# DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps

*Bin Liu*[*], *Suman Nath*[‡], *Ramesh Govindan*[*], *Jie Liu*[‡]
[*]*University of Southern California,* [‡]*Microsoft Research*

## Abstract

Ad networks for mobile apps require inspection of the visual layout of their ads to detect certain types of *placement frauds*. Doing this manually is error prone, and does not scale to the sizes of today's app stores. In this paper, we design a system called DECAF to automatically discover various placement frauds scalably and effectively. DECAF uses automated app navigation, together with optimizations to scan through a large number of visual elements within a limited time. It also includes a framework for efficiently detecting whether ads within an app violate an extensible set of rules that govern ad placement and display. We have implemented DECAF for Windows-based mobile platforms, and applied it to 1,150 tablet apps and 50,000 phone apps in order to characterize the prevalence of ad frauds. DECAF has been used by the ad fraud team in Microsoft and has helped find many instances of ad frauds.

## 1 Introduction

Several recent studies have pointed out that advertising in mobile (smartphones and tablets) apps is plagued by various types of frauds. Mobile app advertisers are estimated to lose nearly 1 billion dollars (12% of the mobile ad budget) in 2013 due to these frauds [4]. The frauds fall under two main categories: (1) *Bot-driven frauds* employ bot networks or paid users to initiate fake ad impressions and clicks [4] (more than 18% impressions/clicks come from bots [13]), and (2) *Placement frauds* manipulate visual layouts of ads to trigger ad impressions and unintentional clicks from real users (47% of user clicks are reportedly accidental [12]). Mobile app publishers are incentivized to commit such frauds since ad networks pay them based on impression count [10, 7, 8], click count [7, 8], or more commonly, combinations of both [7, 8]. Bot-driven ad frauds have been studied recently [4, 20, 44], but placement frauds in mobile apps have not received much attention from the academic community.

**Contributions.** In this paper, we make two contributions. First, we present the design and implementation of a scalable system for automatically detecting ad placement fraud in mobile apps. Second, using a large collection of apps, we characterize the prevalence of ad placement fraud and how these frauds correlate with app ratings, app categories, and other factors.


**Figure 1:** *Placement Fraud Examples*

*Detecting ad fraud.* In Web advertising, most fraud detection is centered around analyzing server-side logs [54] or network traffic [42, 43], which are mostly effective for detecting bot-driven ads. These can also reveal placement frauds to some degree (e.g., an ad not shown to users will never receive any clicks), but such detection is possible only *after* fraudulent impressions and clicks have been created. While this may be feasible for mobile apps, we explore a qualitatively different approach: to detect fraudulent behavior by *analyzing the structure* of the app, an approach that can detect placement frauds more effectively and *before* an app is used (e.g., before it is released to the app store). Our approach leverages the highly specific, and legally enforceable, *terms and conditions* that ad networks place on app developers (Section 2). For example, Microsoft Advertising says developers must not "edit, resize, modify, filter, obscure, hide, make transparent, or reorder any advertising" [11]. Despite these prohibitions, app developers continue to engage in fraud: Figure 1 shows (on the left) an app in which 3 ads are shown at the bottom of a page while ad networks restrict developers to 1 per page, and (on the right) an app in which an ad is hidden behind UI buttons.

The key insight in our work is that manipulation of the visual layout of ads in a mobile app can be programmatically detected by combining two key ideas: (a) a UI automation tool that permits automated traversal of all the "pages" of a mobile app, and (b) extensible fraud checkers that test the visual layout of each "page" for

compliance with an ad network's terms and conditions. While we use the term ad fraud, we emphasize that our work deems as fraud any violation of published terms and conditions, and *does not attempt to infer whether the violations are intentional or not*.

We have designed a system called DECAF that leverages the insight discussed above (Section 3). First, it employs an automation tool called a *Monkey* that, given a mobile app binary, can automatically execute it and navigate to various parts of the apps by simulating user interaction (e.g., clicking a button, swiping a page, etc.). Abstractly, a Monkey traverses a state transition graph, where pages are states, and a click or swipe triggers a transition from one state to the next. The idea of using a Monkey is not new [48, 37, 35]. The key optimization goals of designing a Monkey are good coverage and speed—the Monkey should be able to traverse a good fraction of target states in a limited time. This can be challenging as for many apps with dynamically varying content (e.g., news apps), the state transition graph can be infinitely large. Even for relatively simpler apps, recent work has shown that naïve state traversal based on a UI automation framework can take several hours [35]. Combined with the size of app stores (over a million apps on Google Play alone), this clearly motivates the need for *scalable* traversal[1]. Recent works therefore propose optimization techniques, many of which require instrumenting apps [48] or the OS [37].

A key feature of DECAF is that *it treats apps and underlying OS as black boxes* and relies on a UI automation framework. The advantage of this approach is its flexibility: DECAF can scan apps written in multiple languages (e.g., Windows Store apps can be written in C#, HTML/JavaScript, and C++) and potentially from different platforms (e.g., Windows and Android). However, the flexibility comes at the cost of very limited information from the UI automation framework. Existing automation frameworks provide only information about UI layout of the app's current page and do not provide information such as callback functions, system events, etc. that are required by optimizations proposed in [48] and [37]. To cope with this, DECAF employs several novel techniques: a *fuzzy matching*-based technique to robustly identify structurally similar pages with similar ad placement (so that it suffices for the Monkey to visit only one of them), a machine learning-based predictor to avoid visiting equivalent pages, an app usage based technique to prioritize app navigation paths, and a resource usage based technique for fast and reliable detection of page load completion. The techniques do not need app or OS instrumentation and greatly improves the Monkey's coverage and speed.

The second component of DECAF is to efficiently identify fraudulent behavior in a given app state. We find that, rather surprisingly, analyzing visual layout of an app page to detect possible ad fraud is nontrivial. This is due to complex UI layouts of app pages (especially in tablet apps), incomplete UI layout information from the UI automation framework (e.g., missing $z$-coordinate), mismatch between device's screen size and app's page size (e.g., panoramic pages), and variable behavior of ad networks (e.g., occasionally not serving any ad due to unavailability of specific types of ads), etc. We develop novel techniques to reliably address these challenges.

We have implemented DECAF to run on Windows 8 (tablet) apps and Windows Phone 8 apps (Section 5). Experiments show that DECAF achieves a coverage of 94% (compared to humans) in 20 minutes of execution per app and is capable of detecting many types of ad frauds in existing apps (Section 6).

*Characterizing Ad Fraud.* Using DECAF we have also analyzed 50,000 Windows Phone 8 apps and 1,150 Windows tablet apps, and discovered many occurrences of various types of frauds (Section 7). Many of these frauds were found in apps that have been in app stores for more than two years, yet the frauds remained undetected. We have also correlated the fraud data with various app metadata crawled from the app store and observed interesting patterns. For example, we found that fraud incidence appears independent of ad rating on both phone and tablet, and some app categories exhibit higher incidence of fraud than others but the specific categories are different for phone and tablet apps. Finally, we find that few publishers commit most of the frauds. These results suggest ways in which ad networks can selectively allocate resources for fraud checking.

DECAF has been used by the ad fraud team in Microsoft and has helped detect many fraudulent apps. Fraudulent publishers were contacted to fix the problems, and the apps whose publishers did not cooperate with such notices have been blacklisted and denied ad delivery. To our knowledge, DECAF is the first tool to automatically detect ad fraud in mobile app stores.

## 2 Background, Motivation, Goals and Challenges

**Background.** Many mobile app publishers use in-app advertisements as their source of revenue; more than 50% of the apps in major app stores show ads [30]. To embed ads in an app, an *app publisher* registers with a mobile *ad network* such as AdMob[6], iAd [8], or Microsoft Mobile Advertising [9]. In turn, ad networks con-

[1] It is also possible to scale by parallelizing the exploration of apps on a large cluster, but these resources are not free. Our efficiency improvements are orthogonal to parallelization and enable more efficient cluster usage, especially for apps that may have to be repeatedly scanned because they contain dynamic content.

tract with *advertisers* to deliver ads to apps. Generally speaking, the ad network provides the publisher with an ad control (i.e., a library with some visual elements embedded within). The publisher includes this ad control in her app, and assigns it some screen real estate. When the app runs and the ad control is loaded, it fetches ads from the ad network and displays it to the user.

Ad networks pay publishers based on the number of times ads are seen (called *impressions*) or clicked by users, or some combination thereof. For example, Microsoft Mobile Advertising pays in proportion to total impression count $\times$ the overall click probability.

**Motivation.** To be fair to advertisers, ad networks usually impose strict guidelines (called *prohibitions*) on how ad controls should be used in apps, documented in lengthy *Publisher Terms and Conditions*. We call all violations of these prohibitions *frauds*, regardless of whether they are violated intentionally or unintentionally. There are several kinds of frauds.

*Placement Fraud.* These frauds relate to how and where the ad control is placed. Ad networks impose placement restrictions to prevent impression or click inflation, while the advertiser may restrict what kinds of content (i.e., ad context) the ads are placed with. For instance, Microsoft Mobile Advertising stipulates that a publisher must not "edit, resize, modify, filter, obscure, hide, make transparent, or reorder any advertising" and must not "include any Ad Inventory or display any ads ... that includes materials or links to materials that are unlawful (including the sale of counterfeit goods or copyright piracy), obscene,..." [11]. Similarly, Google AdMob's terms dictate that "Ads should not be placed very close to or underneath buttons or any other object which users may accidentally click while interacting with your application" and "Ads should not be placed in areas where users will randomly click or place their fingers on the screen" [1].

*Interaction Fraud.* Ad networks impose restrictions on fraudulent interactions with ad controls, such as using bots to increase clicks, repeatedly loading pages to generate frequent ad requests, or offering incentives to users. Publishers cannot cache, store, copy, distribute, or redirect any ads to undermine ad networks' business, nor can they launch denial of service attacks on the ad servers [11].

*Content Fraud.* This class of frauds refers to the actual contents within the ad control. Publishers should not modify ad contents, and must comply with content regulations on certain classes of apps (e.g., regulations preventing adult content in apps designed for children). So ad publishers are required to disclose to the ad network what type of apps (or pages) the control is used in, so that the ad network can filter ads appropriately.

Detecting violations of these prohibitions manually in mobile apps can be extremely tedious and error prone. This is because an app can have many pages (many content-rich tablet apps have upwards of hundreds of pages) and some violations (*e.g.,* ads hidden behind UI controls) cannot often be detected by visual inspection. This, combined with the large number of apps in app stores (over a million for both Google Play and Apple's App Store) clearly suggests the need for automation in mobile app ad fraud detection.

**Goals.** In this paper, we focus on automated detection of two categories of placement frauds in mobile apps.

*Structural frauds:* These frauds relate to how the ad controls are placed. Violators may manipulate the UI layout to inflate *impressions*, or to reduce ad's foot print on screen. This can be done in multiple ways:

- An app page contains **too many ads** (Microsoft Advertising allows at most 1 ad per phone screen and 3 ads per tablet screen [11]).
- Ads are **hidden** behind other controls (e.g., buttons or images) or placed **outside the screen**. (This violates the terms and conditions in [11, 1]). Developers often use this trick to give users the feel of an "ad-free app", or to accommodate many ads in a page while evading manual inspection.
- Ads are resized and made **too small** for users to read.
- Ads are overlapped with or placed next to actionable controls, such as buttons, to capture accidental clicks.

*Contextual frauds:* These frauds place ads in inappropriate contexts. For example, a *page context* fraud places ads in pages containing inappropriate (e.g., adult) content. Many big advertisers, especially the ones who try to increase brand image via display ads, do not want to show ads in such pages. Ad networks therefore prohibit displaying ads in pages containing "obscene, pornographic, gambling related or religious" contents [11] Publishers may violate these rules in an attempt to inflate impression counts.

Beyond detecting fraud, a second goal of this paper is to characterize the prevalence of ad fraud by type, and correlate ad fraud with app popularity, app type, or other measures. Such a characterization provides an initial glimpse into the incidences of ad fraud in today's apps, and, if tracked over time, can be used to access the effectiveness of automated fraud detection tools.

**Challenges.** The basic approach to detecting placement fraud automatically is to programmatically inspect the visual elements and content in an app. But, because of the large number of apps and their visual complexity (especially on tablets), programmed visual inspection of apps requires searching a large, potentially unbounded. space. In this setting, inspection of visual elements thus faces two competing challenges: *accuracy*, and *scalability*. A more complete search of the visual elements can

yield high accuracy at the expense of requiring significant computation and therefore sacrificing scalability. A key research contribution in this paper is to address the tension between these challenges.

Beyond searching the space of all visual elements, the second key challenge is to accurately identify ad fraud within a given visual element. Detecting structural frauds in an app page requires analyzing the structure of the page and ads in it. This analysis is more challenging than it seems. For example, checking if a page shows more than one ad (or $k$ ads in general) in a screen at any given time might seem straightforward, but can be hard on a *panoramic page* that is larger than the screen size and that the user can horizontally pan and/or vertically scroll. Such a page may contain multiple ads without violating the rule, as long as no more than one ad is visible in any scrolled/panned position of the screen (this is known as the "sliding screen" problem). Similarly whether an ad is hidden behind other UI controls is not straightforward if the underlying framework does not provide the depths (or, $z$-coordinates) of various UI controls. Finally, detecting contextual fraud is fundamentally more difficult as it requires analyzing the content of the page (and hence not feasible in-field when real users use the apps).

## 3 DECAF Overview

DECAF is designed to be used by app stores or ad networks. It takes a collection of apps and a set of fraud compliance rules as input, and outputs apps/pages that violate these rules. DECAF runs on app binaries and does not assume any developer input.

One might consider using static analysis of an app's UI to detect structural fraud. However, a fraudulent app can dynamically create ad controls or change their properties during run time and bypass such static analysis. Static analysis also fails to detect contextual fraud. DECAF therefore performs dynamic checking (analogous to several recent works [48, 37, 35, 47]) in which it checks the implementation of an app by directly executing it in an emulator.

Unlike previous efforts [48, 37], DECAF uses a *black-box* approach where it does not instrument the app binary or the OS. This design choice is pragmatic: Windows 8 tablet apps are implemented in multiple languages (C#, HTML/Javascript, C++)[2], and our design choice allows us to be language-agnostic. However, as we discuss later, this requires novel techniques to achieve high accuracy and high speed.

Figure 2 shows the architecture of DECAF. DECAF runs mobile apps in an emulator and interacts with the app through two channels: a *UI Extraction* channel for extracting UI elements and their layout in the current

[2]In a sample of 1,150 tablet apps, we found that about 56.5% of the apps were written in C#, 38.3% in HTML/Javascript, and 5.2% in C++.

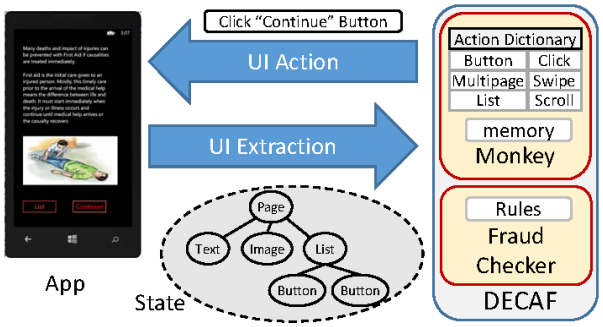



**Figure 2:** *The architecture of* DECAF *includes a Monkey that controls the execution and an extensible set of fraud detection policies.*

page of an app (shown as a Document Object Model (DOM) tree in Figure 2), and a *UI Action* channel for triggering an action on a given UI element (such as clicking on a button). In Section 5, we describe how these channels are implemented. DECAF itself has two key components: (1) a Monkey that controls the app execution using these channels and (2) a fraud checker that examines page contents and layout for ad fraud.

### 3.1 The Monkey

The execution of an app by a Monkey can be viewed as traversal on a state-transition graph that makes transitions from one *state* to the next based on UI inputs, such as clicking, swiping, and scrolling. Each state corresponds to a page in the app, and the Monkey is the program that provides UI inputs (through the UI Action channel) for each visited state.

At each state that it visits, the Monkey uses the UI extraction channel to extract *page information*, which includes (1) structural metadata such as size, location, visibility, layer information ($z$-index) of each ad and non-ad control in the current page, and (2) content such as the text, images, and urls in the page. The information is extracted from the DOM tree of the page; the DOM tree contains all UI elements on a given page along with contents of the elements. The Monkey also has a dictionary of actions associated with each UI type, such as clicking a button, swiping a multi-page, and scrolling a list, and uses this dictionary to generate UI inputs on the UI action channel.

Starting from an empty state and a freshly loaded app, the Monkey iterates through the UI controls on the page to the next state, until it has no transitions to make (either because all its transitions have been already explored, or it does not contain any actionable UI control). Before making a transition, the Monkey must wait for the current page to be completely loaded; page load times can be variable due to network delays, for example. After visiting a state, it uses one of two strategies. If a (hardware or software) back button is available, it retracts to a

previous (in depth-first order) state. If no back button is available (e.g., many tablets do not have a physical back button and some apps do not provide a software back button), the Monkey restarts the app, navigates to the previous state through a shortest path from the first page, and starts the exploration process.

In order to explore a large fraction of useful states within a limited time, the Monkey needs various optimizations. For example, it needs to determine if two states are equivalent so that it can avoid exploring states that have already been visited. It also needs to prioritize states, so that it can explore more important or useful states within the limited time budget. We discuss in Section 4 how DECAF addresses these. The Monkey also needs to address many other systems issues such as dealing with non-deterministic transitions and transient crashes, detecting transition to an external program (such as a browser), etc. DECAF incorporates solutions to these issues, but we omit the details here for brevity.

### 3.2 Fraud Checker

At each quiescent state, DECAF invokes the fraud checker. The checker has a set of *detectors*, each of which decides if the layout or page context violates a particular rule. While DECAF's detectors are extensible, our current implementation includes the following detectors.

*Small Ads:* The detector returns true if any ad in the given page is smaller than the minimal valid size required by the ad network. The operation is simple as the automation framework provides widths and heights of ads.

*Hidden Ads:* The detector returns true if any ad in the given page is (partially) hidden or unviewable. Conceptually, this operation is not hard. For each ad, the detector first finds the non-ad GUI elements, then checks if any of these non-ad elements is rendered above the ad. In practice, however, this is nontrivial due to the fact that existing automation frameworks (e.g., for Windows and for Android) do not provide $z$-coordinates of GUI elements, complicating the determination of whether a non-ad element is rendered *above* an ad. We describe in Section 4.4 how DECAF deals with this.

*Intrusive Ads:* The detector returns true if the distance between an ad control and a clickable non-ad element is below a predefined threshold or if an ad control partially covers a clickable non-ad control. Detecting the latter can also be challenging since the automation framework does not provide $z$-coordinates of UI elements. We describe in Section 4.4 how DECAF deals with this.

*Many Ads:* The detector returns true if the number of viewable ads in a screen is more than $k$, the maximum allowed number of ads. This can be challenging due to the mismatch of apps' page size and device's screen size. To address the sliding screen problem discussed before, a naïve solution would check all possible screen positions in the page and see if there is any violation at any position. We propose a more efficient solution in Section 4.4.

*Inappropriate Context:* The detector returns true if an ad-containing page has inappropriate content (e.g., adult content) or if the app category is inappropriate. Detecting whether or not page content is inappropriate is outside the scope of the paper; DECAF uses an existing system[3] that employs a combination of machine-classifiers and human inputs for content classification.

## 4 Optimizations for Coverage and Speed

The basic system described in Section 3 can explore most states of a given app[4]. However, this may take a long time: as [35] reports, this can take several hours for apps designed to have simple UIs for in-vehicle use, and our work considers content-rich tablet apps for which naïve exploration can take significantly longer. This is not practical when the goal is to scan thousands of apps. In such cases, the Monkey will have a limited *time budget*, say few tens of minutes, to scan each app; indeed, in DECAF, users specify a time budget for each app, and the Monkey explores as many states as it can within that time. With limited time, naïve exploration can result in poor *coverage* of the underlying state transition graph, and consequent inaccuracy in ad fraud detection. In this section, we develop various techniques to address this problem. The techniques fall under three general categories that we describe next.

### 4.1 Detecting Equivalent States

To optimize coverage, a commonly used idea is that after the Monkey detects that it has already explored a state equivalent to the current state, it can backtrack without further exploring the current state (and other states reachable from it). Thus, a key determinant of coverage is the definition of state equivalence. Prior work [35] points out that using a strict definition, where two states are equivalent if they have an identical UI layout, may be too restrictive; it advocates a heuristic for UI lists that defines a weaker form of equivalence.

DECAF uses a different notion of state equivalence, dictated by the following requirements. First, *the equivalence should be decided based on fuzzy matching* rather than exact matching. This is because even within the same run of the Monkey, the structure and content of the "same" state can vary due to dynamic nature of the corresponding page and variability in network conditions. For example, when the Monkey arrives at a state, a UI widget in the current page may or may not appear depending

[3] Microsoft's internal system, used by its online services.

[4] Without any human involvement, however, the Monkey can fail to reach states that require human inputs such as a login and a password.

on whether the widget has successfully downloaded data from the backend cloud service.

Second, *the equivalence function should be tunable* to accommodate a wide range of fraud detection scenarios. For detecting contextual frauds, the Monkey may want to explore all (or as many as possible within a given time budget) distinct pages of an app, so that it can check appropriateness of all contents of the app. In such a case, two states are equivalent only if they have the same content. For detecting structural frauds, on the other hand, the Monkey may want to explore only the pages that have unique structure (i.e., layout of UI elements). In such cases, two states with the same structure are equivalent even if their contents differ. How much fuzziness to tolerate for page structure and content should also be tunable: the ad network may decide to scan some "potentially bad" apps more thoroughly than the others (e.g., because their publishers have bad histories), and hence can tolerate less fuzziness on those potentially bad apps.

DECAF achieves the first requirement by using a flexible equivalence function based on cosine similarity of feature vectors of states. Given a state, it extracts various features from the visible elements in the DOM tree of the page. More specifically, the name of a feature is the concatenation of a UI element type and its level in the DOM tree, while its value is the count and total size of element contents (if the element contains text or image). For example, the feature `(TextControl@2, 100, 2000)` implies that the page contains 100 `Text` UI elements of total size 2000 bytes at level 2 of the DOM tree of the page. By traversing the DOM tree, DECAF discovers such features for all UI element types and their DOM tree depths. This gives a feature vector for the page that looks like: [(`Image`@2, 10, 5000 ), (`Text`@1, 10, 400), (`Panel`@2, 100, `null`),...].

To compare if two states are equivalent, we compute cosine similarity of their feature vectors and consider them to be equivalent if the cosine similarity is above a threshold. This configurable threshold achieves our second requirement; it acts as a tuning parameter to configure the strictness of equivalence. At one extreme, a threshold of 1 specifies *content* equivalence of two states[5]. A smaller threshold implies a more relaxed equivalence, fewer states to be explored by the Monkey, and faster exploration of states with less fidelity in fraud detection. To determine *structural* equivalence of two states, we ignore the size values in feature vectors and use a smaller threshold to accommodate slight variations in page structures. Our experiments indicate that a threshold of 0.92 strikes a good balance between thoroughness and speed while checking for structural frauds.

[5]On rare occasions, two pages with different content can be classified as equivalent if their text (or image) content has exactly the same count and total size. This is because we rely on count and size, instead of contents, of texts and images to determine equivalence of pages.

### 4.2 Path Prioritization

A Monkey is a general tool for traversing the UI state transition graph, but many (especially tablet) apps contain too many states for a Monkey to explore in a limited time budget. Indeed, some apps may even contain a practically infinite number of pages to explore. Consider a cloud-based news app that dynamically updates its content once every few minutes. In this app, new news pages can keep appearing *while* the Monkey is exploring the app, making the exploration a never-ending process. Given that the Monkey can explore only a fraction of app pages, without careful design, the Monkey can waste its time exploring states that do not add value to ad fraud detection, and so may not have time to explore useful states. This is a well-known problem with UI traversal using a Monkey, and all solutions to this problem leverage problem-specific optimizations to improve Monkey coverage. DECAF uses a novel *state equivalence prediction* method to prioritize which paths to traverse in the UI graph for detecting structural fraud, and a novel *state importance assessment* for detecting contextual fraud.

#### 4.2.1 State Equivalence Prediction

To motivate state equivalence prediction, consider exploring all structurally distinct pages of a news-serving app. Assume that the Monkey is currently in state $P_0$, which contains 100 news buttons (leading to structurally equivalent states $P_{0,0} \cdots P_{0,99}$ and one video button (leading to $P_{0,100}$). The Monkey could click the buttons in the same order as they appear in the page. It would first recursively explore state $P_{0,0}$ and its descendent states, then visit all the $P_{0,1-99}$ states, realize that that they are all equivalent to already visited state $P_{0,1}$, return to $P_0$. Finally, it will explore $P_{0,100}$ and its descendant states. This is clearly sub-optimal, since the time required to (1) go from $P_0$ to each of the states $P_{0,1-99}$ (forward transition) and (2) then backtracking to $P_0$ (backward transition) is wasted. The forward transition time includes the time for the equivalent page to completely load (we found this to be as large as 30 sec in our experiments).

**Backward transitions can be expensive.** The naïve strategy above can also be pathologically sub-optimal in some cases. Most mobile devices do not have a physical back button, so apps typically include software back buttons and our Monkey uses various heuristics based on their screen location and name to identify them. However, in many apps, the Monkey can fail to automatically identify the back button (e.g., if they are placed in unusual locations in the page and are named differently). In such cases the Monkey does not have any obvious way to directly go back to the previous page, creating unidirectional edges in the state graph. In our example, if

the transition between $P_0$ and $P_{0,1}$ is unidirectional, the backward transition would require the Monkey to restart the app and traverse through all states from the root to $P_0$, while waiting for each state to load completely before moving to the next state. Overall, the wasted time per button is as high as *3 minutes* in some of our experiments, and this can add up to a huge overhead if there are many such pathological traversals.

The net effect of above overheads is that the Monkey can run out of time before it gets a chance to explore the distinct state $P_{0,100}$. A better strategy would be to first explore pages with different UI layouts ($P_{0,0}$ and $P_{0,100}$ in previous example), and then if the time budget permits, to explore remaining pages.

**Minimizing state traversal overhead using prediction.** These overheads could have been minimized if there was a way to *predict* whether a unidirectional edge would take us to a state equivalent to an already visited state. Our *state equivalence prediction* leverages this intuition, but in a slightly different way. On a given page, it determines which buttons would likely lead to the same (or similar) states, and then explores more than one of these buttons *only if the time budget permits*. Thus, in our example, if the prediction were perfect, it would click on the button leading to the video page $P_{0,100}$ before clicking on the second (and third and so on) news button.

One might attempt to do such prediction based on event handlers invoked by various clickable controls, assuming that buttons leading to equivalent states will invoke the same event handler and those leading to different states will invoke different handlers. The event handler for a control can be found by static analysis of code. This, however, does not always work as event handlers can be bound to controls during run time. Even if the handlers can be reliably identified, different controls may not be bound to different handlers; for example, we found a few popular apps that bind most of their clickable controls to a single event handler, which acts differently based on runtime arguments.

DECAF uses a language-agnostic approach that only relies on the run-time layout properties of the various UI elements. The approach is based on the intuition that UI controls that lead to equivalent states have similar "neighborhoods" in the DOM tree: often their parents and children in the UI layout hierarchy are of similar type or have similar names. This intuition, formed by examining a number of apps, suggests that it might be possible to use machine-classification to determine if two UI controls are likely to lead to the same state.

Indeed, our approach uses supervised learning to construct a binary classifier for binary feature vectors. Each feature vector represents a pair of UI controls, and each element in the feature vector is a Boolean answer to the questions listed in Table 1. For any two UI controls, these questions can be answered from the DOM tree of the page(s) they are in. We construct a binary SVM classifier from a large labelled dataset; the classifier takes as input the feature vector corresponding to two UI controls, and determines whether they are likely to lead to equivalent states (if so, the UI controls are said to be equivalent).

| | |
|---|---|
| **Control Features** | Do they have the same name?<br>Do they have the same ID?<br>Are they with the same UI element type? |
| **Parent Features** | Do they have a same parent name path?<br>Do they have a same parent ID path?<br>Do they have a same parent UI element type path? |
| **Child Features** | Do their children share a same name set?<br>Do their children share a same ID set?<br>Do their children share a same UI element type set? |

**Table 1:** *SVM classifier features*

In constructing the classifier, we explored various feature definitions, and found ones listed in Table 1 to be most accurate. For instance, we found that features directly related to a control's appearance (e.g., color and size) are not useful for prediction because they may be different even for controls leading to equivalent states.

Our Monkey uses the predictor as follows. For every pair of UI controls in a page, the Monkey determines whether that pair is likely to lead to the same state. If so, it clusters the UI controls together, resulting in a set of clusters each of which contains equivalent controls. Then, it picks one control (called the *representative control*) from each cluster and explores these; the order in which they are explored is configurable (e.g., increasing/decreasing by their cluster size, or randomly). The Monkey then continues its depth-first state exploration, selecting only representative controls in each state traversed. After all pages have been visited by exploring only representative controls, the Monkey visits the non-representative controls if the time budget permits. Algorithm 1 shows the overall algorithm. Note that the SVM-based clustering is also robust to dynamically changing pages—since the Monkey explores controls based on their clusters, it can simply choose whatever control is available during exploration and can ignore the controls that have disappeared between the time clusters were computed and when the Monkey is ready to click on a control.

#### 4.2.2 State Importance Assessment

State prediction and fuzzy state matching does not help with state equivalence computed based on page content, as is required for contextual fraud detection. In such cases, the Monkey needs to visit all content-wise distinct pages in an app, and apps may contain too many pages to be explored within a practical time limit.

DECAF exploits the observation that not all pages within an app are equally important. There are pages that users visit more often and spend more time than others. From ad fraud detection point, it is more important to check those pages first, as those pages will show more

**Algorithm 1** Cluster clickable controls

```
1:  INPUT: Set C of clickable controls of a page
2:  OUTPUT: Clickable controls with cluster labels
3:  for i from 1 to C.Length do
4:      C[i].clusterLabel = −1
5:  currentClusterLabel = 1
6:  for i from 1 to C.Length do
7:      if C[i].clusterLabel < 0 then
8:          C[i].clusterLabel = currentClusterLabel
9:          currentClusterLabel++
10:     for j from i+1 to C.Length do
11:         if C[j].clusterLabel < 0 then
12:             {SVM_predict outputs true if two input controls is predicted to be in a same cluster}
13:             if SVM_Predict(C[i],C[j]) then
14:                 C[j].clusterLabel = C[i].clusterLabel
```

ads to users. DECAF therefore prioritizes its exploration of app states based on their "importance"—more important pages are explored before less important ones.

**Using app usage for estimating state importance.** The importance of a state or page is an input to DECAF and can be obtained from app usage statistics from real users, either by using data from app analytic libraries such as Flurry [5] and AppInsight [50] or by having users use instrumented versions of apps.

From this kind of instrumentation, it is possible to obtain a collection of *traces*, where each trace is a path from the root state to a given state. The importance of a state is determined by the number of traces that terminate at that state. Given these traces as input, DECAF combines the traces to generate a *trace graph*, which is a subgraph of the state transition graph. Each node in the trace graph has a *value* and a *cost*, where value is defined as the importance of the node (defined above) and cost is the average time for the page to load.

**Prioritizing state traversal using state importance.** To prioritize Monkey traversal for contextual fraud, DECAF solves the following optimization problem: given a cost budget $B$ (e.g., total time to explore the app), it determines the set of paths that can be traversed within time $B$ such that total value of all nodes in the paths is maximized. The problem is NP-Hard, by reduction from Knapsack, so we have evaluated two greedy heuristics for prioritizing paths to explore: (1) *Best node*, which chooses the next unexplored node with the best value to cost ratio, and (2) *Best path*, which chooses the next unexplored path with the highest total value-total cost ratio. We evaluate these heuristics in Section 6.

Since app content can change dynamically, it is possible that a state in a trace disappears during the Monkey's exploration of an app. In that case, DECAF uses the trained SVM to choose another state similar to the original state. Finally, traces can be useful not only to identify important states to explore, but also to navigate to states that require human inputs. For example, if navigating to a state requires username/password or special text inputs that the Monkey cannot produce, and if traces incorporate such inputs, the Monkey can use them during its exploration to navigate to those states.

### 4.3 Page Load Completion Detection

Mobile apps are highly asynchronous but UI extraction channels typically do not provide any callback to an external observer when the rendering of a page completes. Therefore, DECAF has no way of knowing when the page has loaded in order to check state equivalence. Fixed timeouts, or timeouts based on a percentile of the distribution of page load times, can be too conservative since these distributions are highly skewed. App instrumentation is an alternative, but has been shown to be complex even for managed code such as C# or Java [50].

DECAF uses a simpler technique that works for apps written in any language (including C++, html/javascript). It uses a *page load monitor* which it monitors all I/O (Networking, Disk and Memory) activities of the app process, and maintains their sum over a sliding window of time $T$. If this sum goes below a configurable threshold $\varepsilon$, the page is considered to be loaded; the intuition here is that as long as the page is loading, the app should generate non-negligible I/O traffic. The method has the virtue of simplicity, but comes at a small cost of latency, given by sliding window length, to detect the page load.

### 4.4 Fraud Checker Optimizations

DECAF incorporates several scalability optimizations as part of its fraud checkers.

**Detecting too many ads.** As mentioned in Section 3, detecting whether a page contains more than $k$ ads in any given screen position can be tricky. DECAF uses a more efficient algorithm whose computational complexity depends only on the total number of ads in the page and not on the page or screen size. The algorithm uses a vertical moving window across the page whose width is equal to the screen width and height is equal to the page height; this window is positioned successively at the right edges of rectangles representing ads. Within each such window, a horizontal strip of height equal to the screen height is moved from one ad rectangle bottom-edge to the next; at each position, the algorithms computes the number of ads visible inside the horizontal strip, and exits if this number exceeds a certain threshold. The complexity of this algorithm shown in Algorithm 2, is $O(N^2 \log(N))$, where $N$ is the total number of ads in the page.

**Algorithm 2** Detect ad number violation

1: **INPUT:** Set $D$ of ads in a page, where $D[k]_{R,x}$ and $D[k]_{R,y}$ are the $x$ and $y$ coordinates of the bottom-right corner of the $k$-th ad, and $D[k]_{L,x}$ and $D[k]_{L,y}$ are about the top-left corner; ad number limit $U$
2: **OUTPUT:** Return **true** if the violation is detected
3: $D_x = \{QuickSort\ D$ by $D[k]_{R,x}$'s of ads$\}$
4: $D_y = \{QuickSort\ D$ by $D[k]_{R,y}$'s of ads$\}$
5: **for** $i$ **from** 1 **to** $D$.Length **do**
6: $\quad S_x = \{BinarySearch$ on $D_x$ to get ads with $D[k]_{R,x} \geq D[i]_{R,x}$ and $D[k]_{L,x} \leq D[i]_{R,x} + screenWidth\ \}$
7: $\quad D_y(S_x) = \{$the subset of $D_y$ which is formed by elements in $S_x\}$
8: $\quad$ **for** $j$ **from** 1 **to** $D_y(S_x)$.Length **do**
9: $\quad\quad S_y = \{BinarySearch$ on $D_y(S_x)$ to get ads with $D[k]_{R,y} \geq D_y(S_x)[j]_{R,y}$ and $D[k]_{L,y} \leq D_y(S_x)[j]_{R,y} + screenHeight]\}$
10: $\quad\quad$ **if** $S_y$.Length$> U$ **then**
11: $\quad\quad\quad$ **return true**
12: **return false**

**Detecting hidden and intrusive ads.** As discussed in Section 3, determining if an ad is completely hidden or partially overlapped by other GUI elements is challenging due to missing $z$-coordinates of the elements. To deal with that, DECAF uses two classifiers described below.

*Exploiting DOM-tree structure.* This classifier predicts relative $z$-coordinates of various GUI elements based on their rendering order. In Windows, rendering order is the same as the depth-first traversal order of the DOM tree; i.e., if two elements have the same $x$- and $y$-coordinates, the one at the greater depth of the DOM tree will be rendered over the one at the smaller depth. The classifier uses this information, along with $x$- and $y$-coordinates of GUI elements as reported by the automation framework, to decide if an ad element is hidden or partially overlapped by a non-ad element.

This classifier, however, is not perfect. It can occasionally classify a visible ad as hidden (i.e., false positives) when the GUI elements on top of the ad are invisible and the visibility status is not available from the DOM tree information.

*Analyzing screenshots.* This approach uses image processing to detect if a target ad is visible in the app's screenshots. It requires addressing two challenges: (1) *knowing what the ad looks like*, so that the image processing algorithm can search for target ad, and (2) *refocusing*, i.e., making sure that the screenshot captures the region of the page containing the ad.

To address the first challenge, we use a proxy that serves the apps with *fiducials*: dummy ads with easily identifiable patterns such as a checker-board pattern. The proxy intercepts all requests to ad servers and replies with fiducials without affecting normal operations of the app. The image processing algorithm then looks for the specific pattern in screenshots. To address the refocusing challenge, the Monkey scrolls and pans app pages and analyzes screenshots only when the current screen of the page contains at least one ad.

The classifier, like the previous one, is not perfect either. It can classify hidden ads as visible and vice versa due to imperfections in the image processing algorithm (especially when the background of the app page is similar to the image pattern in the dummy ad) and to the failure of the Monkey to refocus.

*Combining the classifiers.* The two classifiers described above can be combined. In our implementation, we take a conservative approach and declare an ad to be hidden if it is classified as hidden by both the classifiers.

### 4.5 Discussion

**Parallel Execution.** DECAF can scan multiple apps in parallel. We did not investigate scanning a single app with multiple Monkeys in parallel. This is because scanning at the granularity of an app is sufficient in practice (See Section 6.3) and scanning at any finer granularity introduces significant design complexity (e.g., to coordinate Monkeys scanning the same app and to share states among them).

**Smarter Ad Controls.** DECAF's fraud checker optimizations can be implemented within the ad control. Such a "smart" ad control would perform all the DECAF checks while users use an app and take necessary actions if ads are to be shown in fraudulent ways (e.g., not serving ad to or disregarding clicks from the app or the page). This, however, introduces new challenges such as permitting communication among ad controls on the same app page, preventing modification of the ad control library through binary code injection, permitting safe access to the entire app page from within the ad control[6], etc., so we have left an exploration of this to future work.

## 5 Implementation

**Tablet/Phone differences.** We have implemented DECAF for Windows Phone apps (hereafter referred to as *phone apps*) and Windows Store apps (referred to as *tablet apps*). One key difference between our prototypes for these two platforms is how the Monkey interacts with apps. Tablet apps run on Windows 8, which provides Windows UI automation framework (similar to

[6]Especially for HTML apps, an ad control is not able to access content outside its frame due to the same origin policy.

Android MonkeyRunner [2]); DECAF uses this framework directly. Tablet apps can be written in C#, HTML/JavaScript, or C++ and the UI framework allows interacting with them in a unified way. On the other hand, DECAF runs phone apps in Windows Phone Emulator, which does not provide UI automation, so we use techniques from [48] to extract UI elements in current page and manipulate mouse events on the host machine running the phone emulator to interact with apps.

**Other implementation details.** As mentioned earlier, tablets have no physical back buttons, and apps incorporate software back buttons. DECAF contains heuristics to identify software back buttons, but these heuristics also motivate state equivalence prediction. Furthermore, we use Windows performance counter [16] to implement the page load monitor.

To train the SVM classifier for state equivalence, we manually generated 1,000 feature vectors from a collection of training apps and used grid search with 10-fold cross validation to set the model parameters. The chosen parameter set had a highest cross-validation accuracy of 98.8%. Finally, for our user study reported in the next section, we use Windows Hook API [14] and Windows Input Simulator [15] to record and replay user interactions.

## 6 Evaluation

In this section, we evaluate the overall performance of DECAF optimizations. For lack of space, we limit the results to tablet apps only, since they are more complex than phone apps. Some of our results are compared with ground truth, which we collect from human users, but since the process is not scalable, we limit our study to the 100 top free apps (29 HTML/JavaScript apps and 71 C# apps) from the Windows Store. In the next section, we run DECAF on a larger set of phone and mobile apps to characterize ad frauds.

### 6.1 State Equivalence Prediction

To measure accuracy of our SVM model and its effectiveness in exploring distinct states, we need ground truth about distinct pages in our app set. We found that static analysis is not very reliable in identifying distinct states, as an app can include 3rd party libraries containing many page classes but actually use only a handful of them. We therefore use humans to find the ground truth. We gave all the 100 apps to real users, and asked them to explore as many unique pages as possible[7]. For each app, we combined all pages visited by users and counted the number of structurally distinct pages in the app. Since apps typically have relatively small number of structurally distinct pages (mean 9.15, median 5), we found humans to be effective in discovering all of them.

**Accuracy of SVM model.** We evaluated our SVM model on the ground truths and found that it has a false positive rate of 8% and false negative rate of 12%[8]. Note that false negatives do not affect the accuracy of the Monkey; it only affects the performance of the Monkey by unnecessarily sending it to equivalent states. However, false positives imply that the Monkey ignores some distinct states by mistakenly assuming they are equivalent states. To deal with that, we keep the Monkey running and let it explore the remaining states in random order until the time budget is exhausted. This way, the Monkey gets a chance to explore some of those missed states.

**Benefits of using equivalence prediction.** To evaluate the effectiveness of SVM-based state equivalence prediction, we use an SVM Monkey, with prediction enabled, and a Basic Monkey, that does not do any prediction and hence realizes a state is equivalent only after visiting it. We run each app twice, once with each version of the Monkey for 20 minutes.

We measure the Monkey's state exploration performance using a *structural coverage* metric, defined as the fraction of structurally distinct states the Monkey visits, compared with the ground truth found from real users. A Monkey with a structural coverage value of 1 is able to explore all states required to find all structural frauds.

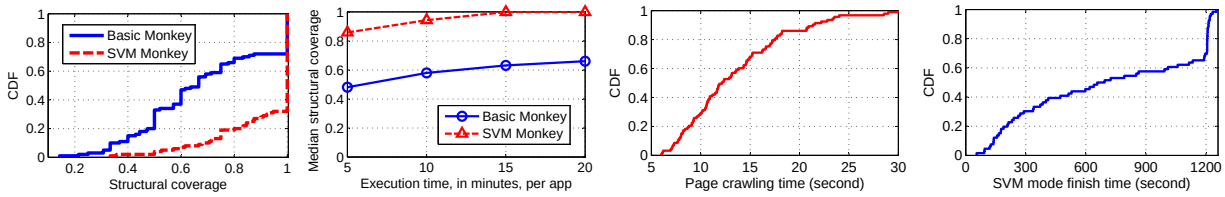
Figure 3(a) shows the structural coverage of the basic and the SVM Monkey, when they are both given 20 minutes to explore each app. In this graph, lower is better: the SVM Monkey achieves less than perfect coverage only for about 30% of the apps, while the basic Monkey achieves less than perfect coverage for over 70% of the apps. Overall, the mean and median coverages of the SVM Monkey are 92% and 100% respectively, and its mean and median coverage improvements are 20.37% and 26.19%, respectively. The SVM Monkey achieves perfect coverage for 71 apps.

Figure 3(b) shows median coverage of the SVM and the basic Monkey as a function of exploration time per app (the graph for mean coverage looks similar, and hence is omitted). As shown, the SVM monkey achieves better coverage for other time limits as well, so for a given target coverage, the SVM Monkey runs much faster than the basic Monkey. For example, the basic Monkey achieves a median coverage of 66% in 20 minutes, while the SVM Monkey achieves a higher median coverage of (86%) in only 5 mins.

The SVM Monkey fails to achieve perfect coverage for 29 of the 100 apps we tried, for several reasons: the Windows Automation Framework occasionally fails to recognize a few controls; some states require app-specific

[7] This also mimics manual app inspection for frauds.

[8] We emphasize that these rates are not directly for fraudulent ads but for predicting whether or not two clickable controls have the same click handler.




(a) Structural coverage in 20 minutes of exploration per app

(b) Median coverage as a function of exploration time

(c) Page crawling speed

(d) App crawling speed

**Figure 3:** *CDF of evaluation result*

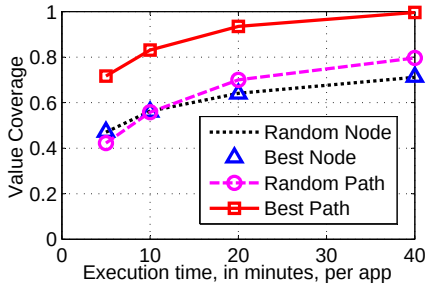



**Figure 4:** *Value coverage as a function of exploration time per app, with various prioritization algorithms.*

text inputs (e.g., a zipcode for location-based search) that our Monkey cannot handle; and some apps just have a large state transition graph. Addressing some of these limitations is left to future work, but overall we are encouraged by the coverage achieved by our optimizations. In the next section, we demonstrate that DECAF can be scaled to a several thousand apps, primarily as a result of these optimizations.

### 6.2 Assessing State Importance

We now evaluate the best node and the best path strategies (Section 4.2.2) with two baselines that do not exploit app usage statistics: *random node*, where the Monkey chooses a random unexplored next state and *random path*, where the Monkey chooses a random path in the state graph.

To assign values and costs of various app states, we conducted a user study to obtain app usage information. We picked five random apps from five different categories and for each app, we asked 16 users to use it for 5 minutes each. We instrumented apps to automatically log which pages users visit, how much time they spend on each page, how long each page needs to load, etc. Based on this log, we assigned each state a value proportional to the total number of times the page is visited and a cost proportional to the average time the page took to load. To compare various strategies, we use the metric *value coverage*, computed as the ratio of the sum of values of visited states and that of all states.

We ran the Monkey with all four path prioritization strategies for total exploration time limits of 5, 10, 20 and 40 minutes. As Figure 4 shows, value coverage increases monotonically with exploration time. More importantly, path-based algorithms outperform node-based algorithms, as they can better exploit global knowledge of values of entire paths; the best-path algorithm significantly outperforms the random-path algorithm (average improvement of 27%), highlighting the value of exploiting app usage. Furthermore, exploring all (content-wise) valuable states of an app can take longer than exploring only structurally distinct states. For the set of apps we use in this experiment, achieving a near-complete coverage takes the Monkey 40 minutes.

### 6.3 Overall Speed and Throughput

Figure 3(c) shows the CDF of time the Monkey needs to explore one app state, measured across all 100 apps. This time includes all the operations required to process the corresponding page: waiting for the page to be completely loaded, extracting the DOM tree of the page, detecting structural fraud in the state, and deciding the next transition. The mean and median times to explore a page is 13.5 and 12.1 sec respectively; a significant component of this time is the additional 5-second delay in detecting page load completion as discussed in Section 4.3. We are currently exploring methods to reduce this delay. Figure 3(d) shows the CDF of time DECAF needs to explore one app. The CDF is computed over 71 apps that the Monkey could finish exploring within 20 minutes. The mean and median time for an app is 11.8 minutes and 11.25 minutes respectively; at this rate, DECAF can scan around 125 apps on a single machine per day.

## 7 Characterizing Ad Fraud

In this section, we characterize the prevalence of ad frauds, compare frauds by type across phone and tablet apps, and explore how ad fraud correlates with app rating, size, and several other factors. To obtain these results, we ran DECAF on 50,000 Windows Phone apps (*phone apps*) and 1,150 Windows 8 apps (*tablet apps*).[9]

[9] For Windows Phone, we consider SilverLight apps only. Some apps especially games are written in XNA that we ignore. Also, the Monkey is not yet sophisticated enough to completely traverse games such as Angry Birds. For Tablet apps, we manually sample simple games from the Games category.

**Table 2:** *Occurrences of various fraud types among all fraudulent apps*

| Fraud type | Phone Apps | Tablet Apps |
|---|---|---|
| Too many (Structural/impression) | 13% | 4% |
| Too small (Structural/impression) | 40% | 54% |
| Outside screen (Structural/impression) | 19% | 4% |
| Hidden (Structural/impression) | 39% | 32% |
| Structural/Click | 11% | 18% |
| Contextual | 2% | 20% |

The Windows 8 App Store prevents programmatic app downloads, so we had to manually download the apps before running them through DECAF, hence the limit of 1,150 on tablet apps. Phone apps are randomly chosen from all SilverLight apps in the app store in April 2013. Microsoft Advertising used DECAF after April 2013 to detect violations in apps and force publishers into compliance, and our results include these apps. Tablet apps were downloaded in September 2013, and were taken from top 100 free apps in 15 different categories.

We did not evaluate state equivalence prediction and state traversal prioritization with these apps. The sheer scale of these apps, and our lack of access to app analytics, made it infeasible to collect the ground truth and usage traces from users required for this evaluation.

**Fraud by Type.** Our DECAF-based analysis reveals that ad fraud is widespread both in phone and in tablet apps. In the samples we have, we discovered more than a thousand phone apps, and more than 50 tablet apps, with at least one instance of ad fraud; the precise numbers are proprietary, and hence omitted.

Table 2 classifies the frauds by type (note that an app may include multiple types of frauds ). Apps exhibit all of the fraud types that DECAF could detect, but to varying degrees; manipulating the sizes of ads, and hiding ads under other controls seem to be the most prevalent both on the phone and tablet. There are, however, interesting differences between the two platforms. Contextual fraud is significantly more prevalent on the tablet, because tablet apps are more content-rich (due to the larger form factor). Ad count violations are more prevalent on the phone, which has a stricter limit (1 ad per screen) compared to the tablet (3 ads per screen).

**Fraud by App Category.** App stores classify apps by category, and Figure 5 depicts distribution of ad fraud frequency across app categories for both phone and tablet apps. In some cases, fraud is equally prevalent across the two platforms, but there are several instances where fraud is more prevalent in one platform than the other. For instance, navigation and entertainment (movie reviews/timings) based apps exhibit more fraud on the phone, likely because they are more frequently used on these devices and publishers focus their efforts on these categories. For a similar reason, tablets show a significantly higher proportion of fraud than phones in the Health, News and Weather, and Sports categories.

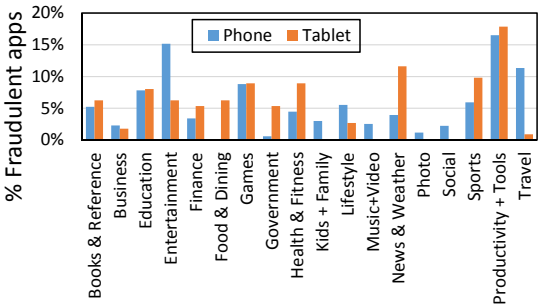



**Figure 5:** *Distribution of fraudulent apps over various categories*

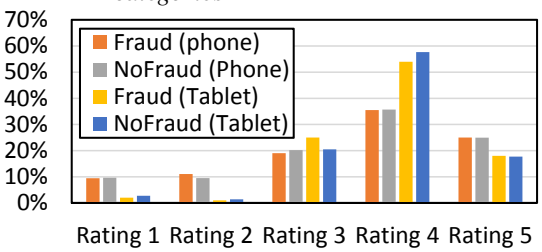



**Figure 6:** *Distribution of ratings for fraudulent and non-fraudulent phone and tablet apps*

**Frauds by rating.** We also explore the prevalence of fraud by two measures of the utility of an app. The first measure is its rating value, rounded to a number from 1-5, and we seek to understand if fraud happens more often at one rating level than at another. Figure 6 plots the frequency of different rating values across both fraudulent and non-fraudulent apps, both for the phone and the tablet. One interesting result is that the distribution of rating values is about the same for fraudulent and non-fraudulent apps; i.e., for a given rating, the proportion of fraudulent and non-fraudulent apps is roughly the same. Fraudulent and non-fraudulent phone apps have average ratings of 1.8 and 1.98. For tablet apps, the average ratings are 3.79 and 3.8, for fraudulent and non-fraudulent apps respectively[10]. If rating is construed as a proxy for utility, this suggests that the prevalence of fraud seems to be independent of the utility of an app.

A complementary aspect of apps is *popularity*. While we do not have direct measures of popularity, Figure 7 plots the cumulative distribution of rating counts (the number of ratings an app has received) for phone apps, which has been shown to be weakly correlated with downloads [22] and can be used as a surrogate for popularity (the graphs look similar for tablet apps as well). This figure suggests that there are small distributional differences in rating counts for fraudulent and non-fraudulent apps; the mean rating counts for phone

[10] Average ratings for tablet apps are higher than that for phone apps because we chose top apps for tablet.

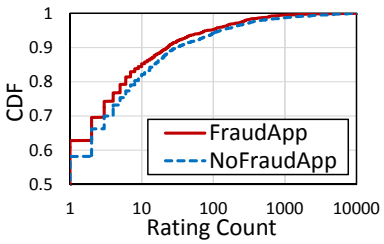



**Figure 7:** *CDF of of rating counts for phone apps*

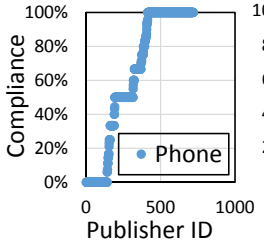



**Figure 8:** *Compliance rate of publishers with multiple apps*

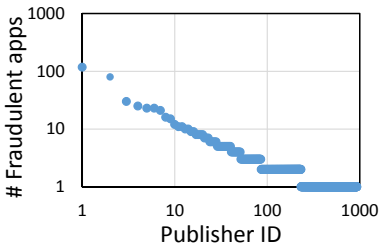




**Figure 9:** *Fraudulent app count per phone app publisher*

apps is 83 and 118 respectively, and for tablet apps is 136 and 157 respectively. However, these differences are too small to make a categorical assertion of the relationship between popularity and fraud behavior.

We had expected to find at least that lower popularity apps, or apps with less utility would more likely exhibit fraud behavior, since they have more incentive to do so. These results are a little inconclusive and either suggest that our intuitions are wrong, or that we need more direct measures of popularity (actual download counts) to establish the relationship.

**The propensity of publishers to commit fraud.** Each app in an app store is developed by a *publisher*. A single publisher may publish more than one app, and we now examine how the instances of fraud are distributed across publishers. Figure 8 plots the compliance rate for phone and tablet apps for publishers who have more than one app in the app store. A compliance rate of 100% means that no frauds were detected across all of the publisher's apps, while a rate of 0% means all the publisher's apps were fraudulent. The rate of compliance is much higher in tablet apps, but that may also be because our sample is much smaller. The phone app compliance may be more reflective of the app ecosystem as a whole: a small number of publishers never comply, but a significant fraction of publishers commit fraud on some of their apps. More interesting, the distribution of the number of frauds across publishers who commit fraud exhibits a heavy tail (Figure 9): a small number of publishers are responsible for most of the fraud.

**Takeaways.** These measurement results are actionable in the following way. Given the scale of the problem, an ad network is often interested in selectively investing resources in fraud detection, and taken together, our results suggest ways in which the ad network should, and should not, invest resources wisely. The analysis of fraud prevalence by type suggests that ad networks could preferentially devote resources to different types of fraud on different platforms; for instance, the ad count and contextual frauds constitute the lion's share of frauds on tablets, so an ad network may optimize fraud detection throughput by running only these checkers. Similarly, the analysis of fraud by categories suggests categories of apps to which ad networks can devote more resources, and points out that these categories may depend on the platforms. The analysis also points out that ad networks should not attempt to distinguish by rating or rating count. Finally, and perhaps most interesting, the distribution of fraud counts by publisher suggests that it may be possible to obtain significant returns on investment by examining apps from a small set of publishers.

## 8 Related Work

DECAF is inspired by prior work on app automation and ad fraud detection.

**App Automation.** Today, mobile platforms like Android provide UI automation tools [3, 2] to test mobile apps. But these tools rely on the developer to provide automation scripts, and do not provide any visibility into the app runtime so are inefficient and cannot be easily used for detecting ad frauds.

Recent research efforts have built upon these tools to provide full app automation, but their focus has been on different applications: automated testing [37, 53, 18, 19, 28, 32, 45] and automated privacy and security detection [26, 29, 38, 49]. Automated testing efforts evaluate their system only on a handful of apps and many of their UI automation techniques are tuned to those apps. Systems that look for privacy and security violations execute on a large collection of apps but they only use basic UI automation techniques. Closest to our work is AMC [35], which uses automated app navigation to verify UI properties for vehicular Android apps, but reported exploration times of several hours per app and has been evaluated on 12 apps. In contrast to all of these, DECAF is designed for performance and scale to automatically discover ad frauds violations in several thousand apps. Symbolic and concolic execution [25, 21] are alternative techniques for verifying properties of code, and have been applied to mobile apps [19, 45]. For discovering UI properties, UI graph traversal is a more natural technique than concolic execution, but it may be possible to detect ad fraud using concolic execution, which we have left to future work.

Tangentially relevant is work on crowdsourcing GUI testing, and automated execution frameworks for Ajax Web apps (Crawljax [41], AjaxTracker [36] and ATUSA [40]). DECAF can use crowdsourcing to obtain state importance, and Ajax-based frameworks don't deal with mobile app specific constraints.

**Ad Fraud.** Existing works on ad fraud mainly focus on the click-spam behaviors, *characterizing* the features of click-spam, either targeting specific attacks [17, 20, 44, 46], or taking a broader view [23]. Some work has examined other elements of the click-spam ecosystem: the quality of purchased traffic [52, 55], and the spam profit model [34, 39]. Very little work exists in exploring click-spam in mobile apps. From the controlled experiment, authors in [23] observed that around one third of the mobile ad clicks may constitute click-spam. A contemporaneous paper [27] claimed that they are not aware of any mobile malware in the wild that performs advertising click fraud. Unlike these, DECAF focuses on detecting violations to ad network terms and conditions, and even before potentially fraudulent clicks have been generated.

With regard to detection, most existing works focus on bot-driven click spam, either by analyzing search engine query logs to identify outliers in query distributions [54], characterizing networking traffic to infer coalitions made by a group of bot-driven fraudsters [42, 43], or authenticating normal user clicks to filter out bot-driven clicks [31, 33, 51]. A recent work, Viceroi [24], designed a more general framework that is possible to detect not only bot-driven spam, but also some non-bot driven ones (like search-hijacking). DECAF is different from this body of work and focuses on user-based ad fraud in the mobile app setting rather than the click-spam fraud in the browser setting – to the best of our knowledge, ours is the first work to detect ad fraud in mobile apps.

## 9 Conclusion

DECAF is a system for detecting placement fraud in mobile app advertisements. It efficiently explores the UI state transition graph of mobile apps in order to detect violations of terms and conditions laid down by ad networks. DECAF has been used by Microsoft Advertising to detect ad fraud and our study of several thousand apps in the wild reveals interesting variability in the prevalence of fraud by type, category, and publisher. In the future, we plan to explore methods to increase the coverage of DECAF's Monkey, expand the suite of frauds that it is capable of detecting, evaluate other metrics for determining state importance, and explore attacks designed to evade and DECAF and develop countermeasures for these attacks.

**Acknowledgements.** We thank the anonymous referees and our shepherd Aruna Balasubramanian for their comments. We are grateful to Michael Albrecht, Rich Chapler and Navneet Raja from Microsoft for their feedback on early design of DECAF.

## References

[1] AdMob Publisher Guidelines and Policies. http://support.google.com/admob/answer/1307237?hl=en&ref_topic=1307235.

[2] Android Monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[3] Android UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html.

[4] Bots Mobilize. http://www.dmnews.com/bots-mobilize/article/291566/.

[5] Flurry. http://www.flurry.com/.

[6] Google Admob. http://www.google.com/ads/admob/.

[7] Google Admob: What's the Difference Between Estimated and Finalized Earnings? http://support.google.com/adsense/answer/168408/.

[8] iAd App Network. http://developer.apple.com/support/appstore/iad-app-network/.

[9] Microsoft Advertising. http://advertising.microsoft.com/en-us/splitter.

[10] Microsoft Advertising: Build your business. http://advertising.microsoft.com/en-us/splitter.

[11] Microsoft pubCenter Publisher Terms and Conditions. http://pubcenter.microsoft.com/StaticHTML/TC/TC_en.html.

[12] The Truth About Mobile Click Fraud. http://www.imgrind.com/the-truth-about-mobile-click-fraud/.

[13] Up To 40% Of Mobile Ad Clicks May Be Accidents Or Fraud? http://www.mediapost.com/publications/article/182029/up-to-40-of-mobile-ad-clicks-may-be-accidents-or.html#axzz2ed63eE9q.

[14] Windows Hooks. http://msdn.microsoft.com/en-us/library/windows/desktop/ms632589(v=vs.85).aspx.

[15] Windows Input Simulator. http://inputsimulator.codeplex.com/.

[16] Windows Performance Counters. http://msdn.microsoft.com/en-us/library/windows/desktop/aa373083(v=vs.85).aspx.

[17] S. Alrwais, A. Gerber, C. Dunn, O. Spatscheck, M. Gupta, and E. Osterweil. Dissecting Ghost Clicks: Ad Fraud Via Misdirected Human Clicks. In *ACSAC*, 2012.

[18] D. Amalfitano, A. Fasolino, S. Carmine, A. Memon, and P. Tramontana. Using GUI Ripping for Automated Testing of Android Applications. In *IEEE/ACM ASE*, 2012.

[19] S. Anand, M. Naik, M. Harrold, and H. Yang. Automated Concolic Testing of Smartphone Apps. In *ACM FSE*, 2012.

[20] T. Blizard and N. Livic. Click-fraud monetizing malware: A survey and case study. In *MALWARE*, 2012.

[21] S. Bugrara and D. Engler. Redundant State Detection for Dynamic Symbolic Execution. In *USENIX ATC*, 2013.

[22] P. Chia, Y. Yamamoto, and N. Asokan. Is this App Safe? A Large Scale Study on Application Permissions and Risk Signals. In *WWW*, 2012.

[23] V. Dave, S. Guha, and Y. Zhang. Measuring and Fingerprinting Click-Spam in Ad Networks. In *ACM SIGCOMM*, 2012.

[24] V. Dave, S. Guha, and Y. Zhang. ViceROI: Catching Click-Spam in Search Ad Networks. In *ACM CCS*, 2013.

[25] C. Cadar D. Dunbar and D. Engler. Klee: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *USENIX OSDI*, 2008.

[26] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: an Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX OSDI*, 2010.

[27] A. Felt, Porter, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A Survey of Mobile Malware in the Wild. In *ACM SPSM*, 2011.

[28] S. Ganov, C. Killmar, S. Khurshid, and D. Perry. Event Listener Analysis and Symbolic Execution for Testing GUI Applications. In *ICFEM*, 2009.

[29] P. Gilbert, B. Chun, L. Cox, and J. Jung. Vision: Automated Security Validation of Mobile apps at App Markets. In *MCS*, 2011.

[30] M. Grace, W. Zhou, X. Jiang, and A. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *ACM WiSec*, 2012.

[31] H. Haddadi. Fighting Online Click-Fraud Using Bluff Ads. *ACM Computer Communication Review*, 40(2):21–25, 2010.

[32] C. Hu and I. Neamtiu. Automating GUI Testing for Android Applications. In *AST*, 2011.

[33] A. Juels, S. Stamm, and M. Jakobsson. Combating Click Fraud via Premium Clicks. In *USENIX Security*, 2007.

[34] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. Voelker, V. Paxson, and S. Savage. Spamalytics: An Empirical Analysis of Spam Marketing Conversion. In *ACM CCS*, 2008.

[35] K. Lee, J. Flinn, T. Giuli, B. Noble, and C. Peplin. AMC: Verifying User Interface Properties for Vehicular Applications. In *ACM MobiSys*, 2013.

[36] M. Lee, R. Kompella, and S. Singh. Ajaxtracker: Active Measurement System for High-fidelity Characterization of AJAX Applications. In *USENIX WebApps*, 2010.

[37] A. MacHiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In *ACM FSE*, 2013.

[38] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. A Whitebox Approach for Automated Security Testing of Android Applications on the Cloud. In *AST*, 2012.

[39] D. McCoy, A. Pitsillidis, G. Jordan, N. Weaver, C. Kreibich, B. Krebs, G. Voelker, S. Savage, and K. Levchenko. PharmaLeaks: Understanding the Business of Online Pharmaceutical Affiliate Programs. In *USENIX Security*, 2012.

[40] A. Mesbah and A. van Deursen. Invariant-based Automatic Testing of AJAX User Interfaces. In *ICSE*, 2009.

[41] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling AJAX-based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web*, 6(1):1–30, 2012.

[42] A. Metwally, D. Agrawal, and A. El Abbadi. DETECTIVES: DETEcting Coalition hiT Inflation attacks in adVertising nEtworks Streams. In *WWW*, 2007.

[43] A. Metwally, F. Emekci, D. Agrawal, and A. El Abbadi. SLEUTH: Single-pubLisher attack dEtection Using correlaTion Hunting. In *PVLDB*, 2008.

[44] B. Miller, P. Pearce, C. Grier, C. Kreibich, and V. Paxson. What's Clicking What? Techniques and Innovations of Today's Clickbots. In *IEEE DIMVA*, 2011.

[45] N. Mirzaei, S. Malek, C. Pasareanu, N. Esfahani, and R. Mahmood. Testing Android Apps through Symbolic Execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[46] T. Moore, N. Leontiadis, and N. Christin. Fashion Crimes: Trending-Term Exploitation on the Web. In *ACM CCS*, 2011.

[47] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: a Pragmatic Approach to Model Checking Real Code. In *USENIX OSDI*, 2002.

[48] Suman Nath, Felix Lin, Lenin Ravindranath, and Jitu Padhye. SmartAds: Bringing Contextual Ads to Mobile Apps. In *ACM MobiSys*, 2013.

[49] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic Security Analysis of Smartphone Applications. In *ACM CODASPY*, 2013.

[50] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *USENIX OSDI*, 2012.

[51] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. Wang, and C. Cowan. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *IEEE S & P*, 2012.

[52] K. Springborn, , and P. Barford. Impression Fraud in Online Advertising via Pay-Per-View Networks. In *USENIX Security*, 2013.

[53] W. Yang, M. Prasad, and T. Xie. A Grey-box Approach for Automated GUI-model Generation of Mobile Applications. In *FASE*, 2013.

[54] F. Yu, Y. Xie, and Q. Ke. SBotMiner: Large Scale Search Bot Detection. In *ACM WSDM*, 2010.

[55] Q. Zhang, T. Ristenpart, S. Savage, and G. Voelker. Got Traffic? An Evaluation of Click Traffic Providers. In *WebQuality*, 2011.