real people and make the fake clicks unpredictable to evade the dynamic analysis. Therefore, we use this as an indicator to identify fraudulent apps. The F-score for this feature, as measured on our ground-truth set, is illustrated in Table 1.

(5) Size of the DDG (DDGSize). Our manually labeled dataset shows that fraudulent apps tend to process the data several times before passing it to the `MotionEvent.obtain` as its coordinate parameters, while benign apps tend to directly take the return value of the methods like `getX` as the coordinate parameters. The larger size of the DDG indicates that the data have been processed more times before being passed to the `MotionEvent.obtain`. The F-score for this feature is shown in Table 1.

Meanwhile, the attackers also tend to change their behaviors to evade detection, which can be detected by analyzing the trigger conditions of the click events. The unique software and hardware resources on mobile devices enable fraudulent apps to cover their behaviors with a wider spectrum of triggers, that is, conditions under which the hidden operations will be performed [33]. For example, in Fig. 4 at line 4, the fraudulent app tries to fool detectors by randomizing the trigger condition of the click event, which is like a human's click timing pattern and difficult to be triggered by dynamic analysis. Therefore, we also focus on the trigger condition of click events and characterize it into two features:

Table 1: F-score of features

AxisAPI	ViewSizeAPI	Const	RandAxis
0.81	0.79	0.83	0.62

DDGSize	RandCondition	SysAPI
0.82	0.54	0.61

[1] F-score is calculated based on classification with each single feature.

(6) Random Numbers in Condition Expression (RandCondition). Many fraudulent apps tend to randomize the trigger conditions and trigger frequency of `humanoid attacks` to simulate legitimate clicks, which makes the fake clicks indistinguishable and undetectable. Additionally, dynamic analysis requires much time to interact with these apps so it is difficult to cover all paths of the `humanoid attack`. Hence, we regard the invocations of functions in the process formation of the trigger conditions, which can generate random numbers as a feature to identify the `humanoid attack`.

(7) System Call in Condition Expression (SysAPI). Some hidden sensitive instances with a similar purpose to the `humanoid attack` have been discussed [15, 33]. They are subject to some system properties or environment parameters (*i.e.*, OS or hardware traces of a mobile device). They can only be exposed to an app through system interfaces. Hence, we can infer that the condition of the `humanoid attack` is also expected to involve, directly or indirectly, one or more API calls for interacting with the OS, and we regard them as another feature. The F-score for this feature is 0.61 as illustrated in Table 1.

Although all the above features can contribute to the detection of `humanoid attacks` to a certain degree, certain kinds of `humanoid attacks` may involve several features and a single feature may cause high false positives and negatives. Therefore, none of those features can work alone. [51] Hence, our key idea is to use some of these features collectively. We finally combine all 7 features into the same feature space according to our experiment result, which is illustrated in Section 5.1. Furthermore, we apply normalization to the features before feature vectors formalization because the components of the features are different. To determine the weight of each feature, the entropy weight method is deployed by `ClickScanner`.

4.3 Fraud Decision Module

Existing click fraud detection models either need to specify many rules for classification [8, 14, 28, 32], which leads to high false negatives due to the incomplete and statistically unrepresentative rules, or require a large number of malicious samples as the training set [11], which is unrealistic due to the lack of labeled datasets. Moreover, since these existing approaches rely heavily on the knowledge of certain rules and training set labels, they may fail to handle subsequent variant click fraud. To overcome these limitations, we build an effective classifier based on Variational AutoEncoders (VAEs) with limited knowledge about fraudulent examples. This can reduce the researchers' dependence on fraudulent data sets and is more robust to variants of such newly discovered attacks.

In a nutshell, a VAE is an autoencoder whose encoding distribution is regularized during training in order to ensure that its latent space has good properties so that it can be used to generate

new data that is similar to the inputs. We use benign examples to train our classifier and determine whether an input is benign or not according to the reconstruction error between the input and output. Specifically, the encoder is a neural network. Its input is x, which is the feature vectors generated by the `ClickScanner`. The encoder's output is a hidden representation z, which is the aforementioned latent space. The encoder will perform dimensionality reduction on the input x because the encoder must learn an efficient compression of the data into this lower-dimensional space. The decoder is another neural network. Its input is the representation z, and its outputs are the parameters to the probability distribution of the data with weights and biases ϕ. Some information may be lost due to the dimensionality reduction of the encoder, and some new data are generated due to the random sampling of the decoder. We can use the reconstruction error to measure the difference between the input and output.

In the training phase of our classifier, it is trained with benign examples' feature vectors in advance so that its encoder will be able to learn the representations of benign examples. To do this, we randomly selected 10,000 benign apps from [1] for training. We train the VAE with feature vectors of those APKs that are not marked by all the engines from VirusTotal [46], a website that aggregates many antivirus products and online scan engines to check for viruses. Although, as mentioned in [52], the detection results of VirusTotal are not always reliable, because we use many benign samples for training, a relatively small number of fraudulent samples that are not detected by VirusTotal will not affect the distribution of the latent space. Once the classifier has been well trained, its encoder will learn benign examples' representations in the latent space. After training, we feed a tested app's feature vector, which is extracted from the newly formed DDG in the `Extractor`, to the VAE and output the reconstructed feature vector containing the information of the latent space in the training phase. Our classifier will consider its input to be fraudulent only when its reconstruction error exceeds a certain threshold t. It is similar to building a borderline that encompasses all benign examples so that we only need to check whether an input is in the borderline by computing the reconstruction error.

To the best of our knowledge, our model is the first Android `humanoid attack` detector with limited knowledge about fraudulent examples. Due to the lack of malicious examples in reality, this makes `ClickScanner` practical to deploy.

5 MEASUREMENTS

As mentioned in Section 3.1, due to the absence of existing benchmarks in this research area, we manually label 100 apps containing 50 fraudulent examples and 50 benign examples as our SEED APPS for fine-tuning and accuracy tests. Then we utilize `ClickScanner` to conduct the first large-scale measurement of the `humanoid attack` in the current app market based on 120,000 apps (10,000 top-rated apps from Google Play, 10,000 top-rated apps from Huawei App-Gallery, and 100,000 randomly selected apps from Google Play), and elaborate on several important findings. All experiments are performed on a Windows 10 Desktop, equipped with 8 CPU Cores at 3.6GHz and 32 GB of RAM.

Table 2: Performance of the classifier

	Precision	Recall	F-score
SEED APPS	48/50 = 96%	48/50=96%	0.960

5.1 Evaluation of ClickScanner

5.1.1 Fine-tuning of `ClickScanner`. Before utilizing ClickScanner to conduct large-scale analysis, it is necessary to fine-tune the `ClickScanner`'s parameters on SEED APPS to achieve the best performance. As mentioned in Section 4.2, there are seven different features for `ClickScanner`. To determine the best feature combinations, we traverse all combinations from 2 to 7 features and show their best performances with the ROC (Receiver Operating Characteristic) curves in Fig. 7. It is observed that the performance is improved by adding new features. When there are 5 features selected (*AxisAPI*, *ViewSizeAPI*, *RandAxis*, *DDGSize*, *RandCondition*), adding more features only leads to a slight improvement in accuracy, which demonstrates that our feature vector can adequately describe the app behaviors and help classifiers to identify fraudulent apps. We also evaluate `ClickScanner` under different thresholds t for threshold selection. It is observed when t is set to 2.04, `ClickScanner` achieves the best performance. In the following measurement study, to achieve the best accuracy, t is set to 2.04 and all 7 features are selected by `ClickScanner`.

5.1.2 Effectiveness of `ClickScanner`. Table 2 shows when choosing the above parameters, 48 apps out of 50 fraud apps in the SEED APPS are successfully recognized by `ClickScanner`. `ClickScanner` achieves the F-score of 0.960, showing its effectiveness in detecting `humanoid attacks`. There are 2 false positive and 2 false negative cases and we discuss the root causes in Section 7. Note that since SEED APPS are independently extracted from manual check (see Section 3.1), `ClickScanner` achieves high average values of precision and F1-score with limited knowledge about malicious examples, which implies its effectiveness. The effectiveness is also substantiated with the high precision of detection in the wild (see Section 5.2.2).

5.1.3 Comparison with the State-of-the-Art Ad Fraud Detection Tool. Currently, the most up-to-date fraud detection tool is FraudDetective implemented by Kim et al. [20], which computes a full stack trace from an observed ad fraud activity to a user event by connecting fragmented multiple stack traces. It is an effective tool which could detect three types of ad fraud. However, like other tools that use dynamic analysis, it incurs a large time overhead to execute apps and interact with them. Additionally, since some fraudulent apps will randomly trigger the `humanoid attack`, FraudDetective may not be able to cover all the program paths, and thus it is difficult for FraudDetective to trigger all `humanoid attacks` discovered in our study.

Since FraudDetective and their datasets are not publicly available, we make our best effort to craft the datasets of ClickScanner as similar as possible to the ones used by FraudDetective, and we acknowledge that data duplication may exist between ClickScanner and FraudDetective. The two datasets were obtained around the same time with a similar data collection methodology. In our BASIC DATASET, we collected 10,000 top-rated apps in total from
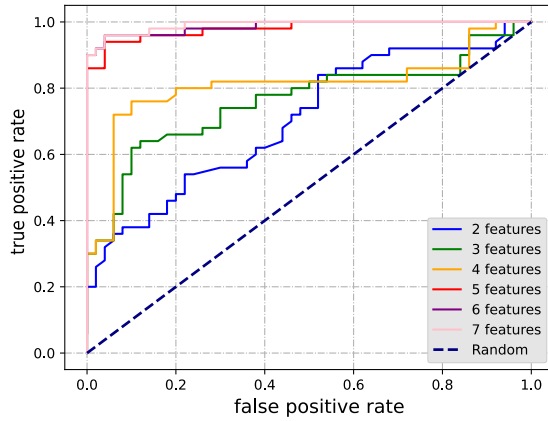
Figure 7: The ROC curve for different combination of features.

Figure 8: The performance of the VirusTotal in vetting fraudulent apps identified by `ClickScanner`.

each category from Google Play updated in July 2020, and we also conducted a longitudinal study, collecting fraudulent apps with different versions published from August 2017 to December 2020. The researchers of FraudDetective collected the top 10,024 apps from each of the Google Play categories from April 2019 to September 2020 and randomly sampled additional 38,148 apps from APK mirror sites. We believe that these two datasets are likely to have overlapped apps. However, FraudDetective may fail to trigger humanoid attacks discovered in this paper since it is based on dynamic analysis alone. In their evaluation part, FraudDetective did not detect any app that generates a forged click among the 48,172 apps crawled from the Google Play Store. By contrast, `ClickScanner` successfully identified 157 fraudulent apps among 20,000 apps in the BASIC DATASET as shown in Section 5.2.1. In summary, `ClickScanner` outperforms existing detection tools in the aspect of detecting humanoid attacks, and we leave the comparison on the same large-scale dataset for future work.

5.2 Detecting Humanoid Attacks in App Markets

We first build a BASIC DATASET with 20,000 top-rated apps and show the detection results of `ClickScanner` below. We show more details of the fraudulent apps we have detected in Appendix A.

5.2.1 The Collection of the BASIC DATASET. The BASIC DATASET contains 20,000 apps, in which every 10,000 app is top-rated and sorted by downloading numbers in each category from Google Play and Huawei AppGallery updated in July 2020. We choose these apps because, in app markets, the few most popular apps contribute to a high majority of app downloads. Therefore, by analyzing the BASIC DATASET, we can focus on humanoid attack cases with the highest influence [36]. Additionally, Google Play and Huawei AppGallery are the biggest app markets in the U.S. and China respectively, which ensure that our study is representative in scope. *In this study, if not otherwise specified, all analysis is done using the BASIC DATASET.*

5.2.2 The Humanoid Attack Cases Detected by `ClickScanner`. After conducting analysis of 20,000 apps, `ClickScanner` identifies 170 suspicious click activities from 166 suspicious apps. Then, by
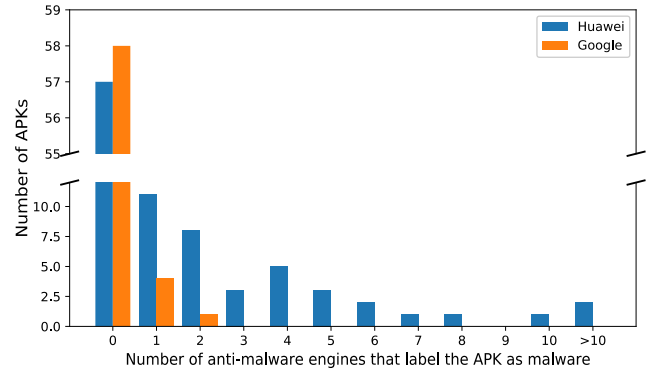
manually checking those apps' decompiled codes, 74 humanoid attack activities in 63 apps and 140 humanoid attack activities in 94 apps from Google Play and Huawei AppGallery are found respectively. The precision rate of 94.6% over 20,000 apps demonstrates the effectiveness of our classifier and fine-tuned parameters. As a consequence of over 1.2 billion downloads of such fraudulent apps in the market, the humanoid attack is very likely to have deceived both advertisers and users with fake ad clicks which corresponds advertisers huge losses. We attempted to perceive the status quo (until March 29, 2021) of 63 fraudulent apps and 94 fraudulent apps in the BASIC DATASET we detected from Google Play and Huawei, respectively. We found that 13 of 63 in Google and 26 of 94 in Huawei have been removed, but the remaining apps are still publicly available. We are also in the process of liaising with relevant app vendors for responsible disclosure. And Google notified us that they have received our report.

To quantify the damage of humanoid attacks in the real world, it is important to know the category distributions of identified malicious apps. Thus, for each fraudulent app, we extract its category (*e.g.*, books, education, weather) labeled by app markets. And analyze the distribution of Apps affected by humanoid attacks and show them in the Appendix B due to the page limitation.

5.2.3 Comparison with Existing Detection Engines. To compare the performance between `ClickScanner` and existing click fraud detection schemes, we use VirusTotal [46], a detection platform that integrates 60 anti-malware engines including Kaspersky [24] and McAfee [29], to double check the ad fraud apps identified by `ClickScanner`. Note that, as mentioned in [35, 52], the performance of VirusTotal is not stable due to its design flaw. Therefore, to eliminate the detection error of VirusTotal, we uploaded those apps in July 2020 and January 2021 respectively to ensure the accuracy of the detection results, and the results are the same for both app sets. Although VirusTotal is not tailored for identifying humanoid attacks, we use it as a baseline since it provides basic functionality to spot some kinds of ad frauds as "adware". The results of VirusTotal and `ClickScanner` are shown in Fig. 8. It is observed that 58 and 57 apps in Google Play and Huawei AppGallery successfully bypass all detection engines of VirusTotal, and only 5 apps can be detected by more than 7 engines. These results demonstrate that
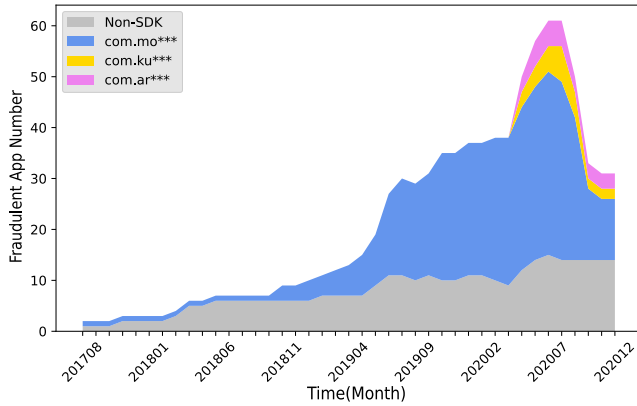
Figure 9: The stacked line chart of the number of fraudulent apps categorized by fraudulent SDKs from Google Play.

our `ClickScanner` outperforms existing detection engines in terms of click fraud detection.

5.3 Humanoid Attacks in SDKs

By conducting a longitudinal study on the aforementioned detected apps on Google Play.[3] We note that fraudulent SDK injection has played an increasingly important role in `humanoid attacks`. We will elaborate upon the analysis below.

5.3.1 Fraudulent SDKs in Humanoid Attacks. For the 157 apps detected by `ClickScanner` in BASIC DATASET, we manually analyze the reasons leading to `humanoid attacks`. It is observed that the `humanoid attack` can be divided into non-SDK-based cases and SDK-based cases. In the former, the attackers directly inject the fraudulent click codes into the apps' local codes. However, in the latter case, the fake clicks are generated by the deployed third-party ad SDKs. As shown in Table 3, 67% and 95.2% fraudulent apps from Huawei AppGallery and Google Play belong to SDK-based cases, meaning that SDK based attack is the dominant manner of `humanoid attacks` in our dataset.

For the 63 fraudulent apps in Google Play, we collect all their versions published from August 2017 to December 2020 and determine whether the attack is caused by the SDK. As shown in Fig. 9, it is observed that the proportion of ad SDK-based attack (*i.e.*, injecting fraudulent codes into the third-party ad SDKs) has increased from 14% in June 2018 to 83% in August 2020, which means the SDK based attack is now a dominant attack approach. Besides, the most popular fraudulent SDKs in Google Play are com.mo***, com.ku***, and com.ar***. We observe the open source code on the Github, analyze the advertising SDK downloaded from official channels and confirm that all 7 SDKs labeled by ClickScanner were created fraudulently by SDK developers before publishing. The remaining 4 SDKs are not publicly available.

5.3.2 New Findings on SDK com.mo***. Another interesting finding from Fig. 9 is that the attacks caused by SDK com.mo*** suffer an

unusual decrease after August 2020.[4] We analyze the apps deploying this SDK and find that 28 of the 39 apps have removed the SDK after August 2020. Then, by searching the commit record of it on GitHub, we also find that the `humanoid attack` code was removed after the version published in November 2020. Inspired by a report from Forbes News in August 2020 revealing a malicious ad SDK named Mintegral [22], which was used by 1,200+ apps and was caught for performing click frauds since July 2019, we conjecture that the app developers may have noticed this issue and subsequently removed the ad SDK from their apps, or Google enforced a new policy against this SDK to disallow apps using it to be published in the Google Play Store.

We also analyze the distribution of fraudulent SDKs of all 157 apps according to their categories. We show the detailed results in Appendix C due to the page limitation, which shows that the use of SDK to commit `humanoid attack` is not an isolated case against a certain category of apps, but a common method deserving attention.

5.4 Analysis on the EXTENSIVE DATASET

Extensive dataset. The aforementioned analysis is all based on the BASIC DATASET. To present a comprehensive overview of `humanoid attacks`, we continue to randomly collect 100,000 apps from the dataset published in [1], downloaded from Google Play updated on January 1, 2021, which are not in the BASIC DATASET.[5]

Measurement findings. From 100,000 randomly selected apps, `ClickScanner` identifies 584 apps as fraudulent. Given that there is no ground truth for the identification of click fraud in our BASIC DATASET, each app must be inspected to confirm the existence of click fraud. However, since inspecting more than 500 apps is time-consuming, we instead choose to randomly sample 60 apps (>10%) from the EXTENSIVE DATASET. After inspecting each app and identifying the click fraud actions, we show 100% precision over the randomly sampled dataset and this suggests the effectiveness of `ClickScanner`. With more time and effort, manual check of a sample size greater than 200 apps (>30%) would give a more pronounced precision rate. In view of the current best effort in manual verification, we highlight that currently there is no benchmark dataset publicly available for any accuracy-comparison of `humanoid attack` identification approaches. Table 3 shows that the proportion of SDK based cases among the EXTENSIVE DATASET is only 15.3% which is much less than that in the BASIC DATASET (*i.e.*, 83.4%). The possible reasons are as follows: 1) fraudulent SDKs are more likely to infect popular apps to commit more `humanoid attacks`; 2) developers of popular apps pay more attention to app security compliance, and generally will not add fraudulent codes to apps themselves. Therefore, we reveal that the `humanoid attack` has different properties depending upon app popularity, which will require different approaches for vetting.

5.5 Time Overhead

To evaluate the time overhead of `ClickScanner`, we first divide 120,000 apps from both the BASIC and EXTENSIVE datasets into five

---

[3]Since Huawei AppGallery does not provide download channels for historical versions, we only studied those apps on Google Play, which were downloaded from apkpure [5].

[4]We give more analysis of com.mo*** in Section 6.3.
[5]We did not collect apps from Huawei AppGalley since there are not as many apks available as on Google Play.

Table 3: Fraudulent app distribution by SDKs on Google Play and Huawei AppGallery

App market	Total downloads	Detection result		
		SDK	# of apps	Ratio
Huawei AppGallery	10,000	yes	72	67.0%
		no	22	33.0%
Google Play	10,000	yes	59	95.2%
		no	4	4.8%
Google Play	100,000	yes	90	15.3%
		no	494	84.7%

categories with the apk sizes of 0 ∼ 10 MB, 10 ∼ 50 MB, 50 ∼ 100 MB, 100 ∼ 200 MB and above 200 MB respectively. Then we record the average running time of `ClickScanner` on each category. Note that the timeout of `ClickScanner` is set to 300 seconds, and only 742 out of 120,000 apps (0.62%) do not terminate within 300 seconds.

We further compare the time overhead of `ClickScanner` with other previous tools (*e.g.*, FraudDetective [20], FraudDroid [14], MAdFraud [11], DECAF [28], AdSherlock [8], and Clicktok [32]). Table 4 lists the average time overhead of `ClickScanner` and other tools.

FraudDetective tests each app for five minutes. FraudDroid takes at most 216 seconds to analyze an app. MAdFraud needs 120 seconds to run on average. The mean and median time for analyzing an app using DECAF is 11.8 minutes and 11.25 minutes respectively. AdSherlock executes each app in one emulator instance twice, which costs 10 minutes to extract ad fraud traffic patterns offline. Clicktok does not list its runtime performance but it also needs to execute each app for a chosen duration of time to interact with apps using the Monkeyrunner tool [18]. It is observed that the average `ClickScanner` time cost for detecting `humanoid attacks` is about 18.4 seconds per app.

It is worth noting that even in the best case (*i.e.*, 120 seconds in MAdFraud), the time overhead is nearly 6.5 times as much as `ClickScanner`. These results demonstrate that `ClickScanner` significantly outperforms existing tools in terms of the time overhead and that it is practical to deploy `ClickScanner` in the real world. Additionally, it is observed that when changing apk size, the detection time correlated to apk size is relatively stable.

6 CASE STUDY

In this section, we closely analyze the fraud apps discovered by `ClickScanner`. We present four representative `humanoid attack` cases and explain newly obtained insights into how attackers commit mobile click fraud.

6.1 Case 1. Humanoid Attacks after User's Legitimate Actions

To fool the traditional detectors, some `humanoid attacks` will follow the user's legitimate actions. For instance, one fraudulent app, named com.co****.***********, is a communication and social app


```
static final void onClick(AdWebView){
    ......
1.   if(randNum > 0f && randNum > new Random().nextFloat()) {
2.       performRandomClick(AdWebView,...);
     }
......}
```
Generate fake click with random axis values after meeting the random condition

```
private static void performRandomClick(View AdWebView, ...) {
    ......
1.float randWidth = new Random.nextFloat() * AdWebView.getWidth();
2.float randHeight= new Random.nextFloat() * AdWebView.getHeight();
3.MotionEvent v9 = MotionEvent.obtain(..., randx, randy,...);
4.AdWebView.dispatchTouchEvent(v9);
    ......
}
```

Figure 10: The code snippet from case 1 in the case study. The timing patterns of click fraud vary according to user actions.

with a total of over 570 million downloads across all app markets searched on [23] and `ClickScanner` successfully reveals its process of generating `humanoid attacks` in Fig. 10. It first displays the advertisement to the users, and when the users click on the advertisement, it generates a set of random numbers in the trigger condition of fake clicks so that the `humanoid attack` is triggered randomly. The same method is also used in the method of randomly generating click coordinates for `humanoid attacks`. The potential threats with this app are as follows. (*i*) Click frauds as such are difficult to detect with traditional detection methods, because their traffic patterns and click patterns vary with different users and are highly similar to real human clicks. Therefore, the traditional method of identifying click fraud that relies on the difference between patterns of click fraud and normal clicks is ineffective. (*ii*) Attackers will even defraud advertisers for more advertising costs on the grounds that a considerable part of their users are interested in the advertisements they distribute and click twice. (*iii*) We also find that the app is not alone in implementing fraudulent behaviors. Four other apps produced by the same company are manifested to have similar click fraud codes. Their total downloads reached over 658M across all app markets, therefore they have caused huge losses for advertisers.

6.2 Case 2. Humanoid Attack that Can Adaptively Avoid Detection

Some apps predefine fake click's execution logic in their code and receive the click's configuration from remote servers, which can be changed at any time according to the situation. The occurrence of fake clicks will be adaptively controlled locally to avoid detection. As shown in Fig. 11, com.m*.*******.**** is a photography app with over 9.5 billion downloads in all app markets searched on [23] and over 50 million installations since the time it was made available in Google Play. `ClickScanner` has uncovered it conducting `humanoid attacks` against ad agencies. It first loads a configuration URL starting with "https://l**.u****.com:8***/l**/s?" and parses out data such as "ctr", "cvr", "max_clk" and "max_imp" in the returned JSON data structure as fake click configuration. Then it will count every time a fake click is triggered. When the number of fake clicks is greater than the value of the maximum number of clicks (max_clk) sent by the remote server, it will stop conducting the `humanoid attack`. If the number of fake clicks is smaller than "max_clk", it will get the click properties, such as duration, action (up or down), and

Table 4: Runtime overheads evaluation of `ClickScanner` with other tools.

Tools	FraudDetective [20]	FraudDroid [14]	MAdFraud [11]	DECAF [28]	AdSherlock [8]	ClickScanner					
						0-20(MB)	20-50(MB)	50-100(MB)	100-200(MB)	200+(MB)	avg
avg_time(s)	300	216	120	675	600	16.37	20.82	16.11	16.79	22.02	18.42

If the number of fake clicks does not exceed the *max_clk* sent by the server, the ad will be loaded after a random delay and humanoid attack will be implemented

```
new Handler("").postDelayed("" {
  if(clk_count < max_clk) {
    public void run() {
      ......
      loadAD(url);
      ......
    }
  }
},(new Random() .nextInt(6)*1000);
```

```
public void loadAD(url) {
  this.a.loadUrl(url);
  ++ clk_count;

  public boolean shouldOverrideUrlLoading
    (WebView AdView, String ADConfig) {
    ......
    parseConfig(ADConfig);
```
}Start parsing the configuration information of fake click issued by the server

```
private void parseConfig(String ADConfig) {
  String[] clkConfig = ADConfig.split("/");
  this.adClk(Long.parseLong(clkConfig [0]),Integer.parseInt(clkConfig [1]),
        Float.parseFloat(clkConfig[2]),Float.parseFloat(clkConfig[3]));
}
```

```
public static void adClk("", clk_type, clkConfig_x, clkConfig_y, "") {
  MotionEvent v9=MotionEvent.obtain("clk_type, clkConfig_x, clkConfig_y,"");
  adView.onTouchEvent(v9);
  ......Implement humanoid attack based on the parsed click coordinates and click type
}
```

Figure 11: The code snippet from the case 2 in the case study. The configuration and commands of `humanoid attack` come from the server, and the occurrence of fake click will be adaptively controlled locally to avoid detection.

coordinates from the JSON data structure returned from the configuration URL to construct the *MotinoEvent* object and implement the `humanoid attack`. In doing so, the app can automatically execute a fake click on a random point in the ad view without the user actually clicking on the phone screen. The traffic of this fake click is identical to what would be generated by a real person, and hence the app adaptively avoids detection.

6.3 Case 3. Humanoid Attack through Infected Ad SDKs

The above two apps involve only one developer (although the `humanoid attack` code snippets involved in case 1 exist in multiple apps developed by the same company). As we mentioned in the Section 5, fraudulent developers now have shifted from directly developing app applications to developing SDKs, which can make their click fraud code affect more apps. "com.mo***" is an advertising SDK with `humanoid attack` codes involved in 43 apps out of 120,000 apps. It will override the `onClick()` method when initializing the ad view. This ad view will automatically click itself again when it is clicked by a user, which causes the ad view to be clicked twice in the advertiser's view. This fraudulent SDK has a greater impact than the previous two because all apps that install this SDK will participate in fraudulent activities intentionally or unintentionally. The total number of app installations affected by this SDK reaches 270 million since they were made available on Google Play. It is worth noting that we found the source code of this SDK on the GitHub, and the developer deleted this code by himself on November 5, 2020. We guess this may be related to Google's increasingly strict anti-ad-fraud measures.

6.4 Case 4. Humanoid Attack by Disguising as an Auto-play Video Assist Function

Some fraudulent activities are not only limited to clicking on static ads, but also pretend to be an accessibility service to help users to automatically play videos at the code level, but its real purpose is to click on the ad video. "com.iB***" SDK is a typical representative of this kind of `humanoid attack`. It initializes a VideoWebView class in the VideoAdActivity function, and predefines an `onReadyPlay` callback function in this class. In this callback function, it will check whether the ad video is playing, and if it is not playing, it will automatically click the ad video. The "com.iB***" SDK forces people who want to leave because they are not interested in the ad video to watch the video to increase their ad revenue. The download number of the apps affected by this SDK is over 476 million across all app markets searched on [23].

7 DISCUSSION

Scalability of `ClickScanner`. In this work, we proposed a new time-saving and efficient framework for click fraud detection. To foster a broader impact, we implemented an efficient backward program slicing framework in `ClickScanner` which can detect the assignment process of any variable or parameters of any API. Hence, our framework can be easily extended by researchers for targeted and efficient security vetting of modern Android apps.[6] In the future, we will also enhance `ClickScanner` to search over native code, and also extend it to other problems.

Analysis of misclassification cases. By manual analysis, the reasons for misclassifications are as follows: For the 11 apps in the SEED APPS that are misclassified by `ClickScanner`: 1) False positives: some apps simulate a human click to obtain the focus of the window. Although Android provides a standardized API to achieve this function, obtaining focus through this inappropriate method is rare. 2) False negatives: some views, although they include ad contents, are not regarded as ad views by `ClickScanner`. For the 9 false positives in the BASIC DATASET: 4 of them simulate a human click to obtain the focus of the window; 3 of them simulate a human click to automatically play a non-ad video; 2 of them are game apps, which achieve the expected game effect by shielding the original user's click event in the game webview and generating new click events. To solve these issues, expending more effort on checking whether the click event target is the input box or non-ad video player in the webview, or improving the accuracy of ad view detection are both feasible solutions.

Limitation 1: Fraud codes and instructions issued by remote servers. One of the major limitations in `ClickScanner` is the inability to detect complex JavaScript or encrypted instructions that do not reside in the original fraudulent app and come from a command and control server, which is almost impossible to detect at the code level. To put a detail on `ClickScanner`, it can detect click

---

[6]Publicly available at https://github.com/Firework471/ClickScanner.

fraud generated by the local code. However, if all the execution codes or commands are sent by a server at run time, due to the intrinsic problem of static analysis, `ClickScanner` can only know that the app will dynamically execute some code at a certain moment. Because `ClickScanner` cannot fetch the content of the code, it is impossible to know whether this code is used to perform click fraud. Hence, one future direction of this research is to support context-based static analysis to infer the purpose of the loaded code or instruction and analyze the dynamically loaded code using tools like DyDroid [37].

Limitation 2: Other types of ad views. Although we have used one of the most popular and obfuscation-resilient tools, LibRadar- [30], with several constraints to identify ad views, our taxonomy may still be incomplete since it was built based on the current policies and literature available. This is also one of the main causes of the false-positives and false-negatives. Nevertheless, we can adjust the threshold appropriately to make a trade-off in the result, or we can use more tools [7, 27] to identify ad libraries and ad views. Additionally, due to the scalability of `ClickScanner`, `ClickScanner` is generic and can be extended to support the detection of click frauds on potential new types of ad views.

Limitation 3: Obfuscation. `ClickScanner` 's ability is constrained to its employed static analysis tool FlowDroid. `ClickScanner` can live well with lightweight obfuscation mechanisms since the logic of humanoid attacks is exposed. However, it cannot handle apps that adopt advanced obfuscations or packing techniques to prevent analysis of the app's bytecode. To address this issue, `ClickScanner` could actively interact with other techniques (*e.g.*, deobfuscation, unpacking, binary analysis) to recover the protected bytecode.

8 RELATED WORK

In recent years, research in click fraud detection has mainly focused on dynamic analysis. Some of these approaches analyze ad network traffic and summarize a pattern of click-fraud traffic. Others rely on installing an additional patch or SDK on users' devices to check whether the ad click is fraudulent or not by checking the click pattern of touch events.

Traffic patterns associated with ad network traffic. Some previous works claimed that fraudulent clicks have different traffic patterns from benign ones. FraudDroid [14] builds UI state transition graphs and collects their associated runtime network traffics, which are then leveraged to check against a set of heuristic-based rules for identifying ad fraudulent behaviors. MAdFraud [11] automatically runs apps simultaneously in emulators in the foreground and background for 60 seconds each and found ad click traffic that occurred under the testing environment involving no user interaction. Clicktok [32] detects clickspam by searching for reflections of click traffic, encountered by an ad network in the past. Detection with traffic analysis depends primarily on the network traffic set gathered.

Local pattern associated with click events. Some previous works claimed that fraudulent clicks have different performance on users' devices compared to benign ones. FraudDetective [20] computes a full stack trace from an observed ad fraud activity to a user event and generates the causal relationships between user inputs

and the observed fraudulent activity. AdSherlock [8] injects the online detector into the app executable archive and marks the touch events as click fraud when the Android kernel does not generate a *MotionEvent* object or the properties of the generated *MotionEvent* object remain unchanged. DECAF [28] performs dynamic checking in an emulator and marks ad frauds if the layout or page context violates a particular rule. ClickGuard [42] takes advantage of motion sensor signals from mobile devices since the pattern of motion signals is different under real click events and fraud events. However, ClickGuard will likely cause participants concern over data collection [16, 25, 44].

These tools above played an important role in revealing the occurrence of mobile click fraud. However, if fraudulent apps (*e.g.*, the apps implementing `humanoid attacks` proposed in this study) simulate the real human's clicks patterns to bridge the gap between normal clicks and fake clicks in the click patterns, or hide their fraudulent behaviors while executing, or only trigger clicks in a certain period, models may fail at the very first stage. Crucially, most of the previous strategies cannot pinpoint which app class conducts click fraud.

9 CONCLUSION

In this paper, we explored a new and sophisticated click fraud, named `humanoid attack`, and we successfully revealed its attack patterns through static analysis. We designed and implemented `ClickScanner` for uncovering `humanoid attacks`. By applying `ClickScanner` to measure real-world market apps containing 120 thousand apps, `ClickScanner` identifies 157 fraudulent apps from 20,000 top-rated apps in Google Play and Huawei AppGallery. Our work also informs the impact of ad SDKs on click fraud and the distribution of fraudulent apps among different categories and popularity. In conclusion, our work suggests that the `humanoid attack` is still widespread in the current app markets and we hope that the detection tool `ClickScanner` developed in this paper can effectively combat the emerging `humanoid attack`.

REFERENCES

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. Association for Computing Machinery, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[2] Android. 2020. The 20 Most Popular Android Apps in the Google Play Store. https://developer.android.com/reference/android/app/Activity.

[3] Android. 2020. MotionEvent Android Developers. https://developer.android.com/reference/android/view/MotionEvent.

[4] Android. 2020. MotionEvent.obtain Android Developers. https://developer.android.com/reference/android/view/MotionEvent#obtain(long,long,int,float,float,int).

[5] Apkpure. 2020. Download APK free online downloader. https://apkpure.com/.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269.

[7] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 356–367.

[8] Chenhong Cao, Yi Gao, Yang Luo, Mingyuan Xia, Wei Dong, Chun Chen, and Xue Liu. 2021. AdSherlock: Efficient and Deployable Click Fraud Detection for Mobile Applications. *IEEE Transactions on Mobile Computing* 20, 4 (2021), 1285–1297. https://doi.org/10.1109/TMC.2020.2966991

[9] Gong Chen, Wei Meng, and John Copeland. 2019. Revisiting Mobile Advertising Threats with MAdLife. In *The World Wide Web Conference (WWW '19)*. Association for Computing Machinery, New York, NY, USA, 207–217.

[10] Geumhwan Cho, Junsung Cho, Youngbae Song, and Hyoungshick Kim. 2015. An Empirical Study of Click Fraud in Mobile Advertising Networks. In *Proceedings of the 2015 10th International Conference on Availability, Reliability and Security (ARES '15)*. IEEE Computer Society, USA, 382–388.

[11] Jonathan Crussell, Ryan Stevens, and Hao Chen. 2014. MAdFraud: Investigating Ad Fraud in Android Applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '14)*. Association for Computing Machinery, New York, NY, USA, 123–134.

[12] Vacha Dave, Saikat Guha, and Yin Zhang. 2012. Measuring and Fingerprinting Click-Spam in Ad Networks. *SIGCOMM Comput. Commun. Rev.* 42, 4 (Aug. 2012), 175–186.

[13] Android Developers. 2017. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.

[14] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F. Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. 2018. FraudDroid: Automated Ad Fraud Detection for Android Apps. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 257–268.

[15] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. TriggerScope: Towards Detecting Logic Bombs in Android Applications. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 377–396. https://doi.org/10.1109/SP.2016.30

[16] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC '12)*. Association for Computing Machinery, New York, NY, USA, 101–112.

[17] Hamed Haddadi. 2010. Fighting online click-fraud using bluff ads. *ACM SIGCOMM Computer Communication Review* 40, 2 (2010), 21–25.

[18] Google Inc. 2018. Monkeyrunner. https://developer.android.com/studio/test/monkeyrunner.

[19] Md. Shahrear Iqbal, Md. Zulkernine, Fehmi Jaafar, and Yuan Gu. 2016. FCFraud: Fighting Click-Fraud from the User Side. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 157–164. https://doi.org/10.1109/HASE.2016.17

[20] Sooel Son Joongyum Kim, Jung-hwan Park. 2020. The Abuser Inside Apps: Finding the Culprit Committing Mobile Ad Fraud. In *28th Network & Distributed System Security Symposium (NDSS '21)*. 1–16.

[21] Yosuke Kikuchi, Hiroshi Mori, Hiroki Nakano, Katsunari Yoshioka, Tsutomu Matsumoto, and Michel Van Eeten. 2016. Evaluating Malware Mitigation by Android Market Operators. In *Proceedings of the 9th USENIX Conference on Cyber Security Experimentation and Test (CSET'16)*. USENIX Association, USA, 4.

[22] John Koetsier. Aug 24,2020. Malicious Chinese SDK In 1,200 iOS Apps With Billions Of Installs Causing 'Major Privacy Concerns To Hundreds Of Millions Of Consumers'. https://www.forbes.com/sites/johnkoetsier/2020/08/24/malicious-chinese-sdk-in-1200-ios-apps-with-billions-of-installs-causing-major-privacy-concerns-to-hundreds-of-millions-of-consumers/.

[23] KUCHUAN. 2021. KUCHUAN. https://www.kuchuan.com/.

[24] Kaspersky Lab. 2021. Kaspersky Cyber Security Solutions for Home & Business. https://www.kaspersky.com.

[25] Pedro Leon, Blase Ur, Richard Shay, Yang Wang, Rebecca Balebako, and Lorrie Cranor. 2012. Why Johnny Can't Opt out: A Usability Evaluation of Tools to Limit Online Behavioral Advertising. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. Association for Computing Machinery, New York, NY, USA, 589–598.

[26] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon. 2016. An Investigation into the Use of Common Libraries in Android Apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 403–414.

[27] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: Scalable and Precise Third-Party Library Detection in Android Markets. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, 335–346. https://doi.org/10.1109/ICSE.2017.38

[28] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. 2014. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 57–70.

[29] McAfee LLC. 2021. McAfee Total Protection. http://mcafee.com/.

[30] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and Accurate Detection of Third-Party Libraries in Android Apps. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 653–656. https://doi.org/10.1145/2889160.2889178

[31] Bob Mungamuru and Stephen Weis. 2008. Competition and Fraud in Online Advertising Markets. In *Financial Cryptography and Data Security*, Gene Tsudik (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 187–191.

[32] Shishir Nagaraja and Ryan Shah. 2019. Clicktok: Click Fraud Detection Using Traffic Analysis. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '19)*. Association for Computing Machinery, New York, NY, USA, 105–116.

[33] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. 2017. Dark Hazard: Learning-based, Large-Scale Discovery of Hidden Sensitive Operations in Android Apps. In *24th Annual Network and Distributed System Security Symposium (NDSS '17)*. The Internet Society, San Diego, California, USA, 1–15.

[34] Paul Pearce, Vacha Dave, Chris Grier, Kirill Levchenko, Saikat Guha, Damon McCoy, Vern Paxson, Stefan Savage, and Geoffrey M. Voelker. 2014. Characterizing Large-Scale Click Fraud in ZeroAccess. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Association for Computing Machinery, New York, NY, USA, 141–152.

[35] Peng Peng, Limin Yang, Linhai Song, and Gang Wang. 2019. Opening the Blackbox of VirusTotal: Analyzing Online Phishing Scan Engines. In *Proceedings of the Internet Measurement Conference (IMC '19)*. Association for Computing Machinery, New York, NY, USA, 478–485.

[36] DAN PRICE. 2020. The 20 Most Popular Android Apps in the Google Play Store. https://www.makeuseof.com/tag/most-popular-android-apps/.

[37] Z. Qu, S. Alam, Y. Chen, X. Zhou, W. Hong, and R. Riley. 2017. DyDroid: Measuring Dynamic Code Loading and Its Security Implications in Android Applications. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 415–426.

[38] Juniper Research. 2019. Future Digital Advertising: Artificial Intelligence & Advertising Fraud 2019-2023. https://www.juniperresearch.com/researchstore/content-digital-media/future-digital-advertising-strategy-report/subscription/artificial-intelligence-advertising-fraud.

[39] Juniper Research. 2020. Advertising Fraud Losses to Reach $42 Billion in 2019, Driven by Evolving Tactics by Fraudsters. https://www.juniperresearch.com/press/press-releases/advertising-fraud-losses-to-reach-42-bn-2019.

[40] Juniper Research. 2021. juniperresearch. https://www.juniperresearch.com/home.

[41] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. 2012. AdSplit: Separating Smartphone Advertising from Applications. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 553–567. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/shekhar

[42] Congcong Shi, Rui Song, Xinyu Qi, Yubo Song, Bin Xiao, and Sanglu Lu. 2020. ClickGuard: Exposing Hidden Click Fraud via Mobile Sensor Side-channel Analysis. In *IEEE International Conference on Communications (ICC '20)*. Dublin, Ireland, 1–6.

[43] Sam Smith. 2019. ADVERTISING FRAUD LOSSES TO REACH $42 BILLION IN 2019, DRIVEN BY EVOLVING TACTICS BY FRAUDSTERS. https://www.juniperresearch.com/press/press-releases/advertising-fraud-losses-to-reach-42-bn-2019.

[44] Blase Ur, Pedro Giovanni Leon, Lorrie Faith Cranor, Richard Shay, and Yang Wang. 2012. Smart, Useful, Scary, Creepy: Perceptions of Online Behavioral Advertising. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS '12)*.

Association for Computing Machinery, New York, NY, USA, Article 4, 15 pages.

[45] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, Mississauga, Ontario, Canada, 13.

[46] VirusTotal. 2020. Virustotal. https://www.virustotal.com/gui/.

[47] Gang Wang, Tristan Konolige, Christo Wilson, Xiao Wang, Haitao Zheng, and Ben Y. Zhao. 2013. You Are How You Click: Clickstream Analysis for Sybil Detection. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 241–256. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/wang

[48] Kenneth C Wilbur and Yi Zhu. 2009. Click fraud. *Marketing Science* 28, 2 (2009), 293–308.

[49] Haitao Xu, Daiping Liu, Aaron Koehl, Haining Wang, and Angelos Stavrou. 2014. Click Fraud Detection on the Advertiser Side. In *Computer Security - ESORICS 2014*, Mirosław Kutyłowski and Jaideep Vaidya (Eds.). Springer International Publishing, Cham, 419–438.

[50] Linfeng Zhang and Yong Guan. 2008. Detecting Click Fraud in Pay-Per-Click Streams of Online Advertising Networks. In *2008 The 28th International Conference on Distributed Computing Systems (ICDCS '08)*. 77–84. https://doi.org/10.1109/ICDCS.2008.98

[51] Haizhong Zheng, Minhui Xue, Hao Lu, Shuang Hao, Haojin Zhu, Xiaohui Liang, and Keith Ross. 2018. Smoke screener or straight shooter: Detecting elite Sybil attacks in user-review social networks. In *25th Network & Distributed System Security Symposium (NDSS '18)*. 1–15.

[52] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. 2020. Measuring and Modeling the Label Dynamics of Online Anti-Malware Engines. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2361–2378. https://www.usenix.org/conference/usenixsecurity20/presentation/zhu

APPENDIX

A   RESULTS OF FRAUDULENT APPS AND SDKS

Table 5 lists some of the apps of a large download number which implement `humanoid attacks`. The full list will be submitted to Huawei and Google. The download number is only from Google Play or Huawei AppGallery. The highest number of downloads (from Huaweu AppGallery) reached 3.7 billion. `ClickScanner` successfully locates where these click fraud occurs in codes, which will facilitate the digital forensics of these fraudulent behaviors. Meanwhile, we mark out whether these `humanoid attacks` were caused by the advertising SDKs in the sixth column.

B   THE DISTRIBUTION OF APPS AFFECTED BY HUMANOID ATTACKS

To quantify the damage of `humanoid attacks` in the real world, it is important to know the category distributions of identified malicious apps. Thus, for each fraudulent app, we extract its category (*e.g.*, books, education, weather) labeled by app markets and illustrate the statistical results in Fig. 12. It is observed there is a significant difference between the categories of fraudulent apps from Google Play and Huawei AppGallery. For instance, among 25 categories, fraudulent apps from both app markets exist in only 8 categories.

Since the target users in Google Play and Huawei AppGallery are different (*i.e.*, the former mainly targeting U.S. and Europe, and the latter mainly targeting China), the difference of marketing is perhaps caused by the interest/cultural differences of users, and the users' interests in mobile ads [21]. For instance, the fraudulent apps detected in Huawei AppGallery are concentrated in "Education", "Books" and "Shopping" because China has a prosperous online shopping industry and a cultural emphasis on education, while in
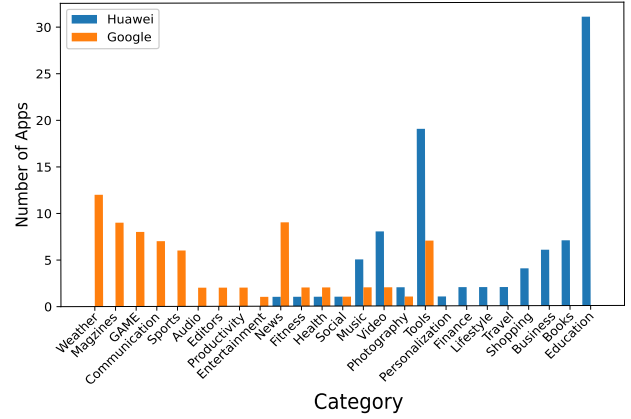

Figure 12: The number of fraudulent apps group by different categories on Google Play and Huawei AppGallery.
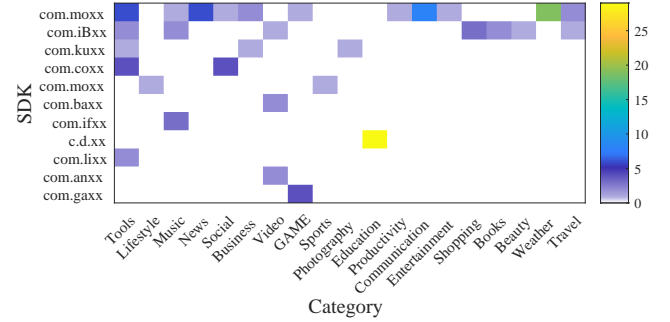
Figure 13: SDKs across app categories. The color encodes the number of apps involving fraudulent SDKs.

Google Play, the detected apps are concentrated in "News", "Magazines" due to the diversity of media in society. It is also worth noting that there are apps from the "Tools" category marked by `ClickScanner` in both markets. This may be because it is easier to obtain system permissions to hide fraudulent activities for these types of apps. In summary, these discoveries indicate that application markets in different regions need to focus on vetting different types of applications.

C   THE DISTRIBUTION OF FRAUDULENT SDKS

We also analyze the distribution of fraudulent SDKs of all 157 apps according to their categories. As shown in Fig. 13, the fraudulent apps existing in nearly half of the categories (19/45) are affected by 11 SDKs in total. This shows that the use of SDK to commit `humanoid attack` is not an isolated case against a certain category of apps, but a common method deserving attention. Furthermore, it is observed that the com.mo*** SDK and com.iB*** SDK have infected a large proportion of applications. Therefore, to effectively thwart `humanoid attacks`, it is vital to put efforts on vetting SDKs as well as the apps.

Table 5: Details of `humanoid` attacks in apps from Google Play and Huawei AppGallery

Package Name	Version	Catgory	Download	Fraud Location	SDK based
com.m*.********	8.7.1.5	Photography	3.7B	com.wa***	No
com.ba**********	7.33.0	Others	200.0M	com.ba***: void continueDispatchTouchEvent()	Yes
com.if******************	7.3.75	Music	200.0M	com.if***: void startClickConfirmBtn()	Yes
com.ut***********	6.5.5	Video Players & Editors	100.0M	com.mo***: void onClick()	Yes
com.ac***************	6.1.10	Weather	100.0M	com.lo***: void e()	Yes
com.co**************	6.8.0.4	Social	100.0M	com.co***: void dispatchTouchEvent()	Yes
com.me*********	3.9.6.0	Photography	100.0M	com.ho***: void a()	No
t*.yi*******	5.10.2	Video	100.0M	com.an***: boolean a()	Yes
com.co**************	6.6.4.2	Social	100.0M	com.co***: void dispatchTouchEvent()	No
com.ai**	3.9.2.3076	Education	96.11M	com.dz***	Yes
com.ti***********	4.3.8	Tools	67.69M	com.ti***: void BMa()	No
com.qu************	1.7.0.0	Tools	57.22M	com.iB***: void a()	Yes
com.cl*************	1.4.7.2	Tools	50.M	com.mo***: void onClick()	Yes
com.pu***********	1.0.8	GAME	50.0M	com.ga***: void c()	Yes
com.fu*****************	3.5.8.7	Video	46.97M	cn.co***: void h()	No
com.hu*******************	6.7.0.3	Others	46.54M	com.co***: void dispatchTouchEvent()	No
com.if************	4.4.1264	Business	42.73M	com.if***: void run()	Yes
com.st*************	4.25	Lifestyle	41.11M	com.xm***: void a()	No
com.lw**********	1.29.32	Books	33.9M	com.lw***: void d()	No
com.du******	4.44.2	Education	31.04M	e***: void a(java.lang.Object,java.util.Map)	No
com.ma**********	2.4.0	Books	30.64M	com.ba***: boolean click()	No
com.xm*************	2.31.5	Shopping	29.15M	com.by***: void a()	Yes
com.dz.*****	3.9.2.3074	Books	24.92M	com.dz***	Yes
com.iy*****************	5.13.5.02	Books	24.73M	com.iB***: void autoClick()	Yes
com.xi*************	3.9.2.3069	Books	20.81M	com.dz***	Yes
com.hi*****************	1.7.2	Sports	19.82M	com.mo***: void c()	Yes
com.xi************	4.8.52	Video	19.55M	com.bi***: boolean O()	No
com.ot*********	1.18	Music	15.1M	com.ot***: void run()	No
com.pe*************	3.1.0	Lifestyle	12.01M	com.mo***: void c()	Yes
com.is****	3.9.2.3068	Books	10.71M	com.dz***	Yes
com.ca****************	3.4.7	Finance	10.17M	com.we***: void a()	No
com.sc**********	6.11.0.3	Entertainment	10.0M	com.mo***: void onClick()	Yes
com.ta**************	6.4.8	Communication	10.0M	com.mo***: void onClick()	Yes
com.li************	5.8.0	Communication	10.0M	com.mo***: void onClick()	Yes
com.im*********	4.10.1.13493	Video Players & Editors	10.0M	com.mo***: void onClick()	Yes
com.cl*****************	1.1.5.2	Tools	10.0M	com.ku***: void onViewClicked()	Yes
me*******.pr*.********	1.3.7	Social	10.0M	com.mo***: void onClick()	Yes
com.ba**********************	9.0.10	Sports	10.0M	com.mo***: void onClick()	Yes
com.cl*****************	1.1.5.2	Tools	10.0M	com.ku***: void onViewClicked()	Yes
com.sw***************	2.5.3	Photography	10.0M	com.ku***: void onViewClicked()	Yes
fm.ca************************	8.16.0	Music & Audio	10.0M	d***: void onClick()	No
com.po***********	6.4.0	News & Magazines	10.0M	e***: void onClick()	No
m*.go************	8.1.5	Music & Audio	10.0M	com.mo***: void onClick()	Yes
lo*****.fl******************	1.2.9	Health & Fitness	10.0M	d***: void a()	No
in******.he****************	1.0.6	Health & Fitness	10.0M	Ql***: void a()	No
com.aw*.*******	5.17.2-10	Weather	10.0M	com.lo***: void d()	Yes
ca********************	1.8.7	Communication	10.0M	com.mo***: void onClick()	Yes
com.xm*******	1.9.4	Tools	9.68M	com.by***: void a()	Yes
com.di*************	3.9.2.3074	Others	7.85M	com.dz***	No
com.lb*.********	6.1.2562	Tools	7.09M	pp***: void a(android.view.View,float,float)	No
com.if************************	7.2.67	Music	7.02M	com.iB***: void autoClick()	Yes
com.me**********************	5.8.0	Tools	5.0M	com.mo***: void onClick()	Yes
com.am******************************	4.7.0.693	Others	5.0M	m***: void onClick()	No
m*.ne************************	2.6.7	Communication	5.0M	c***: void onClick()	No
mo**.in***	16.6.0.50080	Weather	5.0M	com.mo***: void onClick()	Yes
com.wo************************	1.1.7	GAME	5.0M	com.ga***: void c()	Yes
com.fl*******************	6.6.6.1	Others	530.0K	com.co***: void dispatchTouchEvent()	No
com.xi**********	2.20.5	Shopping	480.0K	com.iB***: void a()	Yes
com.me***************************	6.0	Education	470.0K	c***: void a(java.lang.Object,java.util.Map)	Yes
com.yz**************	1.07	Others	250.0K	com.da***: MotionEvent createMotionEvent()	No
com.zh***********************	2.3.8	Books	250.0K	com.zh***: void clickView()	No
com.yz*************	1.07	Others	250.0K	com.da***: MotionEvent createMotionEvent()	No
com.ha**********	1.3.0	Travel	230.0K	com.iB***: void autoClick()	Yes
com.ha**********	1.3.0	Travel	230.0K	com.iB***: void autoClick()	Yes
com.bu**********	1.5.8	Productivity	100.0K	com.bu***: void emulateClick()	No
com.we***********	2.2.4	Video	20.0K	com.we***: void simulateClick()	No
com.se***********	2.3.10	Tools	10.0K	com.se***: void createClickEvent()	No