Understanding and Detecting Overlay-based Android Malware at Market Scales

Yuxuan Yan[1,2], Zhenhua Li[1*], Qi Alfred Chen[3], Christo Wilson[4]
Tianyin Xu[5], Ennan Zhai[6], Yong Li[1], Yunhao Liu[1,7]
[1]Tsinghua University [2]Tencent Mobile Security [3]University of California, Irvine
[4]Northeastern University [5]UIUC [6]Alibaba Group [7]Michigan State University

ABSTRACT

As a key UI feature of Android, *overlay* enables one app to draw over other apps by creating an extra `View` layer on top of the host `View`. While greatly facilitating user interactions with multiple apps at the same time, it is often exploited by malicious apps (malware) to attack users. To combat this threat, prior countermeasures concentrate on restricting the capabilities of overlays at the OS level, while barely seeing adoption by Android due to the concern of sacrificing overlays' usability. To address this dilemma, a more pragmatic approach is to enable the *early detection* of overlay-based malware at the app market level during the app review process, so that all the capabilities of overlays can stay unchanged. Unfortunately, little has been known about the feasibility and effectiveness of this approach for lack of understanding of malicious overlays in the wild.

To fill this gap, in this paper we perform the first large-scale comparative study of overlay characteristics in benign and malicious apps using static and dynamic analyses. Our results reveal a set of suspicious overlay properties strongly correlated with the malice of apps, including several novel features. Guided by the study insights, we build OverlayChecker, a system that is able to automatically detect overlay-based malware at market scales. OverlayChecker has been adopted by one of the world's largest Android app stores to check around 10K newly submitted apps per day. It can efficiently (within 2 minutes per app) detect nearly all (96%) overlay-based malware using a single commodity server.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; • **Networks → Mobile and wireless security**.

ACM Reference Format:
Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. 2019. Understanding and Detecting Overlay-based Android Malware at Market Scales. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19), June 17–21, 2019, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307334.3326094

* Corresponding author.

MobiSys '19, June 17–21, 2019, Seoul, Republic of Korea

ACM ISBN 978-1-4503-6661-8/19/06…$15.00
https://doi.org/10.1145/3307334.3326094

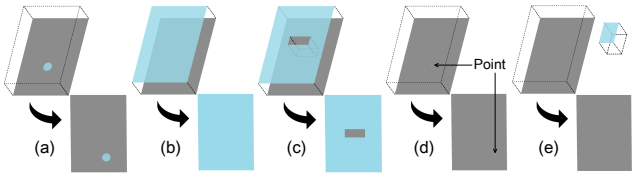

Figure 1: Five common forms of overlays: (a) float, (b) cover, (c) hollow out, (d) single point, (e) offscreen. The upper row plots the underlying host `View` and the overlay lying on top of it, while the lower row plots the user-perceived view in each case. Benign apps typically use cases (a)(b), while malicious apps use all cases (a)–(e). Note that the overlay (even in cases (d) and (e)) can receive all user interaction events with the host `View` if it explicitly specifies certain flags.

1 INTRODUCTION

Overlay is a user interface (UI) feature supported by Android since version 1.0. It enables a mobile app to draw over other apps by creating an extra `View` layer on top of the host `View`, as illustrated in Figure 1. The rationale behind Android's overlay feature is to improve the users' experience when they are interacting with multiple apps at the same time. Indeed, with the limited sizes of smartphone screens, squeezing the UIs of multiple apps on a small screen would significantly impair usability (although Android has launched Multi-Window support for displaying multiple apps in a split-screen mode since version 7.0, Multi-Window is seldom used by today's smartphone users). Overlays have been widely adopted by mobile apps installed on hundreds of millions of mobile devices, such as Facebook, Uber, Messenger, Skype, *etc.* We observe 35.4% of the top-500 popular apps in Google Play Store use overlays.

Unfortunately, the overlay feature is often exploited by malicious apps (or says *malware*) to attack users [3, 6, 9, 19, 22, 25, 31, 37, 40]. Since overlays can intercept user input that is intended for the underlying host `View`, one common attack is to capture sensitive user actions or data on the fly through deceptive overlays, as illustrated in Figures 1(c–e). To make matters worse, a recent study [10] demonstrated that the UI feedback loop can be completely compromised and controlled through the "cloak and dagger" attack that exploits only two Android permissions: `SYSTEM_ALERT_WINDOW` and `BIND_ACCESSIBILITY_SERVICE`, where the former is the permission that allows an app to create overlays on top of all other apps. Surprisingly, `SYSTEM_ALERT_WINDOW` is automatically granted to apps installed from Google Play Store [10].

Given the severity and prevalence of overlay-based attacks, several countermeasures have been proposed to restrict the capabilities of overlays at the OS level [3, 8, 10]. However, these solutions barely

see any adoptions by Android due to the concern of sacrificing usability. For example, the "cloak and dagger" attack can be mitigated by modifying Android to limit the permissions granted to apps and thus restrict their capabilities. Nevertheless, since "limiting those services would render the device unusable," the Android team merely updated the developer documentation without further action, even with full awareness of the security threat [10].

To address this dilemma, a more pragmatic approach is to enable the *early detection* of overlay-based malware at the app market level during the app review process (before the apps are released and installed on user devices), so that all the capabilities of overlays can be retained. Unfortunately, little has been known about the feasibility and effectiveness of this approach due to the lack of understanding of malicious overlays in the wild. Systematically gaining such understanding is rather challenging, since there is no available large-scale dataset of malicious overlays, and it is prohibitively difficult to manually examine every `View` for a large set of apps to collect malicious overlays. Still worse, few regulations and usage references are available for overlay usage at present, making it difficult to define benign and malicious overlays.

Understanding overlay-based malware. To overcome this impasse, our key insight is that we can figure out the characteristics of malicious overlays from *the overlay behavior differences between benign and malicious apps*. Using the ground-truth malware data from one of the world's largest Android app stores, *i.e.,* Tencent App Market (https://sj.qq.com/myapp/) or T-Market, we perform the first large-scale comparative study of the overlay behavior between benign and malicious apps, with both static and dynamic analyses. Compared with static features that can be directly extracted from Android APK files, many important features of overlays are dynamically determined at app runtime. To extract these features, we dynamically exercise apps using the *Monkey* UI exerciser [34].

Our results reveal a set of suspicious overlay properties strongly correlated with the *malice* of apps (an app is labeled to be either "Benign" or "Malicious"). Our key findings are listed as follows: (1) Overlays are used by 51% of malicious apps, and they intentionally make their overlays difficult to detect. (2) `Type`, `Flag`, and `Format` are the three features with the strongest correlations with an app's malice, *e.g.,* 84% of the apps that use `TYPE_SYSTEM_ERROR` overlays are malicious. (3) Overlays' visual coverage exhibits distinct distributions between malicious and benign apps. (4) A programmatically visible overlay can be visually invisible to users, and this fact is often exploited by malicious apps.

In particular, we notice that although there are 52 overlay-related features defined in the Android SDK (see Table 1), they fail to capture several important aspects of overlays in practice. For example, no existing feature corresponds to whether an overlay is actually visible to the user when it is being rendered (*e.g.,* Figures 1(d) and 1(e)). To address this limitation in the Android SDK, we introduce four novel features into our study (which can improve the detection accuracy of our system described later).

Detecting overlay-based malware. Leveraging above insights, we develop a system called OverlayChecker to enable market-scale early detection of overlay-based malicious apps at the app submission time. Such detection capability is highly attractive since it does not require OS-level changes and thus is able to address the tension between usability and security of previously-proposed solutions [3, 8, 10]. Based on the characterization results of malicious overlays, OverlayChecker collects 56 static and dynamic features from each app, and uses them to detect malicious overlay behavior via a *random forest* [4] machine learning model. Our model is trained using large-scale, ground-truth data provided by T-Market.

To make OverlayChecker usable at market scales, it must be efficient enough to analyze a great number of app submissions per day. For this purpose, we built a lightweight Android emulator to accelerate our system, that directly runs the Android OS and apps on x86 architecture, coupled with *dynamic binary translation* [17] to support apps that use Android's native APIs. This custom infrastructure enables OverlayChecker to analyze apps on a commodity x86 server much more quickly, achieving a speedup of 8–10× compared to using the default emulator in Android SDK.

Real-world performance and robustness. OverlayChecker has been integrated into T-Market as a part of the app review process since Jan. 2018. It checks around 10K app submissions per day using a single commodity server. The per-app analysis time is less than two minutes, and OverlayChecker achieves 96% detection precision and 96% recall. Through the lens of malicious overlays, we find that OverlayChecker is especially effective (over 90% accuracy) in detecting certain types of malicious apps, *e.g.,* ransomware, adware, porn-fraud and SMS-fraud apps, due to their heavy reliance on overlays to launch their intended attacks.

Furthermore, we applied OverlayChecker to 10K random apps in Google Play Store and detected 20 previously unknown apps with malicious overlays. Although these apps were removed from the Play Store within days, these incidents demonstrate that despite Google's existing app-security checks, early detection is still necessary to prevent malware from being made available to users.

We present an in-depth analysis of our random forest model to investigate whether attackers will be able to avoid OverlayChecker by adapting their malware's behavior. By interpreting our model, we show that the behaviors of benign and malicious overlays are sufficiently different, which makes it rather difficult for a malicious overlay to avoid OverlayChecker: most existing malicious overlay strategies are entirely precluded, and attackers are left with a significantly weaker range of attack capabilities.

Summary. We make the following contributions:

- We conduct the first large-scale comparative study of overlay characteristics in benign and malicious apps using static and dynamic analyses. Our results reveal a set of suspicious overlay properties that are strongly correlated with the malice of apps, including several novel features.
- Driven by these insights, we build a market-scale early detection system, OverlayChecker, to address overlay-based attacks in the Android ecosystem. It uses a machine learning model that incorporates static and dynamic app features. We developed custom emulation and dynamic execution infrastructure to increase its detection speed and resilience against stealthy malware.
- OverlayChecker is currently deployed on one of the world's largest Android app stores, where it analyzes around 10K apps per day on a single commodity server. We also demonstrate OverlayChecker's applicability to Google Play Store.

2 BACKGROUND

This section introduces the basics of Android overlay, and the common practices of fighting against malicious apps in app stores.

2.1 Android Overlay Basics

In the Android UI framework, an overlay is a special feature enabling one app to create an extra `View` layer that sits atop the host `View`[1]. Different from the host `View` which is almost always in a rectangular shape occupying the full screen of the user device, an overlay possesses a great deal of freedom in terms of *shape*, *area*, and *location*. As shown in Figure 1, an overlay can be (a) a small-area circle floating atop the host `View` at an arbitrary location, (b) a full-screen rectangle completely covering the host `View`, (c) a hollow-out rectangle partially covering the host `View`, (d) a single point that is rather difficult to notice, or (e) a rectangle outside the screen that cannot be noticed by users. All overlays are able to intercept user input that is intended for the underlying host `View` if certain flags are specified. In summary, overlay is a powerful UI feature that allows one app to display something on top of other apps, which can be used to intercept sensitive user input, or alter the user's perceptions of which app is currently active on the screen.

Each overlay's appearance is defined by a number of `Layout` and `View` parameters (an overlay is an object inheriting the `View` class) listed in Table 1. Among the appearance parameters, the `X`, `Y`, `Width`, and `Height` geometry parameters are intuitive. `Gravity` decides the placement of an overlay within a larger UI container. `isOpaque` and `Alpha` together qualify and quantify the transparency. `Background` specifies an overlay's background image or color. `Format` defines the desired bitmap format like `RGBA_8888` (meaning the overlay can be of any transparency), `TRANSPARENT`, and `TRANSLUCENT`.

The capability of an overlay is derived from the specifications of `Type`, `Root`, `ScreenShot`, and `Flags`. When an Android app intends to use the overlay feature, it typically requests the `SYSTEM_ALERT_WINDOW` permission. Despite the proscription from official Android documentation, "*very few apps should use this permission; these windows are intended for system-level interaction with the user*", this permission is still requested by 46% of the benign apps and 75% of the malicious apps hosted in T-Market. This should be the rationale behind Google's decision to automatically grant `SYSTEM_ALERT_WINDOW` to all apps installed from Google Play Store. In comparison, prior to Oct. 2015 Android apps had to explicitly request this permission [28, 29]. More in detail, `SYSTEM_ALERT_WINDOW` overlays have 5 `Types` of display priorities, among which `TYPE_SYSTEM_ERROR` has the highest priority—a `TYPE_SYSTEM_ERROR` overlay can even appear on top of the lock screen interface. In contrast, `TYPE_TOAST` overlays have limited capabilities (*e.g.,* they cannot appear on top of the lock screen interface); as a result, they are less commonly used by app developers. In addition, `Root` and `ScreenShot` define the functionality of an overlay. There are also 31 `Flags` specifying various aspects of overlay behavior, *e.g.,* if `FLAG_WATCH_OUTSIDE_TOUCH` is set, an overlay can receive all the UI events outside its coverage area.

[1]In contrast, overlay provided by the iOS UI framework only enables an app to create an extra `View` layer *within the same app* rather than across apps. Hence, although iOS overlays are unable to facilitate a user's interactions with multiple apps at the same time, they cannot be exploited by an app to launch security attacks on other apps.

Category	Parameters
Appearance	`X`, `Y`, `Width`, `Height`, `Gravity`, `horizontalMargin`, `horizontalWeight`, `verticalMargin`, `verticalWeight`, `screenOrientation`, `isOpaque`, `Alpha`, `Background`, `Format`, `dimAmount`, `screenBrightness`, *VisualCoverage*, *isReallyVisible*
Priority	`Type`
Functionality	`Root`, `ScreenShot`
Quantity	*ActivityCoverage*, *NumOfOverlays*
Flags	`FLAG_FULLSCREEN`, `FLAG_LAYOUT_IN_SCREEN`, `FLAG_ALLOW_LOCK_WHILE_SCREEN_ON`, `FLAG_NOT_FOCUSABLE`, `FLAG_NOT_TOUCH_MODAL`, `FLAG_WATCH_OUTSIDE_TOUCH`, ····· (31 in total)
Static	`BIND_ACCESSIBILITY_SERVICE`, `PACKAGE_USAGE_STATS`

Table 1: Android apps' `Layout` and `View` parameters that determine the overlay behavior. The calculated novel features we design for detecting malicious overlays are in *italic*.

Finally, there are two *static* properties at the app level that can amplify the capability of overlays: `BIND_ACCESSI-BILITY_SERVICE` and `PACKAGE_USAGE_STATS`. The former is used to assist Android users with disabilities, and the latter allows an app to collect the usage statistics of other apps. Although apps must explicitly request permission to use these capabilities, in practice a malicious app can lure users to unknowingly grant them, *e.g.,* by abusing the capability from the `SYSTEM_ALERT_WINDOW` permission [19].

2.2 Security Practices of App Stores

App stores, such as Google Play Store, Apple App Store, and Amazon Appstore, are the de facto platform of mobile app distribution. As of Feb. 2018, there are over 3M Android apps released on Google Play Store. T-Market, the app store we collaborate with in this work, has released over 6M apps since its launch in 2012, with over 30M APKs being downloaded by 20M users every day. To protect users from downloading malicious apps, T-Market conducts a series of app review procedures to examine ∼10K newly submitted apps (including both new and updated apps) from developers on a daily basis. This section takes T-Market as a representative case study to reveal the practices of today's app stores for identifying malicious apps, as well as their utility to OverlayChecker.

To accurately determine the malice of hosted apps, T-Market conducts a sophisticated app review process consisting of fingerprint-based antivirus checking, API inspection, and manual examination. Antivirus checking inspects apps against virus fingerprints [41] from antivirus service companies including Symantec, Kaspersky, Norton, McAfee, and so on. API inspection identifies malicious apps by scanning what Android APIs are invoked in their code; its heuristic lies in that certain patterns (combinations and orders) of API calls imply serious security threats [1]. For those apps whose malice cannot be determined through antivirus checking and API inspection, T-Market assigns security experts to manually examine them with very high precision.

T-Market maintains a massive database of malicious apps captured during the app review process or reported by the users in the field. The dataset is a precious resource in understanding the characteristics of overlays used by malicious apps (refer to §3). Furthermore, the malicious apps recorded in the database, together with the other benign apps, form a large labeled training set, based on which supervised learning can be applied to reveal the key overlay properties associated with malicious apps (refer to §4).

Similar to other app stores, T-Market maintains the category labels of each app (*e.g.,* social networking, shopping, entertainment, and education). These labels are predefined by T-Market and selected by app developers. For malicious apps recorded in the database, T-Market has another set of labels such as ransomware, adware, and SMS-frauds. These labels offer us opportunities to understand the motivations and use cases of overlay-based attacks.

3 UNDERSTANDING OVERLAY BEHAVIOR

This section presents our analysis of overlay behavior of malicious and benign apps. Our objective is to identify features that highly correlate with malicious overlay behavior, which we can then operationalize for early detection of malware. We first introduce the app dataset (§3.1), and then describe our methodology for extracting overlay features (§3.2). Next, we present general statistics (§3.3) and the key overlay features associated with malicious overlay behavior (§3.4). Finally, we summarize our study results in §3.5.

3.1 App Dataset

Being one of the world's largest Android app stores, T-Market has identified and labeled over 2.4M malicious apps and over 6M benign apps since May 2015. Our raw dataset contains all the new and updated apps submitted to T-Market during Jan.–May. 2017. After removing redundant apps (whose APKs have the same MD5 hash value), we are left with 450K unique apps as our final dataset. Here we mainly study the recent submissions (relative to the time we started the project, *i.e.,* Jun. 2017), because many high-profile overlay attacks exploit new features in Android 6.0 [10] and the entire dataset maintained by T-Market includes a huge number of apps with obsoleted overlay usage. We release our dataset used in the study at https://overlaychecker.github.io/.

For every app in the dataset, T-Market provides not only its APK file but also its malice and category labels. Nearly one third (31.7%) of the apps are labelled malicious and thus are quarantined in T-Market's database. Note that the ratio of malicious apps is relatively high because these are the submitted apps *before the app review process*, instead of those released to users. As introduced in §2.2, since T-Market adopts a rather sophisticated and effective app review process, we believe the false positive rate in this labelling is statistically insignificant and thus has negligible impact on our subsequent analysis and system design. In detail, each of the involved antivirus services claims that the false positive rate is less than 5%. When they all label an app as malicious, T-Market takes the app as malicious; if their labels are inconsistent, T-Market manually examines the malice of the app. Assuming that all the antivirus engines are independent of each other (which may overestimate the accuracy of our ground-truth dataset), the precision of our training set should exceed $1 - (1 - 95\%)^4 = 99.9994\%$.

3.2 Overlay Feature Extraction

The first step towards understanding overlay behavior is to extract the features of overlays in each app. Specifically, OverlayChecker identifies overlay-based apps (*i.e.,* apps that actually use one or more overlays) dynamically at the run time. In Android, all overlays must be created by invoking the `addView` API of the `WindowManagerGlobal` class, so we can identify overlay-based apps by checking whether an app has created a `System Window View` (whose `Type` ranges from 2000 to 2099). At the same time, we can extract our concerned dynamic features because they are also attached when the `addView` API is invoked. For each overlay, there are *static* and *dynamic* features requiring different extraction methods. Static features can be directly extracted from an app's APK file. In contrast, dynamic features only exist when the app is executed on user devices [15], and their number is much larger than that of static features. In addition to those original features defined in the Android SDK (refer to Table 1), we design four novel features: `ActivityCoverage`, `NumOfOverlays`, `VisualCoverage`, and `isReallyVisible`. We will detail these features in the context of their use cases.

3.2.1 App Emulation. To extract dynamic features of overlays, we explore each app using the *Monkey* UI exerciser [34] that can generate UI event streams at both application and system levels. Whenever *Monkey* hits an overlay object, it records 54 dynamic features (refer to Table 1) for later analysis. We execute apps and *Monkey* on Android emulators deployed on a commodity x86 server. However, we do not use the default Google's Android device emulator. Since the default emulator is based on full-system emulation built on top of QEMU, its performance is limited and cannot achieve our goal of emulating a large number of apps at scale.

Instead, we built a lightweight emulator that directly runs the Android OS and apps on x86 architecture. First, as for the Android OS we use Android-x86, an open-source x86 porting of the original ARM-based Android OS. Also, to support apps that use Android's native APIs, we implement *dynamic binary translation* [2, 16] based on Intel Houdini [17] to translate ARM instructions into x86 instructions (most dynamic libraries in Android are based on the ARM ISA instead of x86). Moreover, we notice that a very small portion ($< 0.9\%$) of apps cannot run successfully on the abovementioned lightweight Android emulator. To address this problem, we make extra customization in the production system – for those incompatible apps, we roll back (from our lightweight emulator) to the default Google's Android emulator to successfully execute them.

Further, for parallelism we run multiple concurrent emulators on a x86 server, with each bound to one CPU core. Specifically, for a commodity x86 server (HP ProLiant DL-380) with 5×4-core Xeon CPU @ 2.50 GHz and 32-GB memory, we run 16 emulations on 16 cores concurrently and the remaining 4 cores are employed for scheduling, monitoring, and logging. Our lightweight emulator is much more efficient than the default emulator in the Android SDK—typically it can reduce the emulation overhead by 8–10×. Within each emulator, generating and executing 100, 1K, 10K and 100K *Monkey* events take 2, 22, 220 and 2220 seconds on average.

Some malicious apps may attempt to recognize whether they are running on emulators so as to hide their malicious activities. They typically check static configurations of the system, dynamic time intervals of user actions, and sensor data of the device to

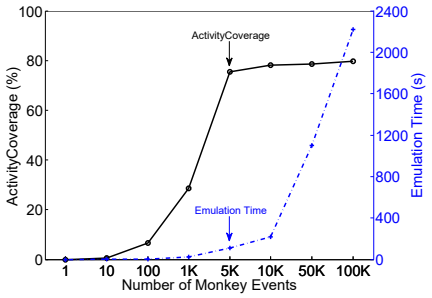

Figure 2: Relationship between the emulation time, `ActivityCoverage`, and number of *Monkey* events.

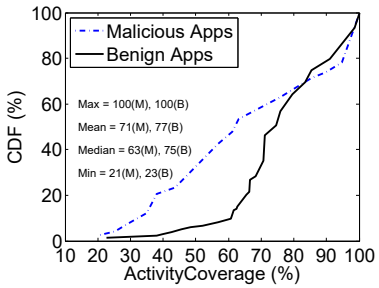

Figure 3: `ActivityCoverage` for all malicious (M) and benign (B) apps when 5K *Monkey* events are executed.

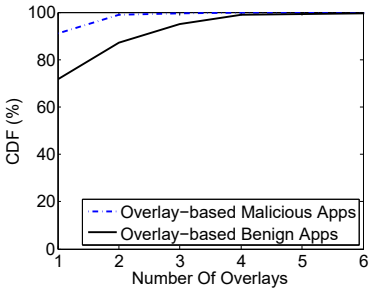

Figure 4: CDF of `NumOfOverlays` for all overlay-based malicious and benign apps.

identify emulators [7, 12, 26]. To prevent adversarial detection, we made three improvements to our emulators to make them behave more consistently with real devices and users. First, we alter the default configurations of the emulators, including their identity (IMEI and IMSI), network properties, and other properties defined in the `build.prop` file such as `PRODUCT` and `MODEL` types [26]. Second, we adjust the execution parameters of *Monkey* to make its generated UI events appear more realistic [34]. For example, we dynamically tune the `throttle` parameter within a reasonable range, so that the occurrence time intervals of the UI events basically comply with real-world cases. Third, we replay traces of sensor data (concerning the acceleration, rotation vector, proximity, orientation, and so forth) collected from a number of real smartphones on our emulators to improve authenticity [7, 12, 33].

3.2.2 UI Exploration. Detecting malicious overlay behavior requires a high UI coverage to examine as many overlays as possible. Initially, we used the `Activity` coverage as the main metric of UI coverage, as each Android app specifies its *possible* (but not necessarily used) `Activity` objects in its `AndroidManifest.xml` configuration file. However, this metric is overly pessimistic, since it counts `Activities` that are not actually referenced by the code. To figure out which `Activities` are *actually* used by an Android app, we write a script to automatically scan the code of its APK file. The scanning results show that for an average app, 88% of its specified `Activities` are actually referenced. Therefore, we define a more accurate metric, `ActivityCoverage`, to quantify the UI coverage. For an app, the `ActivityCoverage` is the ratio of detected `Activities` during emulation over its actually used `Activities`.

Over the apps in our dataset, we observe that generating and executing 100K *Monkey* events generally achieves the highest `ActivityCoverage`. Consequently, it requires around 2220 seconds (= 37 minutes) on average to analyze the overlay behavior of every single app[2]. However, this emulation time is unacceptable to both app stores and developers in practice. From the perspective of app stores, the resulting computation overhead is expensive, *e.g.,* T-Market would need to employ hundreds of servers to handle its current workload. From the perspective of app developers, after submitting an app to the store, they would have to wait for nearly 40 minutes before the app is released to users. This could significantly impair the prosperity of T-Market, given that many rival app stores allow a newly submitted app to be released to users instantly.

To address this problem, we carefully balance effectiveness in terms of `ActivityCoverage` and efficiency in terms of emulation time [20]. Figure 2 shows the `ActivityCoverage` achieved with increasing running time of *Monkey*. We can see that as the emulation time increases, the average `ActivityCoverage` quickly grows until it is close to 76%; after that, its growth is flat. Even spending 20× more time to generate 100K *Monkey* events can only increase the `ActivityCoverage` to 78% on average. Therefore, we choose to run the emulation for ∼100 seconds (5K *Monkey* events) to achieve a nearly optimal (76%) `ActivityCoverage` as the "sweet spot" between effectiveness and efficiency.

More specifically, in Figure 3 we plot how `ActivityCoverage` varies between malicious and benign apps when 5K *Monkey* events are executed. There is an obvious distinction between their `ActivityCoverage` rates—for all typical statistical metrics (min, median, mean, and max), benign apps have larger `ActivityCoverage` rates. The *Pearson correlation coefficient* (*PCC*) between `ActivityCoverage` and the malice of apps is non-trivial: $PCC = -0.12$, with p-value<0.001 (in all experiments throughout this paper, we only use the *PCC* results with < 0.05 p-value to ensure their statistical significance; therefore, we do not specify the p-value when presenting *PCC* results hereafter). Hence we hypothesize that malicious apps are intentionally making it hard for dynamic analysis tools to trigger malicious behaviors. Therefore, we use `ActivityCoverage` as a novel feature for detecting malicious overlays.

3.3 Global Statistics

Through the above described app emulation and UI exploration, we find that overlays are pervasively used by more than 40% of the Android apps in our dataset (including both malicious and benign apps). Here we say "more than" because our app emulation and UI exploration processes are not exhausting all overlays used by all apps. Using the malicious app labels provided by T-Market (whose labeling process is detailed in §3.1 and §2.2), we find that overlays are being used by 51% of malicious apps but only 36% of benign apps. As mentioned in §2.1, the overlay feature is actually powerful in acquiring and affecting user interactions (especially for Android apps), which is the major reason why there are so many Android apps that make use of (or take advantage of) overlays.

[2]More in detail, we have modified the employed Android system running on our emulators, so as to selectively authenticate the permissions for permission-based app usages. Nevertheless, we are currently not able to deal with the app usage scenarios where specific usernames and passwords are required.

More in detail, we wonder how many overlays a benign app and a malicious app use respectively. To this end, we devise a new feature NumOfOverlays to count the number of detected overlays in an *overlay-based* app in our study. As depicted in Figure 4, 72% of overlay-based benign apps and 91% of overlay-based malicious apps use only one overlay. Both the average number (1.1) and maximum number (12) of overlays used in malicious apps are smaller than those (average: 1.5, max: 28) used in benign apps. By manually checking a random sample of overlay-based malicious and benign apps, we find that malicious apps usually have less functionality than benign apps, and thus do not need to utilize as many overlays.

3.4 Profiling Key Overlay Features

In this section, we analyze in detail each of the key overlay features, so as to understand *what a malicious overlay looks like* in the wild. All the overlay features studied in this work are manually picked based on our domain knowledge and data analysis. We had tried some automatic methods for feature selection, but the effect turned out to be rather limited. We first analyze the two static features, and then discuss the two dynamic features (Type and Flags) that have the strongest correlations with the malice of apps. Finally, we elaborate several important appearance features.

3.4.1 *Understanding Static Features.*

- BIND_ACCESSIBILITY_SERVICE. This permission is granted for accessibility services specially designed to assist disabled Android users. Unfortunately, because an app with this permission has manifold powerful capabilities (*e.g.,* getting the notification of any event that affects the device, accessing the full View tree, and programmatically performing click or scroll actions), this permission can be exploited to launch powerful attacks [18]. As mentioned in §1, the Android UI feedback loop can be completely compromised and controlled by exploiting the BIND_ACCESSIBILITY_SERVICE and SYSTEM_ALERT_WINDOW permissions (the "cloak and dagger" attack [10]). In our dataset, 1.3% of apps utilize accessibility services, among which 2.5% are malicious. In particular, 0.11% of apps utilize both accessibility services and SYSTEM_ALERT_WINDOW overlays, among which 1.5% are malicious. Although the number of apps using this permission is small, we still pay attention to the simultaneous usages of BIND_ACCESSIBILITY_SERVICE and SYSTEM_ALERT_WINDOW, given the devastating effect of the "cloak and dagger" attack.
- PACKAGE_USAGE_STATS. This permission allows an app to collect the usage statistics of other apps including the foreground app. Acquiring it can help a malicious app launch more intelligent overlay-based attacks. In our dataset, 2.2% of apps utilize this permission, among which 5.2% are malicious. In particular, 0.36% of apps utilize both PACKAGE_USAGE_STATS and overlays, among which 6.4% are malicious. Although the overlay-based attacks coupled with PACKAGE_USAGE_STATS are not so devastating as the "cloak and dagger" attacks, we still need to be cautious of the simultaneous usages of PACKAGE_USAGE_STATS and overlays.

3.4.2 *Understanding* Type *and* Flags.

- Type. An overlay can have a total of 16 Types, among which the six Types listed in Figure 5 are the most frequently used. In comparison, the remaining 10 Types are together used by less than 0.1% of overlay-based apps. Most notably, 84% of the apps that use TYPE_SYSTEM_ERROR overlays are malicious, and the *PCC* between TYPE_SYSTEM_ERROR and the malice of apps is as high as 0.69. This is because TYPE_SYSTEM_ERROR possesses the highest priority among all the Types—a TYPE_SYSTEM_ERROR overlay can even appear on top of the lock screen interface. On the other hand, although TYPE_TOAST overlays are utilized in nearly 1/3 of overlay-based apps, merely 5% of the apps that use TYPE_TOAST overlays are malicious and the corresponding *PCC* is as low as -0.29. This is because "toasts" have relatively limited capabilities and are typically used for transient notifications.
- Flags. We study all the 31 Flags of Android overlays. Figure 6 depicts the statistics of the 13 most frequently used Flags. We observe that the top four Flags' correlations with the malice of apps differ greatly. For example, although FLAG_NOT_FOCUSABLE is used in 2/3 of overlay-based apps, only 10% of these apps are malicious and the corresponding *PCC* is as low as -0.57. The reason is straightforward—a FLAG_NOT_FOCUSABLE overlay cannot get users' input events independently (*i.e.,* it also needs the permission of FLAG_WATCH_OUTSIDE_TOUCH). In contrast, 82% of the apps that use FLAG_FULLSCREEN overlays are malicious and the *PCC* is as high as 0.68. This is because a FLAG_FULLSCREEN overlay can cover the whole screen (including the status bar) and thus can easily deceive mobile users. It is worth noting that although we individually state each Flag here, our adopted random forest model (§4.1) will inherently and automatically take the combinations of these Flags into consideration. As a matter of fact, such automatic combination is also applicable to the other overlay features studied in this work.

3.4.3 *Understanding Appearance Features.*

- Format and Alpha. Among the 18 appearance parameters in Table 1, Format is of the highest importance to malicious overlay behavior since it determines the basic bitmap transparency of an overlay. As shown in Figure 7, among the three major Formats: RGBA_8888, TRANSLUCENT and TRANSPARENT, RGBA_8888 is not only the most frequently used but also the most related to the malice of apps. This is because RGBA_8888 means that the overlay can be of any transparency, and thus gives the overlay the greatest presentation freedom.

  Supplementary to Format, Alpha also impacts the transparency of an overlay. Since Alpha is a continuous value lying between 0.0 (fully transparent) and 1.0 (fully opaque), we manually divide the value scope into three ranges: [0, 0.5], (0.5, 1.0) and 1.0. From Figure 8, we observe that Alpha = 1.0 is not only the most frequently used but also the most related to the malice of apps. This can be reasonably ascribed to the fact that Alpha = 1.0 is the default configuration for a View and few developers would adjust this configuration. Thus, we infer that Alpha should not be an important property in detecting malicious overlay behavior.
- VisualCoverage. Based on our experiences of manually examining the layouts of many overlays used by malicious and benign apps, we propose a novel appearance feature named VisualCoverage that denotes the ratio of the host View's area visually covered by the overlay(s). When the host View is fully or partially covered by an overlay, the overlay's VisualCoverage is

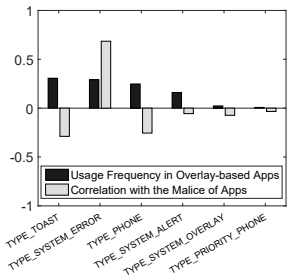

Figure 5: Frequently used `Type`s and their *PCC*s with the malice of apps.

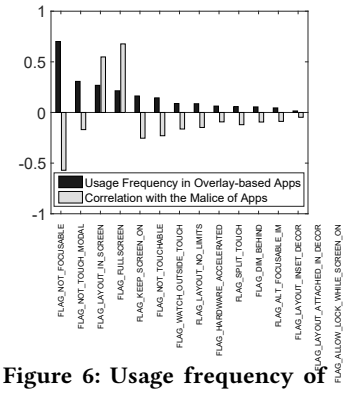

Figure 6: Usage frequency of `Flag`s and their *PCC*s with the malice of apps.

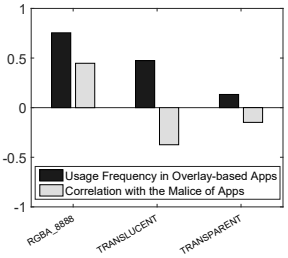

Figure 7: Frequently used `Format`s and their *PCC*s with the malice of apps.

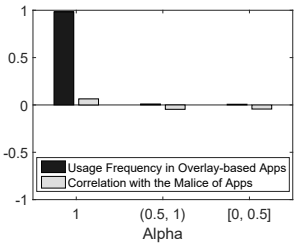

Figure 8: Usage frequency of different ranges of `Alpha` & the *PCC*s with the malice of apps.

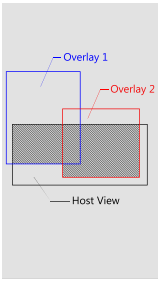

Figure 9: Our calculation of the `VisualCoverage` for multiple overlays.

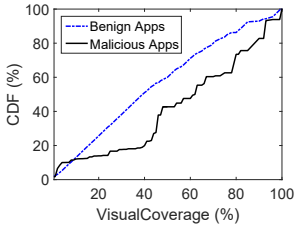

Figure 10: CDF of the overlay(s)' `VisualCoverage` for benign and malicious apps.

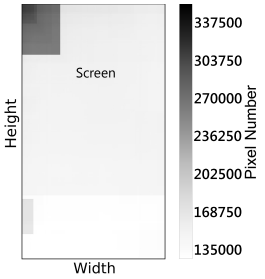

Figure 11: Heat map of the `VisualCoverage` scopes for benign apps' overlays.

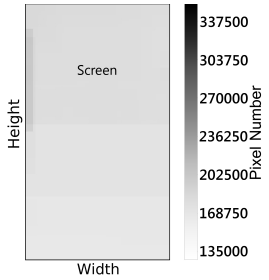

Figure 12: Heat map of the `VisualCoverage` scopes for malicious apps' overlays.

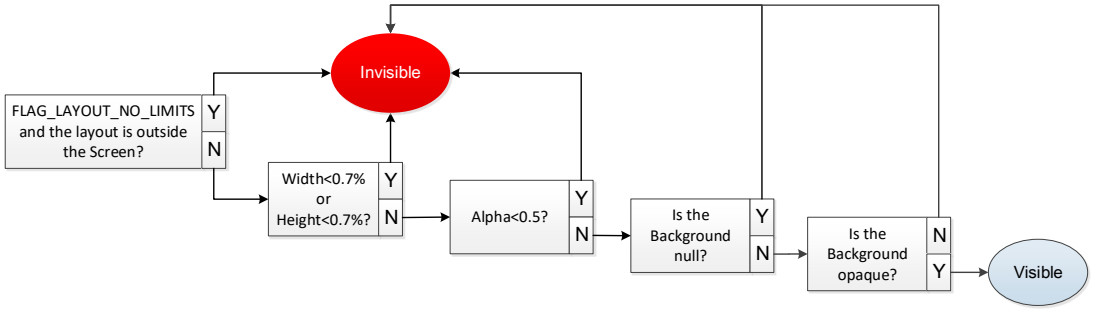

Figure 13: Flow chart for our calculation of `isReallyVisible` for an overlay. If an overlay is an instance of a `ViewGroup`, it is really visible as long as one child `View` of the `ViewGroup` is really visible.

calculated by dividing the intersection area by the area of the host `View`. A more complex case is in Figure 9—there is one host `View` and two overlays on the screen, so the overlays' `VisualCoverage` is calculated by dividing the shaded area by the area of the host `View`. As shown in Figure 10, overlays' `VisualCoverage` exhibits distinct distributions between malicious and benign apps. For benign apps' overlays, `VisualCoverage` is almost uniformly distributed; for malicious apps' overlays, the distribution of `VisualCoverage` is highly skewed. Thus, the *PCC* between `VisualCoverage` and the malice of apps is fairly high (0.36).

- `Y` and `Gravity`. We further consider each overlay's `VisualCoverage` *scope*, denoting the host `View`'s geometric scope visually covered by the overlay. Figures 11 and 12 plot the heat maps of the `VisualCoverage` scopes for benign and malicious apps' overlays. The frame of each figure represents the screen of common Android smartphones. Again, we notice distinct distributions between the overlays' `VisualCoverage` scopes of malicious and benign apps. Specifically, we observe that for benign apps' overlays, the `VisualCoverage` scopes tend to locate at the top left corner of the screen (*i.e.,* a small-area rounded or squared overlay floating at the top left corner, showing system status information). But for malicious apps' overlays, the `VisualCoverage` scopes do not have a preferred region in the screen. This indicates that an overlay's `Y` coordinate and `Gravity` are also correlated with

the malice of its affiliated app—recall that `Gravity` decides the placement of an overlay within a larger UI container.

- `isReallyVisible`. Visibility is critical for a user's perception of an overlay. Unfortunately, a programmatically visible overlay can be visually invisible to users, *e.g.,* in Figures 1(b) and 1(c) if the overlay is transparent, in Figure 1(d) where the overlay is too small to see with naked eyes, or in Figure 1(e) where the overlay is outside the screen. This fact is often exploited by malicious apps. To cope with this issue, we calculate a novel feature `isReallyVisible` based on an overlay's appearance features including `Width`, `Height`, `Alpha`, `Background`, `isOpaque`, and so on. The workflow is plotted in Figure 13. Among all the apps that have used overlays, 31.2% of malicious apps and 15.6% of benign apps are using overlays that are not really visible, showing the significance of `isReallyVisible`.

3.5 Summary of the Study Results

Our comparative study leads to a series of useful insights with respect to malicious overlay behavior:

- Overlays are used by more than 40% of Android apps overall in our dataset, and 51% of malicious apps.
- On the other hand, both the average and maximum numbers of overlays used in malicious apps are smaller than those in benign apps. This is because malicious apps usually have less functionality than benign apps.
- We observe malicious apps intentionally make it hard for dynamic analysis tools to detect their overlays.
- `Type`, `Flag`, and `Format` are the three features that correlate most strongly with an app's malice, *e.g.,* 84% of the apps that use `TYPE_SYSTEM_ERROR` overlays are malicious (*PCC* = 0.69); more than two thirds of the apps that use `FLAG_FULLSCREEN` or `FLAG_LAYOUT_IN_SCREEN` overlays are malicious (*PCC* = 0.68 and 0.55); and 48% of the apps that use `RGBA_8888` overlays are malicious (*PCC* = 0.45).
- We design a complex feature `VisualCoverage` that reveals distinct distributions between malicious and benign apps' overlays.
- A programmatically visible overlay can be visually invisible to users, and this fact is often exploited by malicious apps. To make it clear, we develop a novel feature `isReallyVisible` based on multiple existing appearance features.

4 DETECTING OVERLAY-BASED MALWARE

Guided by our study results in §3, we build the OverlayChecker system to detect overlay-based malware (§4.1) and evaluate its efficacy using T-Market as a real-world case study (§4.2). We demonstrate its extensibility by applying it to Google Play Store (§4.3) and discuss its robustness to attackers' evasion attempts (§4.4).

4.1 Design and Implementation

OverlayChecker takes an app's APK file as input and outputs the estimated malice of every detected overlay in the app. It checks each app with three steps:

- **Step 1: Overlay feature extraction** creates the features for detection, as described in §3.2.
- **Step 2: Overlay/app classification** evaluates the malice (between 0 and 1.0) of every detected overlay based on the extracted features. The per-overlay scores are then combined to label the app as malicious or benign with respect to overlay behavior.
- **Step 3: Assisting app review** helps the app store vendor to review and release newly submitted apps.

The classification model used in **Step 2** is built from *random forest learning* [4] using the labeled training set consisting of known malicious and benign apps from T-Market (refer to §3.1), together with their extracted 56 features per overlay. To calibrate the random forest probabilities, We used a parametric approach based on Platt's sigmoid model. We use a random forest model for OverlayChecker because it exhibits higher precision and recall than a single decision tree on our task. Additionally, the training process of a random forest is faster than that of a complex classifiers (*e.g.,* SVM), and the resulting trained model is interpretable. We present a detailed comparison of five classifiers in §4.2.

To label newly submitted apps as malicious (M) or benign (B) in terms of overlay behavior, OverlayChecker uses a three-step process. *First*, OverlayChecker quantifies the malice of each detected overlay in a given app as a *confidence* value, denoted as *CoM* (Confidence of Malice), between 0 and 1.0 using the classification model. In Figure 14 we plot the CDF of the malice of all overlays detected in our study, from which we observe distinct distributions between malicious apps and benign apps. *Second*, OverlayChecker produces a confidence score for the entire app; to be conservative, we use the malice of the most malicious overlay in the app. *Third*, OverlayChecker labels the app as malicious if the confidence is above a specific threshold. Based on the malice of known apps provided by T-Market, we draw the curves for both false positive rate and false negative rate, as illustrated in Figure 15. Then, we find the minimum sum of the two rates as the confidence threshold by adding the two curves together. For our studied dataset, we configure the confidence threshold as 0.24 (Malicious: >0.24, Benign: ≤0.24). While the concrete value of our calculated confidence threshold may not be applicable to other app stores, it can be easily and automatically calculated by them with the same methodology.

Integration to a real app market. OverlayChecker has been integrated into T-Market as a part of app review process since Jan. 2018. It automatically detects overlay-based malicious apps submitted to T-Market in a user-transparent manner, so all the apps currently released on T-Market (which users can see and download) are safe in terms of their utilized overlays.

4.2 Evaluation

The key efficacy metrics of OverlayChecker are its precision, recall, and execution speed. In Table 2 we present a comparison of six different machine learning classifiers trained on: (1) the 52 overlay features from the Android SDK, and (2) these 52 features plus the four novel features we developed in §3.4. Following the best practices in machine learning, we leverage 10-fold cross-validation when evaluating precision and recall. We see that our four novel features improve the detection accuracy of malicious overlays by ~3% regardless of classifier – note that such 3% increase is not trivial

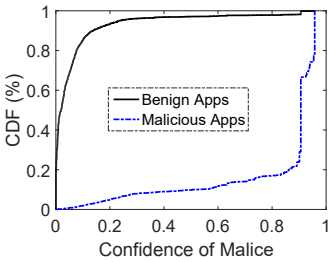

Figure 14: CDF of the CoM (Confidence of Malice) of all the overlays detected in our study.

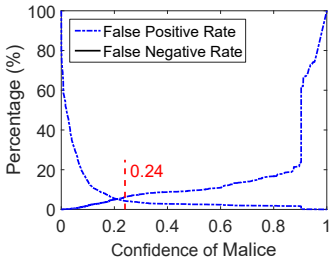

Figure 15: Minimize false positive + negative rate by tuning the threshold for CoM.

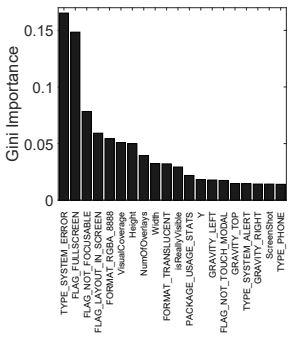

Figure 16: Top-ranking features for early detection of malicious overlay behavior.

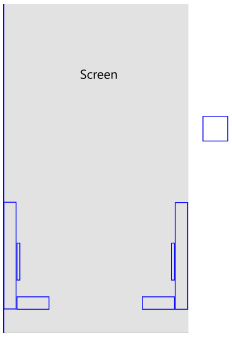

Figure 17: Du-Antivirus leverages multiple overlays with weird shapes and locations.

Algorithm	Precision (56 / 52)	Recall (56 / 52)	Training Time (56 / 52) second
Naive Bayes	0.88 / 0.85	0.72 / 0.73	2 / 2
Logistic Regression	0.87 / 0.84	0.84 / 0.81	6 / 6
Random Forest	0.96 / 0.93	0.96 / 0.93	35 / 35
RBF-SVM	0.95 / 0.91	0.95 / 0.91	1626 / 1625
Linear-SVM	0.93 / 0.90	0.93 / 0.90	11551 / 11542
Poly-SVM	0.91 / 0.89	0.91 / 0.88	59294 / 58762

Table 2: Efficacy of different classification algorithms using 56 overlay features *vs.* the 52 original overlay features.

when the precision/recall is already very high (> 90%), thus demonstrating their utility. Also, we see that the random forest model achieves the highest precision (96%) and recall (96%) amongst the six models, while having a relatively short training time of 35 seconds on a commodity x86 server. These results explain our decision to use a random forest model in OverlayChecker.

To understand the 4% false positives, we manually inspected the detection logs and found that the 4% errors reflect some problems stemming from the abuse of overlays. In fact, although we determined that the 4% false positives are benign apps, 97% of them had irregular overlay behavior, particularly using `TYPE_SYSTEM_ERROR` overlays. As a matter of fact, the 4% false positive rate is considered acceptable by T-Market, as apps flagged by OverlayChecker receive a 2nd-round manual review. Specifically, among the nearly 10K apps submitted per day, there are usually about 1200 apps flagged by OverlayChecker, among which nearly 1150 are meanwhile labeled as malicious by T-Market (using its own high-precision security checking mechanisms, refer to §2.2). Thus, OverlayChecker only has around 50 (= 1200 − 1150) apps requiring manual review, and only around a couple of them are false alarms.

As we observe in §3.3, overlays are used by 51% of malicious apps, so the high recall (96%) illustrates that OverlayChecker is able to detect nearly half (49% = 51%×96%) of *all* malicious apps hosted on T-Market using solely automated overlay behavior analysis. More specifically, using the category labels of apps maintained by T-Market (§2.2), OverlayChecker can detect the vast majority (90+%) of certain types of malicious apps, *e.g.,* 99% of ransomware, 97% of adware, 93% of porn-fraud, and 90% of SMS-fraud apps. The reason is intuitive: such malicious apps heavily rely on overlays to launch their desired attacks. Ransomware apps use `TYPE_SYSTEM_ERROR` overlays to show ransom messages on top of users' lock screens; adware and porn-fraud apps exploit `SYSTEM_ALERT_WINDOW` overlays to show ads on top of other apps; SMS-fraud apps use `SYSTEM_ALERT_WINDOW` overlays to capture users' telephone call information (for sending fraud SMS messages later).

In detail, Figure 16 presents the Gini [5] importance of the top-20 important features in our trained random forest model (we use Gini importance to profile the correlations between the overlay features and the malice of apps due to its special suitability for the random forest model). Overall, the results are consistent with our measurement findings in §3.3 and §3.4. For example, `TYPE_SYSTEM_ERROR`'s highest importance complies with its highest correlation with the malice of apps. Similarly, the very high importance of `FLAG_FULLSCREEN` and `FLAG_LAYOUT_IN_SCREEN` conform to their high correlations with the malice of apps. Specially, our introduced novel features `VisualCoverage`, `NumOfOverlays` and `isReallyVisible` rank the 6^{th}, 8^{th} and 11^{th}.

Figure 16 also shows the high effectiveness of dynamic features. Among the 56 features, only two (`PACKAGE_USAGE_STATS` and `BIND_ACCESSIBILITY_SERVICE`) are static and their importances ranked only 12th and 44th. As discussed in §3.2, this is because many important characteristics of overlays only exhibit dynamically at app runtime. This thus concretely shows that it is necessary to consider dynamic features to ensure high detection effectiveness.

We update our classification model on a monthly basis using data from newly submitted apps. When the classification model is ready and integrated into OverlayChecker, the evaluation time for one app is within 2 minutes, among which nearly 100 seconds are spent on overlay feature extraction (cf. §3.2.2). OverlayChecker is able to check ~10 K apps submitted to T-Market per day using a single commodity x86 server (cf. §3.2.1 for the detailed configurations).

4.3 Extensibility

OverlayChecker is not limited to T-Market and can be directly applied to other app stores. First, our research methodology can be applied by other app stores as it only requires the APK file and security label of each app as preconditions. At present, almost all app stores can provide the APK files of its hosted apps, and most mainstream app stores maintain their own database of malicious

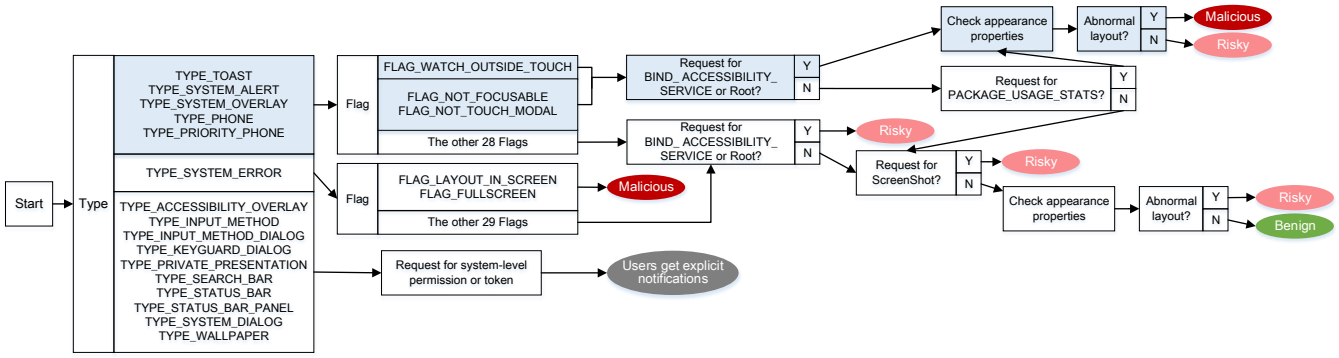

Figure 18: Simplified flow chart of the classification model used by OverlayChecker, illustrating the robustness of OverlayChecker to knowledgeable attackers. As a typical case, the recent overlay-based "cloak and dagger" attacks must go through the steps highlighted in light blue, which always result in a malicious or risky classification.

apps. Further, although the construction of our classification model relies on the app dataset provided by T-Market, once the model is trained OverlayChecker can work independently of T-Market and help other app stores detect malicious overlay behavior.

In order to validate the practical extensibility of OverlayChecker, we applied it to 10 K randomly sampled apps released on Google Play Store. Despite Google's own sophisticated security checking, OverlayChecker is still able to detect 20 (0.2%) apps with malicious overlay behavior (among the 22 apps flagged by OverlayChecker). We manually checked each suspicious app to examine whether there was actually malicious overlay behavior. For example, one suspicious app "Lock Screen Master" earned an alarming CoM value mainly because of its simultaneous usages of `SYSTEM_ALERT_WINDOW` and `BIND_ACCESSIBILITY_SERVICE`. Another suspicious app "Du-Anti-virus" was determined to be malicious because it leveraged multiple overlays with weird shapes and locations as depicted in Figure 17. Interestingly, three days after our experiment these apps were removed from Google Play Store, potentially due to reports from users. However, since these apps were officially available, users who installed them were potentially vulnerable for at least three days. This indicates the pressing need today for a market-scale early detection system like OverlayChecker.

4.4 Robustness to Evasion Attempts

In the learned classification logic in OverlayChecker, it may not be difficult for a knowledgeable attacker to pick a single overlay feature used in our system and make it look (more) benign, but the key point of OverlayChecker's detection is to consider all features of an overlay in combination to determine its malice. This thus significantly raises the bar of creating a powerful malicious overlay, making the detection in OverlayChecker difficult to evade even if the attacker can reverse engineer our classification model (*e.g.,* by trial-and-error attempts). To illustrate that, Figure 18 depicts a simplified flow chart demonstrating the branches of the model. In its essence, the model takes a series of features of a given overlay as inputs, and classifies the overlay to be *malicious*, *benign*, or *risky* (meaning that an overlay is highly likely to be malicious). In our deployed system, every overlay is determined to be either malicious or benign, but because this example is simplified (*i.e.,* missing some evaluation steps) we add the extra category.

As shown in Figure 18, features examined at early stages are more important than those in later stages. `Type`, `Flags`, and requests for `BIND_ACCESSIBILITY_SERVICE` and `Root` permissions are examined first—they play the most important role in determining the malice of an overlay. This is because these features determine the fundamental capabilities of an overlay (as explained in §3.4.1 and §3.4.2), such as whether the overlay can display on top of other `Views`, get the user's input events, and programmatically perform click or scroll actions. In contrast, `ScreenShot` and `PACKAGE_USAGE_STATS` permissions are requested to assist the actions of an overlay, so they are relatively less dangerous and examined afterwards. Appearance features are examined at the last stage to determine whether an overlay exhibits an "abnormal" layout (to the user's eyes). Here "abnormal" means that the overlay has an abnormal appearance in terms of one or more appearance properties (refer to Table 1). Since OverlayChecker is data driven and machine learning-based, it can automatically identify certain combinations of appearance properties as an abnormal layout.

Thus, the organization of the trained classifier precludes attackers from adopting the vast majority of malicious techniques in their overlays. For example, the most recent class of overlay-based attacks, "cloak and dagger" attacks [10], can only go through a fixed path in Figure 18 (highlighted in light blue), which always results in a *malicious* or *risky* classification. An attacker seeking a benign label can evade the detection of our system by carefully tuning certain parameters; in this case, however, he must sacrifice many powerful capabilities, such as binding to accessibility services, accessing user events, or displaying over the full screen. This seriously limits the power of the attacker's overlays, since most existing malicious overlay strategies are precluded (demonstrated by the 96% precision/recall in §4.2). Furthermore, the abnormal layout tests restrict the attacker's ability to obfuscate the overlays from users. Finally, we have to admit that security issues, in particular the overlay-based security, are often games of cat and mouse. As the "cat", OverlayChecker owns its value by precluding most existing malicious overlay strategies, and attackers are left with a significantly weaker range of attack capabilities. Since the model is updated frequently, *e.g.,* daily in the current integration with T-Market, such classification logic will become even more restricted after considering more malicious overlay behaviors in the future.

5 RELATED WORK

Given the prevalence and severity of overlay-based security issues, a number of previous studies have analyzed overlay-based attacks (§5.1) and proposed defense approaches (§5.2).

5.1 Overlay-based Attacks

Direct attacks construct deceptive overlays which confuse users to misinterpret UI interactions. Figure 1 classifies such attacks into five groups. *First*, malicious *redressing* overlays (in Figure 1(a)) impersonate small UI widgets (*e.g.,* buttons) as a part of the current UI window, thus triggering users to click [19]. *Second*, malicious transparent overlays (in Figure 1(b)) are made invisible to cover victim apps, causing users to see the visible one but operate on the invisible one. Massive GUI hijacking attacks based on these transparent overlays have been reported to lure users to type passwords (by hijacking keyguards) or grant permissions (by hijacking security alerts) [3, 9, 30, 40]. Malicious transparent overlay attacks can also be launched through `WebView` in Android to compromise web content [21, 22]. *Third*, malicious *hollow-out* overlays (in Figure 1(c)) selectively uncover UI components of victims apps, misleading users over the meaning of the interaction by manipulating the covered overlay [19]. *Fourth*, malicious *hover* overlays (in Figure 1(d)) are too tiny in size to be noticed visually. For example, hover overlays have reportedly been abused by malicious apps to capture sensitive inputs (*e.g.,* passwords and credit card numbers) [10, 37]. *Finally*, malicious overlays *outside the screen* (in Figure 1(e)) cannot be noticed by users, but can still maliciously capture UI events.

Indirect attacks. Overlay-based attacks can also be constructed indirectly through UI inference and user behavior analysis. An adversary can launch overlay-based attacks by inferring UI states using shared memory side channels [6]. Moreover, the location of screen taps on mobile devices can be identified from certain sensors [11, 13, 14, 23, 24, 32, 36, 39]. This empowers non-trivial overlay-based attacks based on users' tapping behavior. In addition, Ren *et al.* reveal the design flaws of Android's task management that make it vulnerable to *task hijacking*, which tricks the system to display malicious overlays whenever the user launches an app [31].

5.2 Defense Approaches

Security indicator-based defenses. WhatTheApp adds an on-device security indicator to the system navigation bar to identify the top `Activity` and inform users about the origin of the app with which they are interacting [3]. However, WhatTheApp is vulnerable to timing attacks because the security indicator is calculated periodically—a malicious overlay can be inserted within the period [9]. To fix this problem, Overlay Mutex was proposed to prevent a background non-system app from rendering on top of any foreground apps [9]. Note that WhatTheApp, Overlay Mutex, and other UI/security indicator-based defenses [8] require domain knowledge to modify the existing Android framework, making them difficult to deploy by the current Android community.

Static detection-based defenses. Prior work proposes to detect malicious overlays by analyzing overlay properties specific to predefined attack vectors using static program analysis [3]. Despite the same target, OverlayChecker differs in the following aspects. First, OverlayChecker extracts both static and dynamic properties of each overlay. Thus, it does not suffer from the limitations of static methods, such as dealing with reflection, class loading, or native code [3]. Second, OverlayChecker uses a data-driven approach based on a large-scale, real-world dataset of malicious apps (refer to §3), rather than predefined attack vectors.

DECAF. Liu *et al.* present the DECAF system for detecting ad fraud in Windows phone apps [20]. While DECAF and OverlayChecker target different threats (ad fraud versus malicious overlays), they have several commonalities: (1) an early detection approach with *Monkey* emulations; (2) some ad fraud windows are in fact overlays; and (3) adoption by mainstream app stores (Microsoft and T-Market). On the other hand, they have essential methodological differences. In particular, DECAF leverages legally enforceable terms and conditions to detect ad fraud, while OverlayChecker acquires its detection mechanisms via a comparative study of the overlay behavior between benign and malicious apps, since there are no regulations on the overlay usage. In fact, this makes OverlayChecker more complicated and comprehensive.

WhatTheApp and CDS. Bianchi *et al.* present the WhatTheApp system which uses static analysis techniques to determine whether there is a malicious app running on an Android device [3]. It is commendable that they consider a wide variety of attack vectors to judge malicious Android apps. Unfortunately, WhatTheApp has been proven flawed in [9] that it is quite easy for a malicious app to bypass the periodic checking of WhatTheApp; moreover, WhatTheApp is vulnerable to side-channels attack [35]. In contrast, OverlayChecker does not have these problems since it is data driven and a machine learning-based early detection system. Thus, it can detect more crafty malicious apps and is more user-friendly.

CDS is another static analysis system which leverages only four fixed features to determine the malice of an overlay-based app [38], and it uses alert windows to remind users. Compared to our OverlayChecker, its examined features are much fewer and thus not comprehensive enough; also, its used alert windows are vulnerable to the overlay attacks themselves while OverlayChecker is immune to this problem as an early detection system.

ClickShield. Possemato *et al.* present the ClickShield system for detecting clickjacking on Android [27], as well as the "hide overlays" defense proposed by Google which is only useful for system apps. ClickShield and OverlayChecker have a similar target but essentially different methodologies for malware detection. ClickShield examines a series of rules such as the position and pixel difference, so it owns the advantages of simplicity and stability. Nevertheless, an attacker can evade its detection by designing certain conditions beyond the rules, *e.g.,* it will suffer from a large amount of calculation on pixel difference during the malware detection. Compared to ClickShield, OverlayChecker is more complicated and may generate unstable results due to its data-driven methodology, but is more powerful in terms of detection coverage and robustness.

6 CONCLUSION

Usability and security often constitute two sides of a tool in real world. At present in the Android OS, there is enormous tension between the remarkable usability and severe security threats of

overlays. Without effective countermeasures, attackers can exploit overlays to fully compromise and control the UI feedback loop of Android devices. This paper addresses this tension by exploring the possibility of enabling the detection of overlay-based malicious apps at the app market level. We conduct a comparative study of the overlay behavior between benign and malicious apps, based on a large-scale, ground-truth dataset from T-Market, one of the world's largest Android app stores. Guided by a number of useful insights revealed by our study, we design and deploy the OverlayChecker system to quickly and automatically detect overlay-based malicious apps with high precision and recall. OverlayChecker is integrated into T-Market as an important part of the app review process, and we apply OverlayChecker to random apps in Google Play Store to further confirm its efficacy.

ACKNOWLEDGMENTS

We sincerely thank our shepherd Dr. Landon Cox and the anonymous reviewers for their valuable feedback. We also appreciate Hai Long and Zipeng Wu for their contributions to the deployment of OverlayChecker. This work is supported in part by the National Key R&D Program of China under grant 2017YFB1003000, and the National Natural Science Foundation of China (NSFC) under grants 61632013, 61822205, 61432002 and 61632020.

REFERENCES

[1] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. DroidAPIMiner: Mining Api-Level Features for Robust Malware Detection in Android. In *Proceedings of EAI SecureComm*. Sydney, Australia.

[2] Fabrice Bellard. 2005. QEMU, A Fast and Portable Dynamic Translator. In *Proceedings of USENIX ATC*. Anaheim, CA, USA.

[3] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. What the App is That? Deception and Countermeasures in the Android User Interface. In *Proceedings of IEEE S&P*. San Jose, CA, USA.

[4] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (October 2001), 5–32.

[5] Leo Breiman, Jerome H. Friedman, Richard. Olshen, and Charles J. Stone. 1984. Classification and Regression Trees. *Encyclopedia of Ecology* 40, 3 (January 1984), 582–588.

[6] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. 2014. Peeking into Your App without Actually Seeing it: UI State Inference and Novel Android Attacks. In *Proceedings of USENIX Security*. San Diego, CA, USA.

[7] Sanorita Dey, Nirupam Roy, Wenyuan Xu, and Srihari Nelakuditi. 2013. Leveraging Imperfections of Sensors for Fingerprinting Smartphones. *ACM SIGMOBILE Mobile Computing and Communications Review* 17, 3 (July 2013), 21–22.

[8] Earlence Fernandes, Qi Alfred Chen, Georg Essl, J. Alex Halderman, Z. Morley Mao, and Atul Prakash. 2014. *TIVOs: Trusted Visual I/O Paths for Android*. Technical Report CSE-TR-586-14. University of Michigan, Ann Arbor.

[9] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J. Alex Halderman, Z. Morley Mao, and Atul Prakash. 2016. Android UI Deception Revisited: Attacks and Defenses. In *Proceedings of FC*. Barbados.

[10] Yanick Fratantonio, Chenxiong Qian, Simon Chung, and Wenke Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proceedings of IEEE S&P*. San Jose, CA, USA.

[11] Mayank Goel, Jacob Wobbrock, and Shwetak Patel. 2012. Gripsense: Using Built-In Sensors to Detect Hand Posture and Pressure on Commodity Mobile Phones. In *Proceedings of ACM UIST*. Cambridge, MA, USA.

[12] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. Reran: Timing and Touch-sensitive Record and Replay for Android. In *Proceedings of ACM/IEEE ICSE*. San Francisco, CA, USA.

[13] Jun Han, Emmanuel Owusu, T. Nguyen Le, Adrian Perrig, and Joy Zhang. 2012. ACComplice: Location Inference Using Accelerometers on Smartphones. In *Proceedings of IEEE COMSNETS*. Bangalore, India.

[14] Ronny Hänsch, Tobias Fiebig, and Jan Krissler. 2014. Security Impact of High Resolution Smartphone Cameras. In *Proceedings of USENIX WOOT*. San Diego, CA, USA.

[15] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps. In *Proceedings of ACM MobiSys*. Bretton Woods, NH, USA.

[16] Hongqi He, Liehui Jiang, Haifeng Chen, and Weiyu Dong. 2012. Hardware/Software Co-design of Dynamic Binary Translation in X86 Emulation. In *Proceedings of IEEE CSAE*. Zhangjiajie, China, 283–287.

[17] Chih-Wei Huang. 2016. Android-x86 Vendor Intel Houdini. https://osdn.net/projects/android-x86/scm/git/vendor-intel-houdini/.

[18] Yeongjin Jang, Chengyu Song, Simon P Chung, Tielei Wang, and Wenke Lee. 2014. A11y Attacks: Exploiting Accessibility in Operating Systems. In *Proceedings of ACM CCS*. Scottsdale, AZ, USA.

[19] Eduard Kovacs. 2016. Most Android Devices Prone to Accessibility Clickjacking Attacks. http://www.securityweek.com/most-android-devices-prone-accessibility-clickjacking-attacks.

[20] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. 2014. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *Proceedings of USENIX NSDI*. Seattle, WA, USA.

[21] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android System. In *Proceedings of ACM ACSAC*. Orlando, FL, USA.

[22] Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. 2012. Touchjacking Attacks on Web in Android, iOS, and Windows Phone. In *Proceedings of FPS*. Montreal, QC, Canada.

[23] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. 2012. TapPrints: Your Finger Taps Have Fingerprints. In *Proceedings of ACM MobiSys*. Low Wood Bay, Lake District, UK.

[24] Sashank Narain, Amirali Sanatinia, and Guevara Noubir. 2014. Single-Stroke Language-Agnostic Keylogging Using Stereo-Microphones and Domain Specific Machine Learning. In *Proceedings of ACM WiSec*. Oxford, UK.

[25] Marcus Niemietz and Jörg Schwenk. 2012. UI Redressing Attacks on Android Devices. In *Proceedings of Black Hat*. Abu Dhabi, United Arab Emirates.

[26] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *Proceedings of ACM EuroSec*. Amsterdam, The Netherlands.

[27] Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. 2018. Clickshield: Are you hiding something? Towards eradicating clickjacking on Android. In *Proceedings of ACM CCS*. Toronto, Canada.

[28] Android Open Source Project. 2015. Allow verifier to grant permissions. https://github.com/aosp-mirror/platform_frameworks_base/commit/4ff3b614ab73539763343e0981869c7ab5ee9979.

[29] Android Open Source Project. 2015. Make SYSTEM_ALERT_WINDOW development permission. https://github.com/aosp-mirror/platform_frameworks_base/commit/01af6a42a6a008d4b208a92510537791b261168c.

[30] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. 2015. *An Investigation of the Android/BadAccents Malware which Exploits a new Android Tapjacking Attack*. Technical Report TUD-CS-2015-0065. Technische Universität Darmstadt.

[31] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. 2015. Towards Discovering and Understanding Task Hijacking in Android. In *Proceedings of USENIX Security*. Washington, D.C., USA.

[32] Roman Schlegel, Kehuan Zhang, Xiao Yong Zhou, Mehool Intwala, Apu Kapadia, and Xiao Feng Wang. 2011. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of ISOC NDSS*. San Diego, CA, USA.

[33] Silvestri Simone, Rahul Urgaonkar, Murtaza Zafer, and Bong Jun Ko. 2018. A Framework for the Inference of Sensing Measurements Based on Correlation. *ACM Transactions on Sensor Networks* 15, 1 (December 2018), 4.

[34] Android Studio. 2015. UI/Application Exerciser Monkey in Android Studio. https://developer.android.com/studio/test/monkey.html.

[35] Ming Tang, Maixing Luo, Junfeng Zhou, Zhen Yang, Zhipeng Guo, Fei Yan, and Liang Liu. 2018. Side-Channel Attacks in a Real Scenario. *Tsinghua Science and Technology* 23, 5 (October 2018), 586–598.

[36] Robert Templeman, Zahid Rahman, David Crandall, and Apu Kapadia. 2013. PlaceRaider: Virtual Theft in Physical Spaces with Smartphones. In *Proceedings of ISOC NDSS*. San Diego, CA, USA.

[37] Enis Ulqinaku, Luka Malisa, Julinda Stefa, Alessandro Mei, and Srdjan Capkun. 2017. Using Hover to Compromise the Confidentiality of User Input on Android. In *Proceedings of ACM WiSec*. Boston, MA, USA.

[38] Longfei Wu, Benjamin Brandt, Xiaojiang Du, and Bo Ji. 2016. Analysis of Clickjacking Attacks and An Effective Defense Scheme for Android Devices. In *Proceedings of IEEE CNS*. Philadelphia, PA, USA.

[39] Zhi Xu, Kun Bai, and Sencun Zhu. 2012. TapLogger: Inferring User Inputs On Smartphone Touchscreens Using On-board Motion Sensors. In *Proceedings of ACM WiSec*. Tucson, AZ, USA.

[40] Lingyun Ying, Yao Cheng, Yemian Lu, Yacong Gu, Purui Su, and Dengguo Feng. 2016. Attacks and Defence on Android Free Floating Windows. In *Proceedings of ACM AsiaCCS*. Xi'an, China.

[41] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of ISOC NDSS*. San Diego, CA, USA.