Overlay-based Android Malware Detection at Market Scales: Systematically Adapting to the New Technological Landscape

Liangyi Gong, Zhenhua Li, Hongyi Wang, Hao Lin, Xiaobo Ma, and Yunhao Liu, *Fellow IEEE*

Abstract—Android *overlay* enables one app to draw over other apps by creating an extra `View` layer atop the host `View`, which nevertheless can be 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 sacrificing overlays' usability; recently, the overlay mechanism has been substantially updated to prevent a variety of attacks, which however can still be evaded by considerable adversaries. To address these shortcomings, a more pragmatic approach is to enable *early detection* of overlay-based malware during the app market review process, so that all the capabilities of overlays can stay unchanged. For this purpose, in this paper we first conduct a large-scale comparative study of overlay characteristics in benign and malicious apps, and then implement the OverlayChecker system to automatically detect overlay-based malware for one of the world's largest Android app stores. In particular, we have made systematic efforts in feature engineering, UI exploration, emulation architecture, and run-time environment, thus maintaining high detection accuracy (97% precision and 97% recall) and short per-app scan time (∼1.7 minutes) with only two commodity servers, under an intensive workload of ∼10K newly submitted apps per day.

Index Terms—Android Overlay; Mobile Malware Detection; App Market; User Interaction; Android Emulation; Machine Learning.

1 Introduction

Overlay has been a user interface (UI) feature of Android, which enables a mobile app to draw over other apps by creating an extra `View` layer atop the host `View`, as illustrated in Figure 1. The rationale behind this 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 Multi-Window [1] (for displaying multiple apps in a split-screen mode) has been supported since Android 7.0, it is seldom used by smartphone users [2]. Overlays have been widely adopted by mobile apps installed on hundreds of millions of mobile devices, such as Facebook, Uber, Messenger, Zoom, *etc*. We observe that as of Nov. 2020, 27.2% (136 out of 500) of the most 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], [4], [5], [6], [7], [8], [9]. 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] demonstrates that the UI feedback loop can be completely compromised and controlled through the "cloak and dagger" attack without the Android permission `SYSTEM_ALERT_WINDOW` which is officially assigned to allow an app to create overlays on top of all other apps.

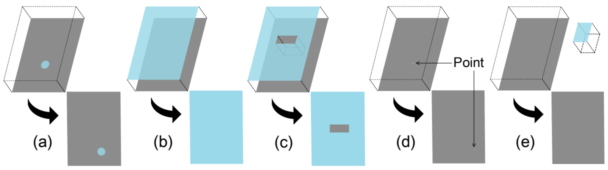

Fig. 1: Five common forms of overlays: (a) float, (b) cover, (c) hollow out, (d) single point, (e) offscreen. The upper row plots the underlying host `View` and the overlay lying on top of it, while the lower row plots the user-perceived view in each case. Benign apps typically use cases (a)(b), while malicious apps use all cases (a)–(e).

Given the severity and prevalence of overlay-based attacks, the Android overlay mechanism has been substantially updated in the release of Android 8.0 in Aug. 2017. Specifically, two-fold measures were taken to facilitate users' comprehensive surveillance on apps' usage of overlays: (1) unifying the original six types of overlays into a single one, and (2) changing the management style of overlay-related permissions from static configuration at installation time to dynamic approval at run time. Unfortunately, despite these efforts, overlay-based attacks still persist due to real-world challenges. On one hand, there still exist ∼40% of Android devices that have not been upgraded to Android 8.0+, thus being vulnerable to overlay-based attacks. For these devices, several countermeasures have been proposed to restrict the capabilities of overlays at the operating system (OS) level [11], [5], [12]. However, these solutions barely see any adoptions by Android due to the concern of sacrificing overlay usability. On the other hand, even for devices running Android 8.0+, malicious apps are still able to launch overlay-based attacks in a two-stage manner [13] – first inducing users to grant overlay-related permissions upon app startup, and then successfully displaying malicious overlays to them during the running process.

To address these issues, 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

- *Liangyi Gong, Zhenhua Li, Hongyi Wang, and Hao Lin are with the School of Software, Tsinghua University. E-mail: {gongliangyi, lizhenhua1983, hywangthu, linhaomails}@gmail.com*
- *Xiaobo Ma is with the Faculty of Electronic and Information Engineering, Xi'an Jiaotong University. E-mail: xma.cs@xjtu.edu.cn.*
- *Yunhao Liu is with the Global Innovation Exchange, Tsinghua University. E-mail: yunhaoliu@gmail.com.*
- *The corresponding author is Zhenhua Li.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2021.3079433, IEEE Transactions on Mobile Computing

released to users), so that all the capabilities of overlays can be retained and even devices that have not been upgraded to Android 8.0+ can be effectively protected. Unfortunately, little has been known about the feasibility and effectiveness of this approach due to the lack of systematic understandings, insights, and datasets of malicious overlays in the wild. 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 these, we collaborate with one of the world's largest Android app stores, *i.e.,* Tencent App Market [14] or Market-T, to perform a large-scale study of overlay behaviors in apps. Using both static and dynamic analyses, we compare the overlay behavior between benign and malicious apps to uncover critical characteristics of malicious overlay in the ground-truth data provided by Market-T. Compared with static features that can be directly extracted from APK files, many important features (*e.g.,* `Type` and `Height`) of overlays are dynamically determined at app runtime. To extract them, we automate apps' running using the *Monkey* UI exerciser [15].

In detail, we find that 37% of malicious apps can still launch overlay-based attacks on Andorid 8.0+ in the two-stage manner described above, indicating a non-trivial threat to even up-to-date systems. Further, our results reveal a set of suspicious overlay properties strongly correlated with the *malice* of apps: (1) Overlays are used by 50% of malicious apps, and they intentionally make their overlays difficult to detect. (2) `Type`, `Flags`, and `Format` are the three features with the strongest correlations with an app's malice, while the new `Type` (`TYPE_APPLICATION_OVERLAY`) in Android 8.0+ is ineffective to early detection. (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 the above insights, we then 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 can address the tension between usability and security of previously-proposed solutions [11], [5], [12]. Based on the characterization results of malicious overlays, OverlayChecker collects 56 static and dynamic features from each app. Moreover, we introduce a normalized feature-frequency encoding scheme, which represents features with their normalized occurrence frequencies. Compared with the traditional *One-Hot* encoding used in our preliminary work [16], this encoding scheme is able to retain more fine-grained feature information that is useful in deciding the malice of quite a few apps. Taking this as input, we detect malicious overlay behavior via a *random forest* machine learning model trained with large-scale, ground-truth data provided by Market-T.

To cope with the large amount of app submissions to Market-T per day, we make multifold endeavors to architect an efficient analysis infrastructure. First, we built a lightweight Android emulator that directly runs the Android OS and apps on x86 architecture, coupled with the state-of-the-art *dynamic binary translation* technique [17] to support apps that use Android's native APIs. Also, we adopt GPU-assisted acceleration to expedite the graphic rendering for apps, which intercepts the "micro" graphic instructions (that are reconstructed from OpenGL instructions by the graphic driver) and directly executes them on x86 servers' dedicated GPUs. Further, noticing that static and dynamic analysis are independent of each other and thus can be done in parallel, we design a publish-subscribe pipeline to optimize the overall workflow. With the above efforts, we achieve a speedup of 8–10× compared to using the default emulator [18] in Android SDK.

Further in practice, we observe that the original *Monkey* UI exerciser bears three shortcomings: redundant actions, action loops, and rigid generation of UI events, which could degrade the UI exploration coverage and hinder the exposure of features. We thus design an app-aware UI exploration strategy by additionally taking the specialties of various apps into account. First, we fine-tune the composition of the generated UI events to reduce redundant actions for the specific types of apps. Second, we leverage an app's UI layout structure and component information to guide the automatic UI exploration to avoid action loops. Third, we adaptively tune the generation frequency of certain UI events to prevent apps from identifying the existence of our emulator and thus suppressing their malicious activities. As a result, we manage to achieve a higher (76%→86%) UI exploration coverage with 40% fewer UI events as compared with the default *Monkey*.

Real-world performance and robustness. OverlayChecker has been integrated into Market-T as a part of the app review process since Jan. 2018 and has been constantly optimized. It works under an intense workload of ∼10K app submissions per day using only two commodity servers (we run Android 6.0 and 8.0 on the two servers respectively). The per-app analysis time is ∼1.7 minutes, and OverlayChecker is able to achieve 97% detection precision and 97% recall as of Mar. 2020. 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 and porn-fraud apps, due to their heavy reliance on overlays to launch intended attacks.

Furthermore, we applied OverlayChecker to 10K random apps in Google Play Store on May 1st, 2020, and detected 25 previously unknown apps with malicious overlays that were caught stealing credentials. Although these apps were removed by Google 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 the random forest model used by OverlayChecker to investigate whether attackers can avoid OverlayChecker by adapting their malware's behaviors. By interpreting our model, we show that the behaviors of benign and malicious overlays are sufficiently different, making it 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.

2 Background

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 on top of the host View. Different from the host View which is almost always in a rectangular shape occupying the full screen of the user device, an overlay possesses plentiful 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 intended for the underlying host View if certain flags are specified [16]. In brief, overlay is a powerful UI feature allowing one app to display something on top of others, which can be used to intercept sensitive user input, or even 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 [19] and View parameters [20] (an overlay is an object inheriting the View class), as 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 that 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. Further on Android 8.0+, another SYSTEM_OVERLAY_WINDOW permission should also be requested. More in detail, SYSTEM_OVERLAY_WINDOW overlays originally have 6 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 [21], which can be used to launch serious attacks such as the ransomware attack. With the release of Android 8.0, to facilitate users' comprehensive surveillance on apps' usage of overlays, the overlay mechanism has been essentially updated by unifying the original 6 Types of overlays into a single Type, which also requires users' explicit approval at app startup time to display atop other apps. 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.

Finally, there are two *static* properties at the app level that can amplify the capability of overlays: BIND_ACCESSIBILITY_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 need to 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 [3].

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 the third quarter of 2020, over 2.86 million Android apps released on Google Play Store (the official app store of Android). Market-T, 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, Market-T 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 Market-T 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.

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

Category	Parameters
Appearance	X, Y, Width, Height, Gravity, isOpaque, 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_NOT_FOCUSABLE, ······ (31 in total)
Static	BIND_ACCESSIBILITY_SERVICE, PACKAGE_USAGE_STATS

To accurately determine the malice of hosted apps, Market-T conducts a sophisticated app review process consisting of fingerprint-based antivirus checking, API inspection, and manual examination. Antivirus checking inspects apps against virus fingerprints from antivirus service companies [16]. 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. For those apps whose malice cannot be determined through antivirus checking and API inspection, Market-T assigns security experts to manually examine them with very high precision.

Market-T maintains a large 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, Market-T maintains the category labels of each app (*e.g.,* game, shopping, and education [22]). These labels are predefined by Market-T and are selected by app developers. For malicious apps recorded in the database, Market-T has another set of labels such as ransomware, adware, and porn-fraud apps. These labels provide us with opportunities to understand the motivations and use cases of overlay-based attacks.

2.3 Threat Model

In this work, we assume malicious apps can launch attacks using either SYSTEM_ALERT_WINDOW overlays, or TYPE_TOAST overlays. Here we refer SYSTEM_ALERT_WINDOW overlays to those requesting the SYSTEM_ALERT_WINDOW permissions. Further, the SYSTEM_OVERLAY_WINDOW permission is a special permission on Android 8.0+, and is not granted at runtime. In fact, apps need to explicitly ask users to additionally grant the SYSTEM_OVERLAY_WINDOW permission through the permissions manager in system settings. Smart attackers often employ social engineering approaches to induce users to authorize the SYSTEM_OVERLAY_WINDOW permission, by purposely requesting a set of common permissions (*e.g.,* to use camera, sensors, location, and SMS) at app startup time. In any case, the attacks can be successfully launched without root privileges. We assume that malicious apps can use overlays in any form (see Figure 1), thus confusing the users to misinterpret UI interactions, luring the users to type passwords or grant certain permissions, and so on. Further, adversaries can (optionally) launch more effective overlay-based attacks by acquiring certain permissions like BIND_ACCESSIBILITY_SERVICE and PACKAGE_USAGE_STATS, or by inferring the UI states in certain ways like the shared-memory side channel.

We assume that attackers have adopted techniques to obfuscate their malicious apps and frustrate analysis. Specifically, using obfuscation techniques [23], application pacing platforms can be employed to alter the structure of malware code and hide the use of malicious overlays, which can help malware spoof and bypass detection mechanisms such as signature matching and static analysis. Also, malicious apps can use various methods to detect whether they are running in emulation environments. For instance, malware can recognize the processor architecture (ARM or x86) by inspecting cache behavior and system configurations, and identify the existence of automated UI exerciser from user actions. If a malicious app notices the existence of an emulator, it may dynamically change its behavior accordingly.

Moreover, we assume the goal of the adversary to be distributing malware through a major app store. This is a powerful mode of attack, since inclusion in an app store "lends credibility" to the malicious app. Additionally, most Android smartphones are configured to only allow apps to be installed from app stores by default. We do not consider attacks where malicious apps are distributed outside of app stores, since mitigating these "sideloading" attacks requires client-side defenses that are beyond our scope.

3 Understanding Overlay Behavior

3.1 App Dataset

This section presents our analysis of overlay behavior of malicious and benign apps. Our raw dataset contains all the new and updated apps submitted to Market-T during Jan. 2017–Sep. 2019. After removing redundant apps (*i.e.,* APKs with the same MD5 hash value), we are left with a total of ∼550K apps as our dataset, and the entire dataset maintained by Market-T includes a huge number of apps with obsolete overlay usage. For every app in the dataset, Market-T provides not only its APK file but also its malice and category labels. Nearly one third (31%) of the apps are labeled malicious and thus are quarantined in Market-T'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 Market-T adopts a rather sophisticated and effective app review process, we believe the false positive rate in this labeling is statistically insignificant [16] and thus has negligible impact on our subsequent analysis and system design.

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 run time. In Android, all overlays are 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. Simultaneously, we can extract 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 at run time when the app is executed, and their number is much larger than that of static features. In addition to those original features defined in the Android SDK, we design four novel features. We will detail these features (refer to Table 1) 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 [15] 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 details) for later analysis. We execute apps and the *Monkey* tool on Android emulators deployed on a commodity x86 server. However, we do not use the default Google's Android device emulator included in the Android SDK [18]. 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 Android emulator that directly runs the Android OS and apps on x86 architecture. First, as for the Android OS we use Android-x86 [24], 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* (DBT) based on Intel Houdini [17] to translate the ARM instructions into x86 instructions (most dynamic libraries in Android are based on the ARM ISA instead of x86). 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 8 Android 6.0-based emulators and 8 Android 8.0-based emulators 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. Meanwhile, when executing *Monkey*'s generated UI events for an app, we use Xposed [25] to intercept the addView API' invocation data (including its name and parameters).

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 identify emulators [26], [27]. To prevent adversarial detection, we made four improvements to our emulators to make them behave more consistently with real devices and users. First, we alter the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2021.3079433, IEEE Transactions on Mobile Computing

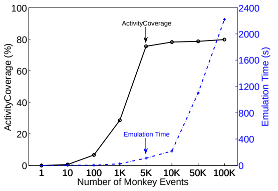

Fig. 2: Relationship between the emulation time, `ActivityCoverage`, and number of *Monkey* events.

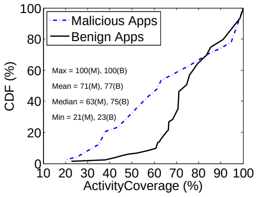

Fig. 3: `ActivityCoverage` for all malicious (M) and benign (B) apps when 5K *Monkey* events are executed.

default configurations of 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 [15]. For example, we tune the `throttle` parameter with a reasonable value, 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, proximity, orientation, and so forth) collected from a number of real smartphones on our emulators to improve fidelity [27]. Finally, we customize and obfuscate the relevant libraries of Xposed, so that the studied apps can hardly detect Xposed's hooking behavior [28].

3.2.2 UI Exploration

Detecting malicious overlay behavior requires a high UI coverage to examine as many overlays. 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, as 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 and analyze the configuration file and the static code of non-obfuscated APKs in our dataset. The scanning results show that for an average app, 88% of its specified `Activities` can be actually referenced. Thus, 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 referenced `Activities` in the configuration file.

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. However, the abovementioned 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.,* Market-T 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 Market-T, 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 [29]. Figure 2 shows the `ActivityCoverage` achieved with increasing running time of *Monkey*. 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* [30]) 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 detect 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 30% 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 Market-T (whose labeling process is detailed in §3.1 and §2.2), we find that overlays are being used by ∼50% of malicious apps but only ∼27% of benign apps. In the meanwhile, we notice that 37% of malicious apps can still launch overlay-based attacks on Android 8.0+ in the two-stage manner mentioned in §1.

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. In our dataset, 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 [16]. 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

3.4.1 Understanding Static Features

`BIND_ACCESSIBILITY_SERVICE.` This permission is granted for accessibility services specially designed to assist disabled Android users [31]. 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 [32]. In our dataset, 1.3% of apps utilize accessibility services, among which 2.5% are malicious. In particular, 0.12% of apps utilize both accessibility services and `SYSTEM_ALERT_WINDOW` overlays, among

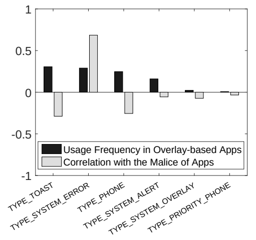

Fig. 4: Frequently used Types and their *PCC*s with the malice of apps before Android 8.0 was released.

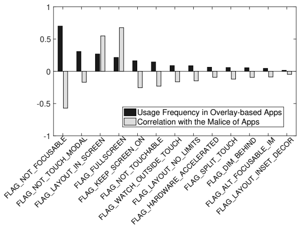

Fig. 5: Usage frequency of Flags and their *PCC*s with the malice of apps.

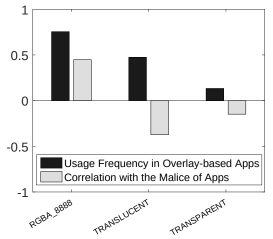

Fig. 6: Frequently used Formats and their *PCC*s with the malice of apps.

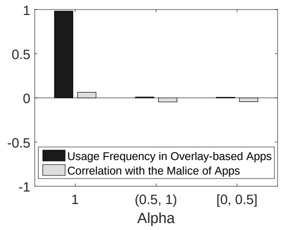

Fig. 7: Usage frequency of different ranges of Alpha, and their *PCC*s with the malice of apps.

which 1.6% 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.

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. Analysis on our dataset shows that 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* Flag

Type. An overlay can have a total of 16 Types before Android 8.0 was released, among which the six Types listed in Figure 4 are the most frequently used in Android 6.0∼Android 7.1. 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 high priority enabling overlay to appear on top of all activity windows, even the lock screen interface [21], which gives a chance to serious overlay-based attacks. Thereby, the six most frequently used Types are deprecated and unified into a new TYPE (TYPE_APPLICATION_OVERLAY) in Android 8.0+ to facilitate users' comprehensive surveillance on apps' usage of overlays. Since the new TYPE is used by both benign and malice apps, it is ineffevtive for detecting malware and therefore is not considered as a key fearture. Moreover, compared with the TYPE_SYSTEM_ERROR overlay, the new TYPE overlay is limited below critical system windows like the status bar or the lock screen, and can only appear atop other activity windows after the explicit authorization of users.

Flag. We study all the 31 Flags of Android overlays. Figure 5 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 9.5% of these apps are malicious and the corresponding *PCC* is as low as -0.55. 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, 85% of the apps that use FLAG_FULLSCREEN overlays are malicious and the *PCC* is as high as 0.7. This is because a FLAG_FULLSCREEN overlay can cover the whole screen (including the status bar) and thus can easily deceive mobile users.

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 6, 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 7, 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 numerous overlays, 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 calculated by dividing the intersection area by the area of the host View. A more complex case is in Figure 8—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. In practice, we find that overlays' VisualCoverage exhibits distinct distributions between malicious and benign apps [16]. 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 (0.4).

Y and Gravity. We further consider each overlay's VisualCoverage *scope*, denoting the host View's geometric scope visually covered by the overlay. Figures 9 and 10 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

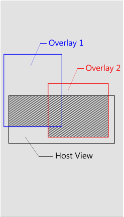

Fig. 8: Our calculation of the VisualCoverage for multiple overlays.

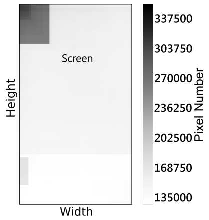

Fig. 9: Heat map of the VisualCoverage scopes for benign apps' overlays.

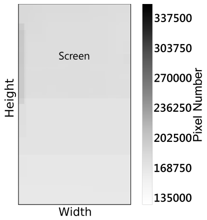

Fig. 10: Heat map of the VisualCoverage scopes for malicious apps' overlays.

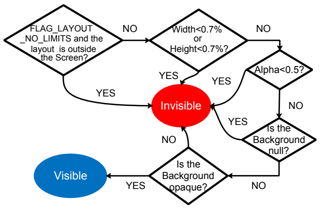
FLAG_LAYOUT _NO_LIMITS and the layout is outside the Screen?
NO
Width<0.7% or Height<0.7%?
NO
Alpha<0.5?
YES
YES
YES
Invisible
NO
YES
Is the Background null?
NO
Visible
YES
Is the Background opaque?
NO

Fig. 11: Flow chart for our calculation of isReallyVisible for an overlay.

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. According to our manual observations, 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 for our calculation of isReallyVisible is plotted in Figure 11. Among all the apps that have used overlays, 33% of malicious apps and 13% of benign apps are using overlays that are not really visible, showing the significance of isReallyVisible in detecting malicious overlays.

3.5 Summary of the Study Results

Our comparative study leads to a series of useful insights with respect to malicious overlay behavior: (1) Overlays are used by more than 30% of Android apps overall in our dataset, and 50% of malicious apps. (2) 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. (3) We observe malicious apps intentionally make it hard for dynamic analysis tools to detect their overlays. (4) Type, Flag, and Format are the three features that correlate most strongly with an app's malice, while the new Type in Android 8.0+ is less effective in early detection, *e.g.,* 84% of the apps that use TYPE_SYSTEM_ERROR overlays are malicious (*PCC* = 0.69) before Android 8.0 was released, however it should be noted that there becomes a single new Type overlay in Android 8.0+; 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). (5) We design a complex feature VisualCoverage that reveals distinct distributions between the overlays of malicious apps and benign apps. (6) 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 System Design and Development

4.1 System Design and Implementation

Guided by our study results in §3, we build OverlayChecker to detect overlay-based malware and evaluate its efficacy.

Overview and workflow. As shown in Fig. 12, once an app is submitted, OverlayChecker first leverages DBT to enable the app to run efficiently on x86 environments (①). To automate various app behaviors, we use the *Monkey* exerciser to automatically generate UI event streams (②). When running the app, multiple defensive interventions are also implemented to prevent intentional detection evasion (③). Next, we extract the app's requested permissions from its metadata (④⑤), and use Xposed to capture the invocation of the addView API during the app's execution (⑥). As a result, we distill the key overlay features as identified in §3 from the above logged data (⑦). The selected features are then encoded in a normalization manner (⑧), and piped into a machine learning classifier (*e.g.,* random forest) to determine the app's malice (⑨). In the remainder of this section, we will describe OverlayChecker's emulation infrastructure used for dynamic and static analyses, as well as feature engineering techniques and model design in detail.

Emulation infrastructure. As introduced in §3.2.1, OverlayChecker builds its app emulation environment atop a customized lightweight Android emulator to extract dynamic features of overlays. Here we focus on further enhancing the underlying emulation infrastructure to cope with several real-world challenges.

In detail, we find that the average ActivityCoverage of the original *Monkey* exerciser is only 76%, which inevitably omits some malicious overlays displayed during the app running process. To better expose and embody overlay features, we further advance our automatic UI exploration methodology targeting *Monkey*'s three major drawbacks – redundant actions, action loops [33], and fixed-rate generation of UI events, which could degrade UI coverage and thus hinder feature exposure.

First, in practice we notice that portions of different UI events vary with apps [34], so the uniformly set composition of UI events tends to reduce the ActivityCoverage of many apps due to redundant actions. We thus fine-tune the composition of the generated UI events (*e.g.,* the portion of touch events) to reduce redundant actions according to the specific types of apps (*e.g.,* shopping or news feed apps). Here we perform category-level rather than app-level analysis for parameter settings given that many apps are obfuscated and thus may incur high analysis workload. Second, we find that action loops root in the random nature of *Monkey*'s generated events, which is inherently limited

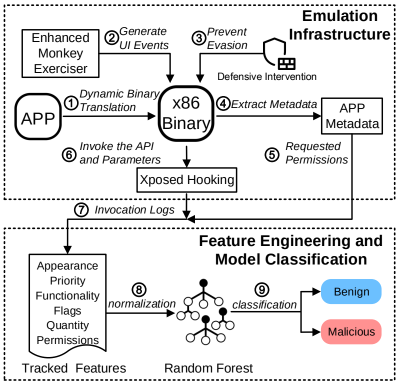

Fig. 12: Architectural overview of OverlayChecker.

due to the lack of information regarding an app's interactable UI components and visited `Activities`. With the UI Automator [35], we leverage an app's UI layout structures as heuristics for triggering actions and `Activities`, as well as record visited `Activities` to detect and avoid severe action loops. Third, since the original UI event generation rate is fixed, we note that the malware can take the interval as a critical indicator of emulation and detection; if the interval is smaller than a set threshold, they would suppress malicious activities (become idle). Hence, once an app is constantly idle during emulation without responding to input events (indicating that the app may have recognized the emulator), we exponentially increase the interval from 500 ms to quickly reach an ideal interval. Note that we stop increasing the interval once the overall waiting is over 2 minutes to avoid significant overhead. If the waiting time exceeds 2 minutes when checking an app, it is then considered to be highly suspicious and submitted for further manual inspection. As a result, we manage to achieve a higher (76%→86%) UI exploration coverage with 40% fewer UI events; more specifically, 99.6% apps exhibit the same set of activities on our emulators as on physical devices.

We note that although fine-grained UI tests bring a higher UI exploration coverage, the execution of more activities increases the emulation time by an average of ∼15%, which may be undesirable for both app developers and the markets. Therefore, we introduce diversified hardware-assisted virtualization techniques with Android-x86 into the underlying runtime, such as Intel VT [36] and KVM [37], to enable our system to fully explore the power of x86 CPUs. In addition, to further improve the performance, VirtIO [38], a para-virtualization technique, is adopted to accomplish GPU-assisted acceleration in graphic rendering which is formerly excuted by the host CPU. Specifically, we first intercept "micro" instructions from the guest (Android-x86) side graphic driver (that disintegrate and reassemble the "micro" instructions from OpenGL instructions) within apps' rendering pipelines, and then execute them atop the GPU on the host (x86 server) side [39], thus essentially surpassing the original CPU-based software rendering. With these efforts [40], we are able to reduce the average scan time per app by around 10% as shown in Fig. 14.

Feature engineering. Having optimized the emulation environment, we then shift our focus to feature engineering. Preliminarily, we adopt the traditional One-Hot scheme to encode extracted features, where 1 denotes the occurrence of the corresponding feature and 0 denotes otherwise. Though yielding generally fine performance (96% precision and 96% recall), we observe that One-Hot encoding has a strong dependency on some key features, resulting in that the malware can accordingly evade detection by not using them, leading to false negatives. Noticing that the problem is mainly caused by the orthogonal and binary dimensions of the One-Hot vector which lacks some essential information, we introduce a new feature-frequency encoding scheme to address this problem by retaining more fine-grained feature information.

TABLE 2: Efficacy of ML algorithms using 56 overlay features *vs.* the 52 original overlay features (excluding our four novel features).

Algorithm	Precision (56 / 52)	Recall (56 / 52)	Training Time (56 / 52) second
Naive Bayes	0.89 / 0.86	0.75 / 0.75	2 / 2
LR	0.88 / 0.85	0.87 / 0.84	6 / 6
Random Forest	0.97 / 0.94	0.97 / 0.94	35 / 35
DNN	0.95 / 0.93	0.95 / 0.93	79 / 75
XGBoost	0.94 / 0.92	0.95 / 0.91	83 / 82
RBF-SVM	0.95 / 0.91	0.95 / 0.91	1626 / 1625
Linear-SVM	0.93 / 0.91	0.93 / 0.91	11551 / 11542
Poly-SVM	0.93 / 0.89	0.92 / 0.89	59294 / 58762

In detail, as shown in §3, we note that there are significant differences of the occurrence frequencies of key features (*e.g.,* `Type`, `Flags`, `Format` and `Alpha`) between benign and malicious apps, which can be contained to expand the dimension. Therefore, the feature-frequency encoding scheme is an efficient approach to obtain ample information. In practice, instead of using 1 or 0 to denote whether the corresponding feature is used or not, the new scheme replaces the homologous bit with the occurrence frequency of each feature. However, experiments show that prominent discrepancies exist among the occurrence frequencies of different features. In other words, features that appear more frequently than others may dominate the attention of machine learning models, thus impairing the performance of the current encoding scheme. To address the issue, we devise an enhanced strategy that transforms the occurrence frequency of each feature into a normalized value, eliminating the influence caused by the diversified occurrence frequencies among features.

Malice detection. Through the normalized feature-frequency encoding scheme, we transform the logs into an $m \times n$-dimension vector, where m is the number of samples, and n is the total number of extracted features. Next, we pipe the engineered feature vector into eight machine learning models (as listed in Table 2) to compare their performance (in terms of precision and recall) and overhead (in terms of training time). To reduce the impact of possible data leakage, which may result in overestimated evaluation results, we leverage 10-fold cross-validation when evaluating precision and recall. In Table 2 we present the comparison results of the eight 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. 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 when the precision/recall is already very high (>90%), thus demonstrating their utility.

We also notice that the evaluated models' advantages lie in different aspects, while no single model outperforms others in all the metrics. In particular, tree-based models (RF and XGBoot) and neural network model (DNN) benefit from the rather skewed distribution of most features in the dataset (*e.g.,* regarding occurrence frequency). However, performance metrics of the DNN model are accompanied by the expense of overfitting, due to its complexity internal network structures. In contrast, the ensemble learning technique of random forest (RF) integrates the power of multiple trained models, and largely enhances its generalization ability,

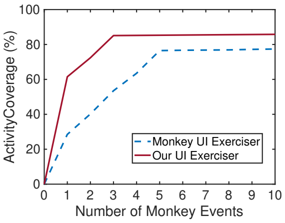

Fig. 13: Comparison of *Monkey* UI exerciser with our enhanced UI exerciser.

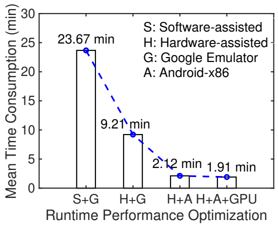

Fig. 14: Effects of various optimization technologies. “+” means combination.

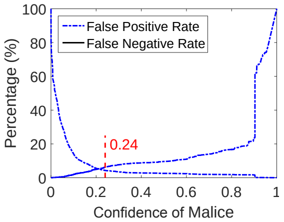

Fig. 15: Minimize false positive + negative rate by tuning the threshold for CoM.

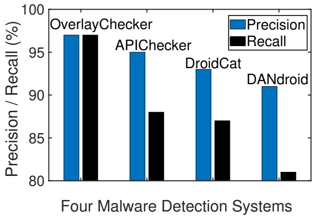

Fig. 16: Performance comparison with state-of-the-art malware detection systems.

thus effectively reducing the performance degradation incurred by overfitting. Also, the model’s simplicity (as compared to complex network models) and the parallel nature of RF’s internal trees make the training process rather efficient. Consequently, the RF model is selected, producing the best precision (97%), best recall (97%), and an acceptable training time.

In practice, to label newly submitted apps as malicious (M) or benign (B) in terms of overlay behaviors, OverlayChecker uses a three-step process. *First*, OverlayChecker quantifies the malice of each detected overlay in a given app as a *confidence* value [16], denoted as *CoM* (Confidence of Malice), between 0 and 1.0 using the classification model. *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 Market-T, we configure the confidence threshold as 0.24 (Malicious: >0.24, Benign: ≤ 0.24) to minimize the false positive and negative rates, as illustrated in Figure 15.

Moreover, we use our dataset to compare OverlayChecker with three recent representative malware detection systems (*i.e.,* APIChecker [22], DroidCat [41], and DANdroid [42]). Based on our collected dataset, we then compare their detection accuracies with OverlayChecker’s, finding that OverlayChecker behaves the best as shown in Fig. 16. Furthermore, we inspect the false positives and negatives of the three researches, observing that the erroneous judgements are mostly because the relevant apps are leveraging other means to conduct attacks rather than overlays. To be more specific, the three researches all pay special attention to common malicious behavior and features (*e.g.,* APIs, intents and permissions). In fact, we note that overlay-based malware has little to do with such common features, making the detection algorithms designed in the three researches incompetent. In contrast, OverlayChecker concentrates on the key features of overlays, and thus is capable of effectively discovering abnormal overlays (which are usually malicious) in Android apps.

4.2 System Deployment and Performance

Distributed deployment. Each app’s analysis is originally comprised of nine steps as shown in Fig. 12, among which several steps do not rely on each other indeed and thus can be executed in parallel. Thereby, we reshuffle the execution sequence of the analysis steps, as depicted in Fig. 17. We introduce a loosely-coupled pipeline instead of the original monolithic back-to-back execution manner, which in detail consists of three components – dynamic analysis, static analysis, and model classification, working together to implement a publish-subscribe system. Specifically, submitted tasks are extracted from a message queue and published to listening channels. Then dynamic and static analysis channels ballot for unfinished tasks (*i.e.,* tasks not being analyzed by both the dynamic and static analysis components) to perform app emulation (corresponding to Step ①②③⑥ in Fig. 12) and metadata extraction (Step ④⑤ in Fig. 12), respectively. When all the above analysis tasks are finished, the classification model can then utilize the collected feature data to determine an app’s malice (Step ⑨ in Fig. 12). As a result, we manage to shorten the average per-app scan time by ∼10%.

Integration to a real app market. OverlayChecker has been integrated into Market-T as a part of app review process since Jan. 2018. It explicitly marked potentially malicious apps with an *“Overlay Risk”* annotation. As of Jun. 2018, OverlayChecker was still able to achieve 96% precision and 96% recall, with the per-app analysis time being ∼1.7 minutes on average. Market-T presented this annotation to users when they viewed the app in the store and encouraged them to take appropriate precautions, *e.g., “This app is using risky overlays (confidence* = 0.74*), please disable its overlay permission at once!”*. We have continuously updated 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. Noticing that all apps in the dataset were forward-compatible (probably because that as of Sep. 2019 there were still 40% smartphones not being upgraded to Android 8.0+), we deploy OverlayChecker on a single commodity x86 server (refer to §4.1 for the detailed configurations) running Android 6.0. As a result, OverlayChecker is able to check ∼10 K apps submitted to Market-T per day.

Unfortunately, Market-T reported that OverlayChecker’s detection performance had exhibited a constant trend of degradation since Jun. 2019 [16] – for example, the precision decreased to 92% and the recall decreased to 90% in Oct. 2019, probably because some overlay-based malware that could bypass the updated overlay mechanism in Android 8.0 proactively abandoned the forward compatibility with the original mechanism in Android 6.0 to intentionally evade the detection of OverlayChecker. To this end, we have upgraded the runtime environment of OverlayChecker since Oct. 2019, where each submitted app was analyzed in parallel on the original Android 6.0-based emulator and the additional Android 8.0-based emulator (so two commodity servers are required now). Finally, OverlayChecker can achieve essentially 97% precision and 97% recall at Market-T as of Mar. 2020. We have provided the up-to-date performance results of the production system within 12 months, as shown in Fig. 18.

To understand the 3% false positives of the random forest model, we manually inspected the detection logs and found that the 97% precision does *not* actually mean 3% *errors* in the classification results. In fact, although we determined that the 3% false positives are benign apps, 97% of them had irregular

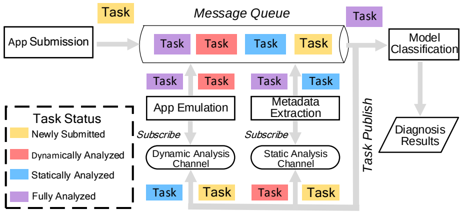

Fig. 17: OverlayChecker's distributed deployment in Market-T.

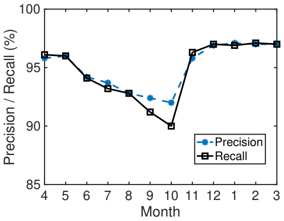

Fig. 18: Online performance of OverlayChecker over 12 months.

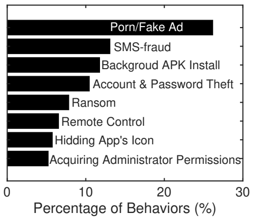

Fig. 19: Distribution of common overlay-based malicious behaviors.

overlay behaviors, because some app developers abuse overlays, particularly using several TYPE_SYSTEM_ERROR overlays or FLAG_FULLSCREEN overlays to exhibit certain content (*e.g.,* advertisements). The developed apps often have top-ranking features [16] (with relatively high Gini importance) that are highly correlated with malware. As a matter of fact, the 3% false positive rate is considered acceptable by Market-T, as apps flagged by OverlayChecker receive a 2nd-round manual review. Specifically, among the nearly 10K apps submitted per day, there are usually about 1100 apps flagged by OverlayChecker, among which nearly 1020 are meanwhile labeled as malicious by Market-T (using its own high-precision security checking mechanisms, refer to §2.2). Thus, OverlayChecker only has around 80 apps requiring manual review, and only around a couple of them are false alarms.

As we observe in §3.3, overlays are used by 50% of malicious apps, so the high recall (97%) illustrates that OverlayChecker is able to detect nearly half (48.5%) of *all* malicious apps hosted on Market-T using solely automated overlay behavior analysis. Specifically, using the category labels of apps maintained by Market-T (§2.2), OverlayChecker can detect the vast majority (90+%) of certain types of malicious apps, *e.g.,* 99% of ransomware, 98% of adware, 94% of porn-fraud, and 92% 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).

Important features and malicious behaviors. By exploring the Gini [43] indices of each tracked features, which is a prevalent metric derived from a trained random forest model to evaluate feature importance, we understand key features most essential to our malware detection model. Here, we use the Gini [43] importance to evaluate important features in our trained random forest model. Overall, the results [16] 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 in Android 6.0. 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} in terms of Gini importance. Among the 56 features, only two (PACKAGE_USAGE_STATS and BIND_ACCESSIBILITY_SERVICE, detailed in §3.4.1) are static and their importances ranked only 12^{th} and 44^{th}. As discussed in §3.2, this is because many important characteristics of overlays only exhibit dynamically at app runtime [16]. This thus concretely shows that it is necessary to consider dynamic features to ensure high detection effectiveness.

Furthermore, Fig. 19 illustrates the distribution of the overlay-based malicious behaviors. As shown, porn/fake adware attack accounts for the largest portion, while malware launching specific attacks such as remote control app attacks is much less pervasive. For app categories, gaming apps are most likely to manifest malicious behaviors (∼14% are malicious), which tend to displaying porn/fake ads, and stealing users' accounts and passwords. In addition, office and system apps usually induce users to give them administrator permissions.

4.3 Extensibility

OverlayChecker is not limited to Market-T 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 apps. Further, although the construction of our classification model relies on the app dataset provided by Market-T, once the model is trained OverlayChecker can work independently of Market-T 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 in Google Play Store on May 1st, 2020. Despite Google's own sophisticated security checking, OverlayChecker is still able to detect 25 (0.25%) apps with malicious overlay behavior. We note that the malicious apps can overlay a web browser window on top of the other apps and load a fake login page to stole users' credentials.

Interestingly, we observed that these apps were removed from Google Play Store in early June, 2020, potentially due to reports from users and researchers. However, since these apps were officially available, users who installed them were potentially vulnerable for over a month. This indicates the pressing need today for an effective and efficient market-scale early detection system.

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 the classification model employed by OverlayChecker (*e.g.,* by trial-and-error attempts).

In particular, the robustness of OverlayChecker is derived from the random forest classification model. The organization of the trained classifier precludes attackers from adopting the vast majority of malicious techniques in their overlays. For example, the most famous class of overlay-based attacks, "cloak and dagger" attacks [11], can only go through a fixed decision path [16], which always results in a malicious classification in our model. An attacker seeking a benign label 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 97% precision/recall in §4.2). Since the model is updated frequently, such classification logic will become even more restricted after considering more malicious overlay behaviors in the future.

5 Related Work

Overlay-based attacks. The most direct overlay-based attacks construct deceptive overlays, confusing users to misinterpret UI interactions. Figure 1 classifies such attacks into five groups. *First*, as shown in Figure 1(a), malicious *redressing* overlays can be constructed to impersonate small UI widgets (*e.g.,* buttons) as a part of the current UI window, thus triggering users to click [3]. *Second*, malicious transparent overlays, as shown 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) [4], [5], [7]. Malicious transparent overlay attacks can also be launched through `WebView` in Android to compromise web content [8]. *Third*, as shown in Figure 1(c), malicious *hollow-out* overlays selectively uncover UI components of victims apps, misleading users over the meaning of the interaction by manipulating the covered overlay [3]. *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) [9], [11]. *Finally*, malicious overlays *outside the screen* (in Figure 1(e)) cannot be noticed by users, but can still maliciously capture UI events. 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 [44]. This empowers non-trivial overlay-based attacks based on users' tapping behavior.

Attack defenses. Bianchi *et al.* propose an on-device defense (known as WhatTheApp) that adds a 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 [5]. However, WhatTheApp is vulnerable to timing attacks because the security indicator is calculated periodically—a malicious overlay can be inserted within the period. Furthermore, attackers can bypass the periodic check by rendering a malicious overlay on top of the victim app and then quickly hiding it. To fix this problem, Overlay Mutex was proposed to prevent a background non-system app from rendering on top of any foreground apps [7].

Moreover, dynamic approaches for capturing malicious overlays at runtime have been proposed [45]. However, as these approaches use runtime monitors, they incur considerable user-side resources (*e.g.,* CPU and battery usage). Additionally, the alert windows for reporting malicious overlays can themselves be attacked by malware. DECAF and OverlayChecker 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.

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 alternatively exploit the original overlay mechanisms on Android 6.0 or the updated overlay mechanisms on Android 8.0+ to launch overlay-based attacks. 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 Market-T, 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 with multi-fold systematical efforts to quickly and automatically detect overlay-based malicious apps with high precision and recall. OverlayChecker is integrated into Market-T 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.

7 Acknowledgment

This work is supported by the National Key R&D Program of China under grant 2018YFB1004700, NSF of China under grants 61902211, 61972313, 61822205, 61632020 and 61632013, NSF of Tianjin under grant 18JCQNJC69900, and the Postdoctoral Science Fund of China under grant 2019M663725.

References

[1] "Multi-Window Support for Android," https://developer.android.com/guide/topics/ui/multi-window.html.

[2] "How to use split screen mode multi-window in android nougat," http://gadgetguideonline.com/android/android-nougat-guides/how-to-use-split-screen-mode-multi-window-in-android-nougat-7-07-1/.

[3] "Most Android Devices Prone to Accessibility Clickjacking Attacks," http://www.securityweek.com/most-android-devices-prone-accessibility-clickjacking-attacks.

[4] L. Ying, Y. Cheng, Y. Lu, Y. Gu, P. Su, and D. Feng, "Attacks and Defence on Android Free Floating Windows," in *Proc. of ACM ASIA CCS*, 2016.

[5] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the App is That? Deception and Countermeasures in the Android User Interface," in *Proc. of IEEE S&P*, 2015.

[6] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into Your App without Actually Seeing it: UI State Inference and Novel Android Attacks," in *Proc. of USENIX Security*, 2014.

[7] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash, "Android UI Deception Revisited: Attacks and Defenses," in *Proc. of Springer FC*, 2016.

[8] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking Attacks on Web in Android, iOS, and Windows Phone," in *Proc. of Springer FPS*, 2012.

[9] E. Ulqinaku, L. Malisa, J. Stefa, A. Mei, and S. Capkun, "Using Hover to Compromise the Confidentiality of User Input on Android," in *Proc. of ACM WiSec*, 2017.

[10] "Cloak & Dagger," https://cloak-and-dagger.org/.

[11] Y. Fratantonio, C. Qian, S. Chung, and W. Lee, “Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop,” in *Proc. of IEEE S&P*, 2017.

[12] E. Fernandes, Q. A. Chen, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash, “TIVOs: Trusted Visual I/O Paths for Android,” University of Michigan, Ann Arbor, Tech. Rep. CSE-TR-586-14, May 2014.

[13] “Android’s overlay attacks in 2020 ,” https://labs.f-secure.com/blog/how-are-we-doing-with-androids-overlay-attacks-in-2020/.

[14] “Tencent App Market,” https://sj.qq.com/.

[15] “UI/Application Exerciser Monkey in Android Studio,” https://developer.android.com/studio/test/monkey.html.

[16] Y. Yan, Z. Li, Q. A. Chen, C. Wilson, T. Xu, E. Zhai, Y. Li, and Y. Liu, “Understanding and Detecting Overlay-based Android Malware at Market Scales,” in *Pro. of ACM MobiSys*, 2019.

[17] “Android-x86 Vendor Intel Houdini,” https://osdn.net/projects/android-x86/scm/git/vendor-intel-houdini/.

[18] “Google Android Emulator shipped with Android Studio,” https://developer.android.com/studio/run/emulator.html.

[19] “Layout parameters for Android overlays,” https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html.

[20] “Android Developer Documentation for View properties,” https://developer.android.com/reference/android/view/View.html.

[21] “Android Developer Documentation for TYPE_SYSTEM_ERROR,” https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#TYPE_SYSTEM_ERROR.

[22] L. Gong, Z. Li, F. Qian, Z. Zhang, Q. A. Chen, Z. Qian, H. Lin, and Y. Liu, “Experiences of Landing Machine Learning onto Market-scale Mobile Malware Detection,” in *Proc. of ACM EuroSys*, 2020.

[23] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, “Understanding Android Obfuscation Techniques: A Large-scale Investigation in the Wild,” in *Pro. of SecureComm*, 2018.

[24] “Android-x86 - Porting Android to x86,” http://www.android-x86.org/.

[25] “XposedBridge,” https://github.com/rovo89/XposedBridge/wiki/Development-tutorial.

[26] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, “Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware,” in *Proc. of ACM EuroSec*, 2014.

[27] S. Dey, N. Roy, W. Xu, and S. Nelakuditi, “Leveraging imperfections of sensors for fingerprinting smartphones,” *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 17, no. 3, pp. 21–22, 2013.

[28] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, “Spotless sandboxes: Evading Malware Analysis Systems using Wear-and-tear Artifacts,” in *Proc. of IEEE S&P*, 2017.

[29] B. Liu, S. Nath, R. Govindan, and J. Liu, “DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps,” in *Proc. of USENIX NSDI*, 2014.

[30] “Pearson correlation coefficient,” https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

[31] “Android Developer Documentation for the Accessibility,” https://developer.android.com/guide/topics/ui/accessibility/index.html.

[32] Y. Jang, C. Song, S. Chung, T. Wang, and W. Lee, “A11y Attacks: Exploiting Accessibility in Operating Systems,” in *Proc. of ACM CCS*, 2014.

[33] Y. L. Arnatovich, M. N. Ngo, T. H. B. Kuan, and C. Soh, “Achieving High Code Coverage in Android UI Testing via Automated Widget Exercising,” in *Proc. of IEEE APSEC*, 2016.

[34] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, “PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps,” in *Proc. of ACM MobiSys*, 2014, pp. 204–217.

[35] “UIAutomator,” https://developer.android.com/training/testing/ui-automator.

[36] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, “Intel Virtualization Technology,” *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[37] C. Dall and J. Nieh, “KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor,” 2016.

[38] R. Russell, “Virtio: Towards a De facto Standard for Virtual I/O Devices,” *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 95–103, 2008.

[39] L. Gong, H. Lin, Z. Li, F. Qian, Y. Li, X. Ma, and Y. Liu, “Systematically Landing Machine Learning onto Market-Scale Mobile Malware Detection,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1615–1628, 2020.

[40] Q. Yang, Z. Li, Y. Liu, H. Long, Y. Huang, J. He, T. Xu, and E. Zhai, “Mobile Gaming on Personal Computers with Direct Android Emulation,” in *Proc. of ACM MobiCom*, 2019.

[41] H. Cai, N. Meng, B. Ryder, and D. Yao, “Droidcat: Effective Android Malware Detection and Categorization via App-level Profiling,” *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2019.

[42] S. Millar, N. McLaughlin, J. Martinez del Rincon, P. Miller, and Z. Zhao, “DANdroid: A Multi-View Discriminative Adversarial Network for Obfuscated Android Malware Detection,” in *Proc. of ACM CODASPY*, 2020.

[43] L. Breiman, J. Friedman, R. Olshen, and C. Stone, “Classification and regression trees,” *Encyclopedia of Ecology*, vol. 40, pp. 582–588, 1984.

[44] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia, “PlaceRaider: Virtual Theft in Physical Spaces with Smartphones,” in *Proc. of ISOC NDSS*, 2013.

[45] L. Wu, B. Brandt, X. Du, and B. Ji, “Analysis of Clickjacking Attacks and An Effective Defense Scheme for Android Devices,” in *Proc. of IEEE CNS*, 2016.

Liangyi Gong received the B.S. and Ph.D. degrees from the School of Computer Science and Technology, Harbin Engineering University, China, in 2010 and 2016 respectively. He is a post-doctoral researcher at the School of Software and BNRist, Tsinghua University. His research interests are in the areas of network security and mobile computing.

Zhenhua Li received the B.S. and M.S. degrees from Nanjing University in 2005 and 2008 respectively, and the Ph.D. degree from Peking University in 2013, all in computer science and technology. He is an Associate Professor with the School of Software and BNRist, Tsinghua University. His research areas cover mobile cloud computing and future networking. He is a member of the IEEE and ACM.

Hongyi Wang is working towards the B.S. degree at the Department of Electronic Engineering, Tsinghua University. Her research areas mainly include cloud computing, mobile computing, and future networking.

Hao Lin received the B.S. degree from the School of Software, Tsinghua University in 2020. He is working towards the Ph.D. degree also at the School of Software, Tsinghua University. His research areas mainly include network measurement and mobile systems.

Xiaobo Ma is currently an associate professor with MOE Key Lab for Intelligent Networks and Network Security, Faculty of Electronic and Information Engineering, Xi’an Jiaotong University, Xi’an, China. He is also with Shaanxi Province Key Laboratory of Computer Network, Xi’an, China. He is XJTU Tang Scholar, a member of IEEE, and served as a co-chair of ACM CoNEXT 2019 Student Workshop. His research interests include cyber security.

Yunhao Liu received the B.S. degree in Automation Department from Tsinghua University, an M.S. and a Ph.D. degree in Computer Science and Engineering from Michigan State University. He is now a Professor and Dean of Global Innovation Exchange at Tsinghua University. His research interests include sensor network and IoT, RFID, distributed systems, and cloud computing. He is Fellow of the IEEE and ACM.