DroidPill: Pwn Your Daily-Use Apps

Chaoting Xuan[†], Gong Chen[‡], Erich Stuntebeck[†]
[†]VMWare
[‡]Georgia Institute of Technology
{cxuan, estuntebeck}@vmware.com, gchen@ece.gatech.edu

ABSTRACT

Nowadays, attacking and defending Android apps has become an arms race between black hats and white hats. In this paper, we explore a new hacking technique called the *App Confusion Attack*, which allows hackers to take full control of benign apps and their resources without device rooting or privilege escalation. Conceptually, an App Confusion Attack hijacks the launching process of each benign app, and forces it to run in a virtual execution context controlled by hackers, instead of the native one provided by the Android Application Framework. This attack is furtive but has dangerous consequences. When a user clicks on a benign app, the malicious alternative can be loaded and executed with an indistinguishable user interface. As a result, hackers can manipulate the communication between the benign app and the OS, including kernel and system services, and manipulate the code and data at will. To demonstrate this attack, we build *DroidPill*, a framework for malware creation that employs the app virtualization technique and the design flaws in Android to achieve such attacks with free apps. Our evaluation results and case studies show that DroidPill is practical and effective. Lastly, we conclude this work with several possible countermeasures to the App Confusion Attack.

Categories and Subject Descriptors

I.3.2 [**Security and Privacy**]: Intrusion/Anomaly Detection and Malware Mitigation—*Social Engineering Attacks*

Keywords

Mobile System Security; Android Malware; App Confusion Attack; App Virtualization

1. INTRODUCTION

Android is the most popular mobile platform in terms of the number of users [1]. Google Play hosts over 2.5 million Android applications (apps) [2]. Therefore, it is not surprising that this open-source software stack for mobile devices experiences the largest number of malware threats, as evidenced by the report that about 97% of smartphone malware is related to Android [3].

According to [4] [5] [6], Android malware in the wild can be classified into four categories: (1) data theft (e.g., spyware), (2) extortion (e.g., ransomware, and SMS fraud), (3) privilege escalation, and (4) remote control. Except for the last one, which is used for network attacks (e.g., botnet), the first three categories may require malware to exploit other benign apps and their data on devices. For example, ransomware can blackmail victims by locking other apps from being launched, or encrypting valuable data on SD cards [7] [8] [9].

To date, various techniques for hacking Android apps (e.g., vulnerability exploitation, and phishing attack) have been studied and reported [11] [12] [13]. However, these attacks are either unable to access benign apps' internal data, or targeted to a specific app version. More importantly, it is difficult for these techniques to pass through scanning services deployed by app stores. In this paper, we introduce a new attack into the Android malware families, called the *App Confusion Attack*. In this attack, a malicious app hijacks a user's entry to a benign app, and forces the app to run in the virtual execution context under the control of the malware, which is transparent to the user. By sandboxing the benign app, the malware can abuse its code and data arbitrarily.

In order to instantiate the attack, we built *DroidPill*, a framework for malware creation based on *app virtualization*. The app virtualization technique, which we classify into two categories (i.e., inclusive and exclusive) in this paper, encapsulates an app (guest app) in a restricted execution environment within the context of another app (sandbox app). The virtual execution context is carefully crafted such that it is oblivious to the app or app user, while maintaining compatibility with the native context provided by the Android Application Framework. Also, DroidPill uses two different attack vectors, *app shortcut manipulation* and *top activity preemption*, to realize the App Confusion Attack unobservable to users.

There are two distinct characteristics when using DroidPill. First, DroidPill can simultaneously attack multiple benign apps on a device without carrying their code and data. Hackers can utilize this feature to precisely profile users by monitoring their daily app usage. For example, based on the in-app observation of a user's emails, social activities and browsing history, a malware can uncover numerous pieces

AsiaCCS '17, April 4–6, 2017, Abu Dhabi, United Arab Emirates.

DOI: http://dx.doi.org/10.1145/3052973.3052986

of private information (e.g., job, social circle, and personal interests) and build the user's profile. Second, using DroidPill to compromise security-sensitive apps at the application layer could lead to system-wide attacks. For example, a malware can capture the entire network traffic by attacking a VPN app on the device.

To sum up, our contributions include:

- We propose the App Confusion Attack, which hijacks the launching process of each benign app into a virtual environment context controlled by a sandbox app, and takes full control of those benign apps and their resources without device rooting or privilege escalation.
- We implement DroidPill, which uses the app virtualization technique, to launch the App Confusion Attacks with multiple benign apps simultaneously on a device. To our best knowledge, this is the first paper using app virtualization from the hacker's perspective.
- We introduce five use cases with DroidPill, which range from intercepting network traffic to stealing ad revenue, and demonstrates the severity of such attacks for both mobile users and app developers. The demo videos are available at: *https://www.youtube.com/channel/UCRCTTU-bGVpd3o9pmm5THTg*

The rest of this paper is structured as follows. We compare and contrast different attack models, including the App Confusion Attack, and then elaborate the kernel technique for DroidPill, app virtualization, in Section 2. Afterwards, Section 3 details DroidPill's system design and implementation, and Section 4 presents two attack vectors we use. In Section 5, we evaluate DroidPill's performance and itemize several attack cases. While the limitations and the countermeasures are discussed in Section 6, we overview a few related work in Section 7 and conclude the paper in Section 8.

2. CONCEPTS

In this section, we explain two notions that will be frequently used for the rest of this paper: (1) app attacks, which include the App Confusion Attack, and (2) app virtualization, which is used for implementing DroidPill.

2.1 App Attacks

We study how hackers exploit benign apps with an installed malware by comparing and contrasting the existing attack schemes and our App Confusion Attack.

2.1.1 Existing Attacks

Generally, hackers may take one of the following four approaches: (1) ask for dangerous Android permissions (e.g., READ_SMS, and READ_CONTACTS), with which a malware can "legitimately" acquire sensitive data from other apps via the Android framework API calls [4] [5], (2) attack a benign app by exploiting its vulnerabilities (e.g., weak authentication, and SQL injection [13] [14]), (3) apply the UI[1] confusion attacks (e.g., phishing, and tapjacking) to spoof user communications for designated actions (e.g., type in password on a fake login page [15] [16] [17]), and (4) masquerade as a target app by repackaging it with a malicious payload, and lure victim users to install the repackaged app [18]. However, each approach has its limitation.

First, Android does not expose most apps' internal data, so merely requiring the permissions does not guarantee that a malware can obtain the desired data from its target app. For example, a malware can read a user's browser history by requesting the READ_HISTORY_BOOKMARKS permission, but it cannot access the cookies saved in the browser's internal storage. Furthermore, if the user browses websites in incognito mode, then the URLs and cookies are not even stored on disk, thus unavailable to the malware as well.

Second, people treat software vulnerabilities more and more seriously these days. Google's bug bounty program rewards bug hunters for discovering vulnerabilities in Android OS, system apps, and regular Google apps [19]. In addition, several tools (e.g., [20]) have been developed to help people automatically scan app vulnerabilities. For critical vulnerabilities in Android, OEMs can push the patches to devices over the air. Meanwhile, developers can also patch their apps via the app update mechanisms. As a result, many exploits, that are available in old versions of Android OSes or apps, are often blocked in newer versions.

Third, in current UI confusion attacks, malware tricks users into taking a single-step action (e.g., click a button), which restrains the impact of these attacks. For example, such an attack cannot monitor a sequence of users' interactions with a target app, or exploit app functionalities which require multi-step actions (e.g., transfer money from one account to another).

Last, most users tend to install apps from a trusted marketplace, which may deploy the scanning service (e.g., appclone detection [6]) to detect malware. Consequently, hackers may not be able to publish their repackaged apps there. Moreover, the app repackaging approach cannot attack the pre-installed apps (e.g., Chrome) on a device.

2.1.2 App Confusion Attack

In the App Confusion Attack, the malware hijacks a benign app's launching procedure via such techniques as app shortcut manipulation and top activity preemption. Thus, when the user attempts to start the benign app (e.g., click the app icon), the malware is triggered to steal the screen focus by creating a virtual execution context and forcing the benign app to be executed within it. More importantly, the malware should be carefully designed such that the victim user cannot visually distinguish the two situations: (1) the benign app's executions in the virtual execution context, and (2) the native execution context provided by the Android Application Framework. In this way, the malware can perform malicious tasks without awareness at the user level.

The App Confusion Attack is close to the App Masque Attack found in iOS 8.1 and earlier versions [21]. In the app masque attack, the malicious repackaged app can silently replace a benign app installed on a device, even though they are signed by different developers' certificates. The two attacks are similar in the sense that the malware hides its malicious activities by acting on behalf of the guest app, which is transparent to users. However, the app masque attack does not work for Android, because its app update mechanism follows the same origin policy that forbids patching apps by different developers.

[1]User Interface

Table 1: App Attacks

	Code&Data	App Version	App-Clone Detection
Permission Request	Weak	N/A	N/A
Vulnerability Exploitation	N/A	Weak	N/A
UI Confusion	Weak	N/A	N/A
App Repackaging	Strong	Strong	Weak
App Confusion	Strong	Strong	Strong

In comparison with the existing app attack schemes, the App Confusion Attack has three appealing characteristics to hackers. First, this attack scheme gives the malware complete control over a guest app's code and data inside the virtual context, which may lead to the system-wide security breaches. This overcomes the shortcomings of permission request attacks and UI confusion attacks. Second, the App Confusion Attack occurs at the application layer, and does not require exploiting vulnerabilities in the kernel or system apps; therefore, it is possible to attack a wide range of the apps running on various Android versions. Actually, the App Confusion Attack can be used as plan B for hackers when their efforts of escalating privileges are failed. Last, in an App Confusion Attack, the malware does not need to carry a guest app's code and data; therefore, it has little footprints to app-clone scanners. Compared with the repackaged apps, it has a better chance to survive the app-clone detection run by app stores. Table 1 compares the App Confusion Attack and the other schemes.

2.2 App Virtualization

2.2.1 Classification

In Android, app virtualization is a technique that allows a sandbox app to create a virtual execution context, in which a guest app can be loaded and executed in the same way as it runs in the native execution context provided by Android OS. Sandbox apps are actually regular Android apps that do not have any system privilege. Functionally, app virtualization consists of *OS service virtualization* and *storage virtualization*. While OS service virtualization converts virtual OS services used by guest apps to physical OS services used by sandbox apps, storage virtualization translates virtual storage used by guest apps to physical storage used by sandbox apps.

In OS service virtualization, the sandbox app plays the role of a broker: to the guest app, it acts on behalf of the Android system services and provide the same service interfaces. Whereas, to Android system services, it hides the guest app's identity and issues its service requests. In such a situation, an identity gap exists between the guest app and the Android system services: While the guest app claims that "I am Alice", Android system services say that "You are Bob". In storage virtualization, the similar identity gap exists between the guest app and the OS storage system. In sandbox apps, virtualization logic should be designed and implemented at discretion to fill such identity gaps; otherwise, it will result in run-time exceptions or unpredicted program behaviors.

Table 2: App Virtualizations

	Inclusive	Exclusive
Guest App	Yes	No
System Services Emulation	Yes	No
App Object Translation	Yes	Yes

App virtualization can be classified into two catogories: *inclusive app virtualization* and *exclusive app virtualization*. In an inclusive app virtualization system, the sandbox app is designed to virtualize any non-system app, and it can create a workspace to manage and execute multiple guest apps at the same time. Whereas, in an exclusive app virtualization system, the sandbox app is exclusively constructed for virtualizing a predefined group of guest apps. It cannot work with any app outside the group.

For inclusive app virtualization, the sandbox app needs to emulate several core system services (e.g., PackageManager, and ActivityManager), which facilitate the tasks of managing multiple guest apps (e.g., IPC communications between guest apps or their components). *App Object Translation* (AOT) is used to achieve the virtualization. Specifically, when the sandbox app is installed, it registers a list of dummy app objects (e.g., components) via the AndroidManifest file. Afterwards, when a guest app is started within the sandbox app, the latter creates an app object map that associates app objects of the guest app with the registered dummy ones, which have not been mapped yet. When the guest app is actually executed, the sandbox app uses the app object map to perform object translation between different components. Once the guest app is killed, the corresponding dummy app objects are freed and available for the use by other guest apps. On the contrary, an exclusive virtualization system does not need to emulate any core system service, and the sandbox app and its guest apps are installed simultaneously. Although it also uses AOT to implement virtualization logic, the app object maps are generated at creation of the sandbox apps, and remain unchanged at runtime. Table 2 compares the two app virtualizations.

2.2.2 Existing Solutions

Boxify [22] and NJAS [23] are two existing app virtualization systems that sandbox unmodified and untrusted apps in stock Android. In their implementations, a sandbox app contains at least two functional components: *sandbox service* and *broker*. While the former is mainly responsible for virtual context setup and initialization (e.g., install the hooking code), the latter performs virtualization logic and enforces security policies.

Boxify is an inclusive virtualization system that leverages the "isolated process" feature to create a tamper-proof app sandboxing system. In Android, an app running in an isolated process cannot conduct any operation which requires Android permissions (e.g., access data on SD card). In Boxify, the sandbox app's broker runs in a normal process, and

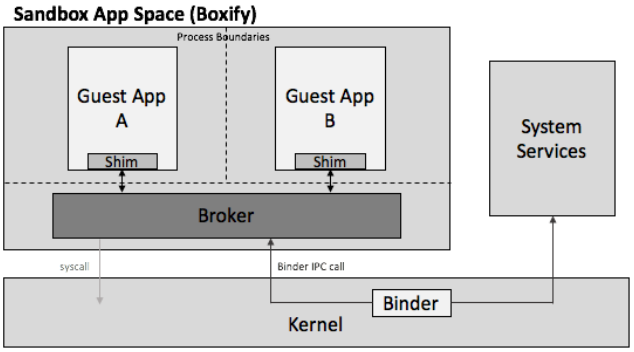

Figure 1: System Architecture (Boxify)

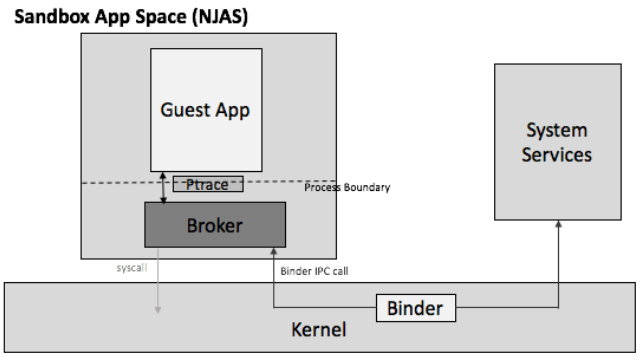

Figure 2: System Architecture (NJAS)

the guest apps run in different isolated processes along with the sandbox service. At start time, the sandbox service installs the GOT[2] hooks, called *Shim*, that can redirect the Binder IPC and the guest app's system-level calls to the broker. After the guest app is loaded, the broker monitors references and performs virtualization logic. Figure 1 displays the system architecture of Boxify.

NJAS is an exclusive virtualization system that builds its sandbox system on top of the *ptrace* mechanism. Unlike Boxify, NJAS sets up the broker and a guest app in separated processes that have the same app privileges. It relies on the ptrace system call to ensure that the broker can intercept and inspect system calls made by the guest app, including adverse ptrace calls' prevention. NJAS suffers two limitations, that lead to an unfledged app virtualization system: First, its sandbox app can only run one guest app; Second, it does not have sufficient AOT and misses translating a number of app object types. Figure 2 depicts the system architecture of NJAS.

3. DESIGN AND IMPLEMENTATION

3.1 Design Rationale

DroidPill, as an offense system, has distinct security requirements from Boxify and NJAS, which are designed for defensive purposes. First, Boxify and NJAS must strictly adhere to the "reference monitor" properties (i.e., complete mediation and tamper proof [24]), which are not mandatory for DroidPill. Second, DroidPill should meet the *UI Integrity* requirement, under which a user cannot visually tell whether a guest app runs in the native or virtual execution context.

Without worrying about the security requirements of reference monitors, we decide to move the sandbox app's broker into the same process space as the guest app. By hooking DVM and patching native libraries' GOTs, the broker can mediate both application-level and system-level API calls made by the guest app. In addition, the broker can access the abundant Java API semantics to overcome semantic gap between the system-level view and the application-level view for app data [25] and enable more flexible attacks.

Ideally, DroidPill should use inclusive app virtualization, which permits a sandbox app to virtualize non-system apps. However, inclusive app virtualization breaks the UI Integrity requirement for the majority of Android devices. Android overview screen is the system-level UI that displays running activities and tasks on a device. Users often use it to switch or kill running apps by clicking or swiping out the app icons. For Android 4.4 and older versions, which still account for about 47% of the total Android devices using Google play store as of October 2016 [26], an activity's icon and label defined in AndroidManifest are registered at app installation, and they cannot be changed at runtime. For those devices, a dummy activity is always displayed with a fixed icon and label on the overview screen, which is not visually aligned with the mapped activity of a guest app. This is not acceptable to DroidPill, because a user can easily detect the visual anomaly of a virtual guest app on the overview screen. Another design concern about inclusive app virtualization is that a sandbox app needs to acquire a large number of Android permissions in order to work with a variety of guest apps, which instantly exposes the DroidPill malware to scrutiny by security scanners on devices or at app stores [27].

Therefore, DroidPill adopts exclusive app virtualization, since sandbox apps are built based on the APK files of predefined guest apps, and original icons of the guest apps can be added to the sandbox apps and statically registered to Android at app installation. Unlike NJAS, DroidPill is able to virtualize multiple guest apps simultaneously. As a result, one DroidPill malware can reliably mount attacks against multiple benign apps on a device.

3.2 System Architecture

In DroidPill, a sandbox app consists of three parts: *bait*, *constructor* and *broker*. Bait is mainly used for luring users to install and execute sandbox apps (malware), and it should provide the functionalities as advertised to users. In addition, bait contains attack vectors that hijack guest apps' launch sequences. Similar to Boxify's sandbox service, constructor is responsible for installing broker and loading guest apps. Broker is in charge of virtualization and attacks against guest apps by mediating communications between a guest app and the OS (i.e., kernel and system services). The constructor and the broker work together to build a virtual execution context for loading and running a guest app. Figure 3 shows the system architecture of DroidPill, in which we exclude the bait and the constructor to compare with Boxify and NJAS.

DroidPill enforces the process segmentation of guest apps. Each guest app runs inside its own process space along with the constructor and the broker, and the bait has its own standalone process. The design is motivated by the Chrome

[2]Global Offset Table

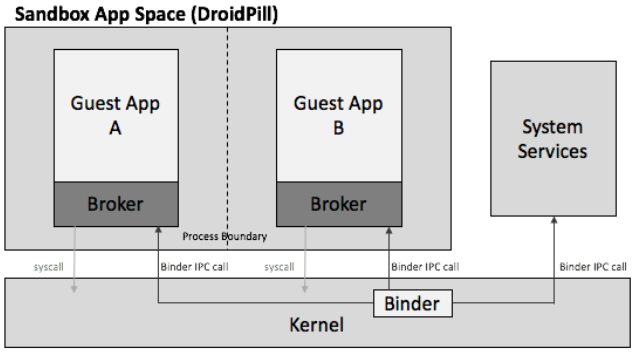

Figure 3: System Architecture (DroidPill)

browser, in which tabs are executed in separated processes, and the exception of one tab does not crash other tabs. Likewise, the process segmentation assures that a guest app is protected from crashes incurred by errors from another guest app. We solve this problem by taking advantage of the Android feature that allows an app to customize the process name for each component. Specifically, a custom process name for a component can be defined in the "process" attribute of the component tag in AndroidManifest. When the first app component of a process is to be executed, the activity manager spawns a new process from the zygote process and names it using the process name defined in AndroidManifest. In DroidPill, the AndroidManifest file of a sandbox app contains not only a set of the bait's components but also the sets of dummy components corresponding to the guest apps. By manipulating the "process" attributes of the component tags, we can assign the bait and the guest apps to different processes at runtime.

Further, when a new process of the sandbox app is created, it can use process names to determine whether to execute the bait or a correspondings guest app, and load the app code to the process' memory space. In implementation, we build a custom *android.app.Application* class and define it in a sandbox app's manifest file. Android ensures this custom Application class is initialized ahead of any other app code at runtime. Each time a sandbox app's new process is started, the initialization function of this custom Application class selects an app and loads its code based on the current process name. Unlike NJAS that only supports virtualization for a single guest app in a sandbox app, DroidPill has two additional features (i.e., *process segmentation*, and *selective app loading process*).

3.3 Constructor

The constructor's task is to install the broker and load guest apps, which is mainly implemented in the initialization function of the custom Application class. The constructor can load the native library where the broker is encapsulated via calling *System.loadLibrary()*. Here we will explain how to load a guest app into the virtual execution context.

3.3.1 Guest App Loading

According to [28], Android allows an app to load the code and data of another app in its own process space via the *Context.createPackageContext()* framework API, even though they come from different developers. Note this API only works for free apps, because the apk files of paid apps are not readable from a non-paid one.

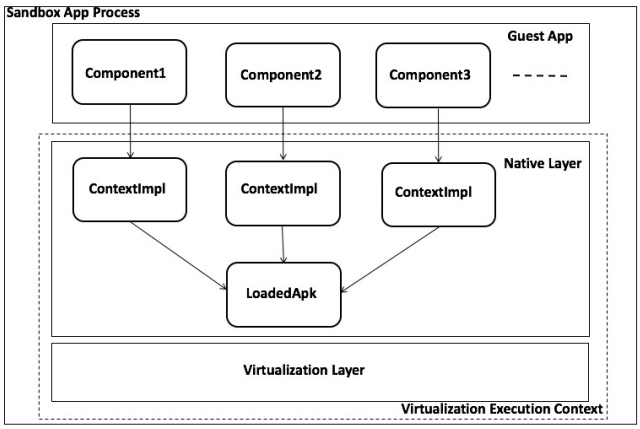

Figure 4: Virtual Execution Context for Guest Apps

In DroidPill, the constructor invokes this API at runtime with the *CONTEXT_INCLUDE_CODE* and *CONTEXT_IGNORE_SECURITY* flags to load a guest app, without containing any data and resources from guest apps in the sandbox app.

3.3.2 Virtual Execution Context

Android provides a variety of framework classes to weave the execution contexts for apps. Among them, two classes are critical (i.e., *android.app.LoadedApk* and *android.app.ContextImpl*). When a new app process is generated and its apk file is load, a LoadedApk object is created to store the metadata for the apk, which includes package name, code loader, resource, and app data paths. During this process, both app code and data (e.g., references) are loaded to the memory, and they are referenced by the code loader object (i.e., *java.lang.ClassLoader*) and resource objects (e.g., *android.content.res.Resources* and *android.content.res.AssetManager*). Therefore, the app data and resources of the apk file can be accessed via the hosting LoadedApk object. Moreover, ContextImpl provides the major interface for an app to interact with the rest of world. Through this interface, the app can connect to the Android system services, launch the activities and services, and visit the package data. To build the execution context for a new app component, Android creates a new ContextImpl object, and attaches the object to the component. As the ContextImpl object includes a LoadedApk object, the component can retrieve the app resources via the resource objects inside the LoadedApk object. Here, each component has its own ContextImpl object, but the app has only one LoadedApk object.

In DroidPill, the virtual execution context of a guest app and its components is composed of two layers (i.e., the native layer and the virtualization layer). The guest app is directly run over the native layer, and the virtualization layer dwells underneath. The native layer is the native execution context of the guest app, and it contains the ContextImpl and the LoadedApk objects of the guest app, which are created when the constructor calls *Context.createPackageContext()* to load the guest app. The virtualization layer intervenes the data transferred between the native layer and the OS, and carries out the virtualization logic. The virtualization layer is generated when the constructor loads the broker's

native library to the memory. Figure 4 depicts a guest app's virtualization execution context.

3.4 Broker

The broker intercepts the guest apps' application-level and system-level API calls by instrumenting DVM and native system libraries, which empowers DroidPill to perform virtualization logic.

3.4.1 Public App Objects

In Android, apps should claim and use the system-wide unique IDs for some app objects at installation or runtime (e.g., components whose ID is the combination of the app name and component name). This assists the system services (e.g., ActivityManager) in mediating the inter-app and inter-component communication, and also centrally managing the global resources (e.g., online account credentials). If an app violates this rule and attempts to use the duplicate object IDs, it can result in the installation's rejection or broken functionality. For example, an authority is used for uniquely identifying the data store associated with a content provider, and its value should be globally distinctive. The Android installer automatically rejects any app that declares a duplicate authority in the manifest file.

While we refer to an app object that must have a system-wide unique ID as a *public app object*, five types of public app objects (i.e., *Component*, *Authority*, *Account Type*, *Custom Permission*, and *Intent Action*) are identified. These public app objects are defined in AndroidManifest or an xml resource file and registered to the package manager or other system services for handling the runtime operations related to the apps. Table 3 displays a list of public app object types that includes their names, definitions in AndroidManifest and duplication penalties.

In addition to Component and Authority, we give a brief explanation of the other public app objects. (1) The Android account manager uses Account Type provided by an app to uniquely represent an online account service, and to help the app perform authentication with backend servers. The account manager always rejects a duplicate Account Type, leading to an app's malfunction. (2) Custom Permissions are app-defined permissions that guide system services to enforce the fine-grained access control over app data. If two apps from different developers define an identical signature-level permission, then the late-installed app may not have the permission granted and may lose the access to its app data. (3) Intent Actions are defined in the intent filters by either system or user apps. When receiving an implicit intent without a component name, activity manager uses its intent action to decide which app to send. If multiple apps register this intent action for their activities, activity manager displays a picker dialogue that shows all apps that accept this intent action, and the user needs to manually choose the one to process the intent. With DroidPill, a user may see the duplicate component icons and labels, one from the guest app and the other from the sandbox app, in the picker dialogue, which breaks the UI Integrity requirement in an event of duplicate actions. Intent Actions in other components do not have this issue.

3.4.2 Virtualization Logic

Droidpill fulfills OS service virtualization in two steps. (1) A sandbox app creates the virtual names for public app objects claimed by a guest app in AndroidManifest, and registers them to the package manager at app installation. An app object map is generated in this step that records the mappings between the real object names defined by the guest app and their virtual names defined by the sandbox app. (2) The sandbox app's broker uses the app object map to conduct the AOT at runtime. Specifically, for an outgoing public app object, the broker renames it from the real name to the corresponding virtual name, while for an incoming public app object, the broker translates it from the virtual name to its real name.

Table 3: Public App Objects

Type	Declaration	Penalty
Component	<manifest> - "package" attribute & component - "android:name"	Installation Reflection
Authority	<provider> - "android:authorities"	Installation Reflection
Account Type	Any in Authenticator and SyncAdapter's config.xml files - "android:accountType"	Broken Functionality
Custom Permission	<permission> and <uses-permission> - "android:name" & Any - "android:permission"	Broken Functionality
Intent Action	<action> - "action:name"	UI Integrity Violation

We instrument the Binder IPC APIs that interact with the system services (e.g., activity manager, and account manager). In this way, the broker is able to inspect and modify the public app objects located in parameters or return values of the hooked APIs exchanged between the guest app and the system services. Comparing with NJAS, which performs AOT over only one of the public app object types, DroidPill's AOT covers five types of public app object, which makes it a more mature system. For example, while the translation of Intent Action allows DroidPill to support invoking an app component via implicit intents within the app, NJAS does not. In our implementation, we hooked over 100 Android framework APIs to gain a sufficient service virtualization.

To achieve storage virtualization, the broker hooks the low-level I/O functions in native libraries to redirect the file accesses from a guest app's storage to the sandbox app's storage. Without this, the access to the guest app's internal data storage will be blocked by Android in that the guest app and the sandbox app have different UIDs. Another task of the broker is to hide the sandbox app in the query results returned from the package manager to the guest app, which can be done by hooking the Binder API calls to the package manager.

3.4.3 An Illustrative Example

Here, we use an example to demonstrate the complexity of the AOT operations. In Android, account manager is designed to facilitate apps' authentication with backend servers, which benefits to app developers. For example, one copy of authentication code can be shared by multi-

```
public class VictimWelcomeActivity extends Activity {
    public final static String ACCOUNT_TYPE = "com.victim.android.account";
    ...

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        String accountName = this.getSharedPreferences("accountdetails",
                MODE_PRIVATE).getString("account_name", null);
        String authTokenType = this.getSharedPreferences("accountdetails",
                MODE_PRIVATE).getString("auth_token_type", null);
        ...

       // we don't have valid auth token, then query the account manager.
        AccountManager accountManager = AccountManager.get(this);
        Account mAccount = new Account(accountName, ACCOUNT_TYPE);
        AccountManagerFuture<Bundle> future = accountManager.getAuthToken(mAccount,
                authTokenType, true, new OnTokenAcquired(), null);
        ...
    }

    ...
}
public class VictimAccountAuthenticator extends AbstractAccountAuthenticator {
    ...

    @Override
    public Bundle getAuthToken(AccountAuthenticatorResponse response, Account account,
        String authTokenType, Bundle options) throws NetworkErrorException {
        ...

        // we can't access auth token, so we need create an intent to display
        // VictimeLogonActivity, and ask user for inputting his credential
        final Intent intent = new Intent(this, VictimLogonActivity.class);
        final Bundle bundle = new Bundle();
        bundle.putParcelable(AccountManager.KEY_INTENT, intent);
        return bundle;
    }

    ...
}
```

Figure 5: Code Snippets

ple apps connecting to the same backend server. Therefore, apps usually create customized authenticator services to handle related tasks, including storing the credentials (e.g., usernames, and passwords), authenticating with backend servers, and caching authentication tokens. An app registers the authenticator service in AndroidManifest via the keyword of *android.accounts.AccountAuthenticator* within the "action" and "meta-data" tags, for which Android can locate in an xml resource file that defines account types and other configuration data. Moreover, account manager maintains a list of registered authenticator services indexed by their account types at runtime, and the app can interact with the account manager to indirectly access these authenticator services via Binder.

Consider a victim app, whose package is named *com.victim.android*, attempts to acquire a valid authentication token to pull out user data from a backend server and displays them on the home page when the app starts. If there is no valid authentication token due to the first use or the token expiration, a logon page is displayed to ask for user credentials. In particular, the victim app has a home page activity called *VictimWelcomeActivity*, and its *onCreate()* function contains the code using a hard-coded account type, *com.victim.android.account*, to query the account manager with *getAuthentication()* for a valid authentication token. In addition, the victim app implements its authenticator service's core functionalities in the *VictimAccountAuthenticator* class. As a result, the account manager can redirect the *getAuthentication()* call made by *VictimWelcomeActivity* to *VictimAccountAuthenticator*'s *getAuthentication()*, which generates the intent for launching *VictimLogonActivity* in that no valid authentication token exists. The relevant code snippets of *VictimWelcomeActivity* and *VictimAccountAuthenticator* are shown in Figure 5.

In order to virtualize the victim app, DroidPill's sandbox app registers a virtual authenticator service with a virtual account type named *com.droidpill.android.account*, and a virtual logon activity named *DroidpillLogonActivity*, via its manifest file. When the victim app runs on top of the sandbox app and invokes *getAuthToken()* inside *VictimWelcomeActivity*'s *onCreate()* function, the broker of sandbox app first intercepts the corresponding Binder call, *IAccountManager$Stub$Proxy.getAuthToken()* and its account parameters containing account type. After then, DroidPill conducts AOT and changes account type from the real one (i.e., *com.victim.android.account*) to the virtual one (i.e., *com.droidpill.android.account*). While the account manager uses the virtual account type to retrieve the virtual authenticator service in cache and invoke its *getAuthToken()* via Binder, the broker captures the corresponding Binder call (i.e., *AbstractAccountAuthenticator$Transport.getAuthToken()*) from the app side, and then reverses the account type. After *VictimAccountAuthenticator.getAuthToken()* is executed by the victim app, it generates a bundle object that encapsulates the intent of launching *VictimLogonActivity*. Then, the bundle object is passed to the account manager via another Binder call (i.e., *IAccountAuthenticatorResponse$Stub$Proxy.onResult()*). The broker intercepts this call and performs AOT over both account type and activity name from the real ones to virtual ones. Later, the account manager uses a Binder call to transfer the modified bundle object to the framework's *AccountManager$AmsTask* object inside the sandbox app process to launches the logon activity. The broker captures the corresponding Binder call, *AccountManager$AmsTask$Response.onResult()*, from the app side, and then reverses the account type and the logon activity name. To sum up, four AOT operations are conducted to get authentication tokens from *VictimWelcomeActivity* during the entire app vitualization process, as depicted in Figure 6.

4. ATTACK VECTORS

In this section, we focus on how to use DroidPill to launch the attack. But, before getting into the details, let's assume that a DroidPill malware has been successfully installed on a device, and the device user treats the malware as a regular app with the bait's functionalities. In order to attack a benign app, DroidPill needs to hijack the user's entrance to a benign app, and force the app to run in the virtual execution context built by the malware. Two methods (i.e., *app shortcut manipulation* and *top activity preemption*) are employed to launch the attack.

4.1 App Shortcut Manipulation

Since Android API level 19 (Kitkat), two new permissions INSTALL_SHORTCUT and UNINSTALL_SHORTCUT are added to AOSP. After acquiring these permissions, an app is able to create and remove shortcuts from the home screen for any free app on the same device, even those with different UIDs. With these two permissions, the DroidPill malware can stealthily substitute the shortcut of a benign app with its own shortcut that has the same icon and label, which is not noticed by the user. As a matter of fact, it has been reported that other Android malware may take advantage of this weakness to mount phishing attacks [5]. However, this method may become ineffective if the user starts the app from another entry point (e.g., app manager, and notification center).

4.2 Top Activity Preemption

According to [11], a technique that allows an app to promptly cover a target activity with another one, when the target activity is brought to the foreground

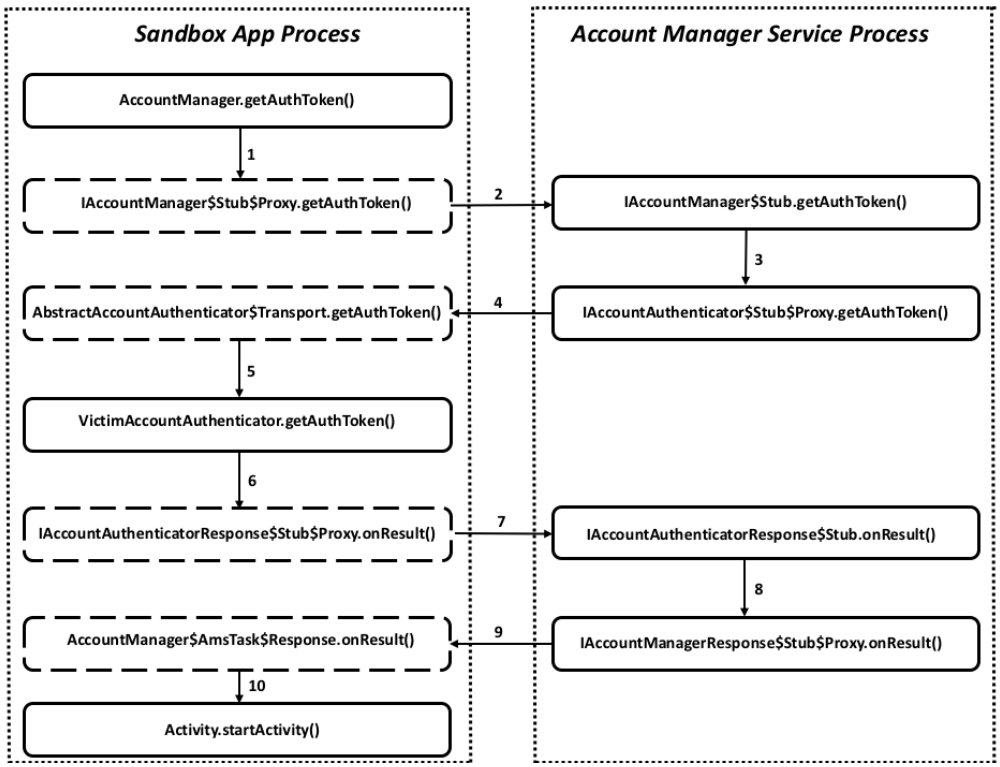

Figure 6: Function Call Flows of the Example (Dotted functions are hooked to perform the AOT operations)

on the screen, is widely used by app lockers. There are three ways for the malware to learn what the on-top activity is: (1) read the system log, (2) run the *android.app.AcitivityManager.getRunningTasks()* API, and (3) perform side-channel attacks via accessing the *proc* file system [10] [12]. While the first two are blocked in Android 4.1 and 5.0, respectively, as far as we know, the last one is still effective for all Android versions.

In order to attack the on-top activity of a benign app launched by a user, the DroidPill malware can employ side-channel attacks to start the same activity in its sandbox to occupy the top activity position, and block the user's view to the original activity running in the native context. Note that the attack should take place quickly such that the user would not see the UI anomaly (e.g., one activity UI showing up twice). However, after this attack, two duplicate activity icons could be displayed in the overview screen, and it breaks UI Integrity. To address this, we leverage the task hijacking attack to remove the duplicated icon [9]. For example, in the DroidPill prototype, we manipulate the "taskAffinity", "allowTaskReparenting" and "exclueFromeRecents" attributes of the guest app's activities in the sandbox app's AndroidManifest in the way such that Android places the native activity and the sandboxed one in the same activity task, and the former is on top of the later in the activity stack. Moreover, *android.app.Activity.onBackPressed()* is hooked to ensure that the native activity's UI is skipped when the "Back" button is clicked. Note that a malware built based on the inclusive app virtualization cannot overcome the duplicate activity icons in the overview screen, because a sandbox app cannot programmatically change the "taskAffinity" attribute of its dummy activities.

Table 4: Quadrant Test Result

	CPU	MEM	I/O	2D	3D
DroidPill	4.76%	1.54%	12.54%	1.25%	6.01%
NJAS	6.87%	0.47%	112.32%	0.00%	9.90%

5. EVALUATION

After implementing DroidPill with 3365 lines of C++ code and 2536 lines of Java code in Android 4.4, we compare its performance with that of NJAS, and then use the prototype to launch five attacks in different scenarios.

5.1 Performance

In order to look into the impact of app virtualization to the performance of apps with DroidPill, we conducted an experiment on a Samsung Note 2 running Android 4.4.4 with two versions of Quadrant (i.e., v1.1.1, and v2.1.1) sandboxed with a DroidPill malware. Quadrant is a benchmark app tailored for testing the performance of Android devices in various aspects, including CPU time, memory speed, I/O operation speed and graphic rendering time. We ran the test app ten times and took the average with a small variance. Becuase NJAS and Boxify reported their performance evaluation results over Quandrant v1.1.1 and v2.1.1 respectively, this allowed us to compare the DroidPill's performance with theirs directly.

Table 4 compares the results of DroidPill from the tests we ran and the publicly available results of NJAS, and demonstrates that the I/O penalty in DroidPill is much lower than that in NJAS. For the reason that, the I/O operations in NJAS trigger the *ptrace* system calls, and generate the context switches between the parent and child processes. Whereas, DroidPill's performance is not substantially pe-

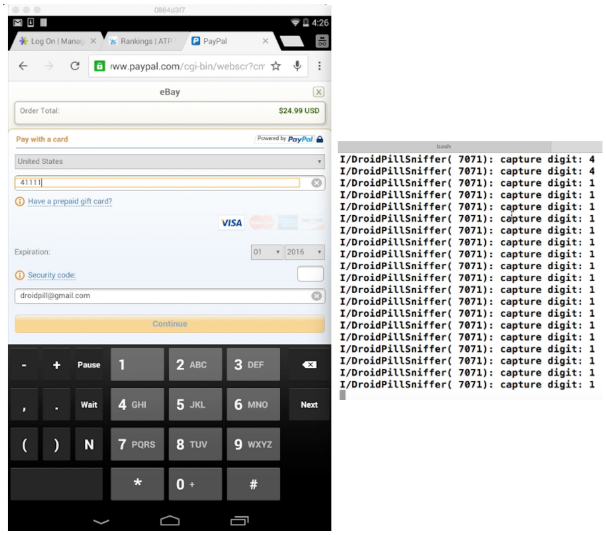

Figure 7: Steal Credit Card Information on Chrome

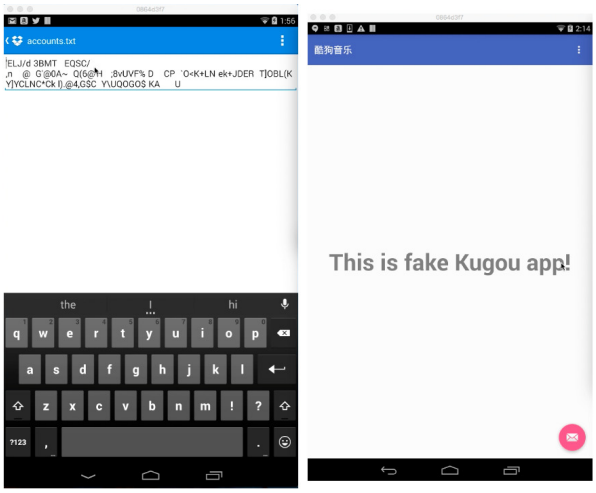

Figure 8: Encrypt a User's File on Dropbox (left) and Hijack an App Installation from Mi Store (right)

nalized in that guest apps and its broker run in the same process, and no additional switch occurs for each I/O operation. For Quadrant v2.1.1, DroidPill's performance penalty is 4.5% in contrast to Boxify's 3.6%. Overall, it can be observed that the results are comparable to NJAS and Boxify, since most of the framework APIs that DroidPill hook are related to Binder calls and should not cause significant overheads for CPU, memory, I/O and display. Throughout the experiments with real-world apps, our DroidPill malware did not encounter any noticeable performance hit. Nonetheless, there is room to improve our current DroidPill implementation. For example, the current AOT operations are written in Java, which incurs execution overhead when switching between native code and Java bytecode within hooking functions at runtime. Such an issue can be eliminated by implementing AOT in C/C++. All in all, we believe that DroidPill's performance falls into an acceptable range for a malware system.

5.2 Case Studies

In order to demonstrate DroidPill's effectiveness, we present five attack examples against real-world apps. Specifically, we implemented the DroidPill malware and tested them on a non-rooted Google Nexus 7. The malware displays a simple welcome activity as the bait, and utilizes the *getRunningTasks()* API to detect and preempt the guest apps' top activities. During each test of the UI Integrity requirement, the overview screen is opened up to show that the icons and labels of sandboxed activities are identical to that of native ones. In the experiments, we didn't publish the malware to any app market due to the legal concern, so we just installed the malware locally. In the future, we would like to work with app market vendors to evaluate the market-level defense against the DroidPill attacks.

5.2.1 *Browser Spyware*

In this attack, a DroidPill malware was created to attack the Chrome browser app (version 48.0.2564.95) and steal all a victim user's searching and browsing history in both normal and private modes. Moreover, it could steal the login credentials and financial information typed in by the user, as shown in Figure 7. In this experiment, we hooked a number of methods from the Android framework and chrome classes (e.g., *android.view.inputmethod.BaseInputConnection* and *org.chromium.chrome.browser.tab.Tap*) that handle the keyboard inputs and load URLs. Note these are the application-level APIs that the malware intercepted. We believe that further instrumentations at this layer could reveal more browsing contents (e.g., encrypted web traffic). Merely hooking the low-level APIs will not give us such visibility to the encryption-protected browsing data.

5.2.2 *Cloud Storage Ransonware*

We used a DroidPill malware to attack the Dropbox app (version 2.4.5.10) with the system-level I/O functions (e.g., *open()*, and *write()*) which monitor what files are downloaded and stored to the local cache. The malware is instructed to silently download all files to the local cache, and encrypt them all. By taking advantage of Dropbox's autosync mechanism, file encryption is automatically propagated to cloud servers and other client devices. Thus, all files in the Dropbox account will be unreadable to the victim user. If the user pays off a ransom, the malware could decrypt the files. Although Dropbox can automatically back up the user's files and allow the user to revert the files to previous versions, a DroidPill ransomware could bypass the recovery mechanism by encrypting the files multiple times to overwrite all plaintext versions. Figure 8 (left) depicts that a Dropbox malware encrypts a text file.

5.2.3 *App Store Client Abuser*

We built a DroidPill malware to hack the Xiaomi Market app (version R.1.4.2), which is the default app store client for Xiaomi phones. In this experiment, the malware abuses the Xiaomi Market app's installing process, and uses a fake app to replace a genuine app downloaded from the Xiaomai app market. In this experiment, we use a Google Nexus 7 to install the market app as a regular app without system permissions through the Android package installer. We hooked the *ActivityManagerProxy.startActivity()* API to intercept the intents that the market app sent to the installer. These intents were characterized with the MIME type of "application/vnd.android.package-archive". Our malware modifies the URIs in the intents that pointed to the apk

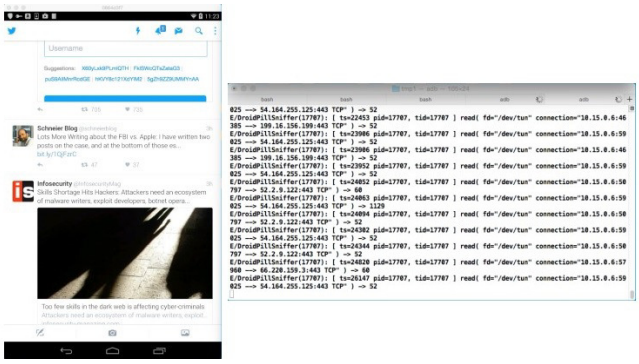

Figure 9: Intercept VPN traffic on Betternet

file of a fake app. Therefore, when the market app attempts to install an app from its online store, only the fake one could be installed, as depicted in Figure 8 (right). Note that Google Play client is not vulnerable to App Confusion Attack, because DroidPill is not assumed to have system permissions.

5.2.4 VPN Traffic Sniffer

Betternet is a free VPN proxy app with over 10 million downloads from Google Play. In order to use the VPN functionality, Android needs users to explicitly grant the permission to the app via a confirmation dialog showing the VPN app's icon and label. It's challenging for DroidPill, since when the malware's sandbox app icon and label are displayed in the dialog instead of the Betternet's, which breaks the UI Integrity requirement in the App Confusion Attack. However, according to [16], those dialogs suffer from tapjacking attacks, we combined the App Confusion Attack with the tapjacking attack to exploit this VPN app.

In our experiment, the Betternet app (version 2.6.2) spawns a child process to run a customized Linux-style executable to perform the VPN task, and the sandbox app's code (i.e., the constructor and the broker) are not loaded and executed in that child process, so DroidPill's existing sandboxing mechanism does not intercept VPN traffic in that child process. Finally, to address this issue, we use *ptrace* for the DroidPill malware to inject a hooking library to the child process, which hooks the system-level I/O function (i.e., *read()*) to capture and view all VPN traffic on the test device. See Figure 9.

5.2.5 Ad Revenue Stealer

We demonstrate that DroidPill is able to exploit the Android ads system and steal the ads revenue from benign apps. However, in order to stay away from legal liability, we do not disclose app names and ad networks that we tested. Let's call them app X and ad network Y. The ad network Y assigns ad unit IDs to app developers when including Y's SDK in their apps. When displaying ads, an app supplies its ad unit ID to Y's ad server, which helps Y map ads to the developer account. In the experiment, we signed up a developer account and acquired an ad unit ID from Y. A DroidPill malware was created for app X. By interposing an API in Y's SDK, the malware was able to replace the X's ad unit ID with the one that Y assigned to us. After displaying/clicking ads in the hacked X for a few days, we found that our developer account received a small amount of money from Y.

6. DISCUSSION

In this section, we mainly answer two questions: (1) What are the limitations of our DroidPill prototype? (2) Are there any possible countermeasures to the App Confusion Attacks?

6.1 Limitations

Although we have successfully launched the attacks with DroidPill, we envision six restrictions, either external or internal, that thwart the current prototype to expand to all app types.

The external restrictions are more or less related to our implementation, which include:

Android Versions The current implementation of DroidPill uses DVM to hook the application-level APIs, and thus only works with Android 4.4 and older versions. Fortunately, the researchers have proposed the ART hooking techniques recently [29] [30], which allows us to transport DroidPill to Lollipop and newer version. Therefore, DroidPill may be extended to support all Android versions. Also, since Android 5.0, apps can better control which entries they generate on the overview screen. For example, Chrome uses this feature to display every tab as a separate app on the overview screen by default. Therefore, an inclusive app virtualization system can take advantage of this feature to establish the UI Integrity requirement, although it cannot utilize top activity preemption to launch the attack.

Monetization Strategies Since the apk files of paid apps and system apps are not accessible for a free app, we can only use DroidPill to launch the attacks for free apps. However a DroidPill malware can download a target paid app's apk file from a hacker-controlled server to instantiate an attack. In such a situation, we cannot reply on *Context.createPackageContext()* to load the guest app to the virtual execution context. Instead, DexClassLoader can be used to directly load the guest app's apk file.

The internal restrictions are unavoidable, which include:

Android Permissions In order to successfully launch the attack, a DroidPill malware needs to request the same permissions as the guest app. However, the malware does not carry the same code and data, and thus it may violate the least privilege principle. If the malware excessively request the "dangerous" permissions, it may be susceptible to the detection of anti-virus scanners.

System UIs DroidPill malware cannot completely prevent a guest app from running and interacting with system services in its native execution context. Simultaneous executions of a guest app in native and virtual environment contexts may generate duplicate app icons and labels in the system UIs (e.g., notifications in notification center, and shortcuts on home screen). Unlike the overview screen for Android 4.4 and older versions, these system UIs are designed to permit showing duplicate items. For example, notification center allows two same notifications from the same app, or launcher apps allow users/apps to manually/programmatically create multiple shortcuts on the home screen. For our design choice, the DroidPill malware can optionally turn off notifications and shortcuts posted to the system UIs by guest apps.

App Activities In Android, app activities could be launched by other system apps with parameters having a launching intent that affect their UIs. For example, a news app may push a news notification with a URL, when a user

reacts to the notification, the app viewer can use the URL to download and display the online news on the device. In such situations, if the news notification is generated by the original app running in the native environment context, the DroidPill malware cannot intercept the "viewer" activity which launchess the intent and the URL, so the victim user cannot see the news in the UI of the sandboxed "viewer" activity.

App State In Android, apps' local storage can be used to save users' state data (e.g., login credentials), and restore the previous app state for next time. However, as a benign app's state data are usually stored in its internal local storage, the DroidPill alternative cannot access them. Therefore, when a sandboxed guest app is started for the first time, it has to run from the initial state. However, users may only need to save its state data once. In fact, the same situation happens on benign apps as well due to reasons such as software update and implementation errors. Therefore, we expect that users may not be aware of the inconvenience is actually caused by our DroidPill malware.

6.2 Countermeasures

In order to mitigate the App Confusion Attacks, we envision the following countermeasures from either of two places: (1) the OS level, and (2) the marketplace level.

At the OS level, in order to defeat the App Confusion Attacks, we can take the following defense strategy: assure that the identity that an app protrays to its user is the same identity that it is seen by the Android OS. Antonio [11] borrows this idea from the anti-phishing solutions in web security, and adds a security indicator to the system navigation bar. This security indicator reveals the real identity of the front-end app to the user, including app name, company name and secure image. Unfortunately, it needs modify the Android OS and framework. An effective and lightweight solution without OS support is still an open problem.

At the marketplace level, the Google Play licensing service provides a practical solution to address the App Confusion Attacks [31]. A developer can split her app in two parts: one boot loader app and one heavy library. The boot loader app is installed on devices, and the library is uploaded to Google's license server. Major functionalities of this app are implemented in the library. To run the app properly, the boot loader app needs to successfully authenticate itself to the license server, and then download and execute the library. During the license authentication process, the boot loader app submits its app name to Google Play client via the Binder API call. To verify the caller's identity, Google Play client invokes *Binder.getCallingUid()* to get the caller's UID and app name from the PackageManager service, then compares it with the app name submitted by the caller. In this way, DroidPill's sandbox app cannot hide its identity and spoof the Google Play client. The license authentication fails. Unfortunately, most free apps do not use this service currently. In addition, the approach cannot protect the apps in other app stores that do not offer the licensing service.

7. RELATED WORK

Invisible Rootkits In the domain of desktop and server, researchers used Virtual Machine Monitor (VMM) to create invisible rootkits [32] [33] [34]. The VMM rootkits are powerful because of the possession of the strong reference monotoring properties (i.e., complete mediation, and tamper proof) as well as the UI Integrity requirement. Unfortunately, the technique suffers from two weaknesses: (1) It is highly hardware-dependent and thus difficult to migrate among different CPU architectures, and (2) It is challenging to find attack vectors for installing the VMM rootkits into target machines. In contrast, DroidPill is an application-level software and thus easier for hackers to lure innocents.

App Sandboxing Studies in app sandboxing and security enforcement can be classified into three types in views of different layers: (1) The use of Inline Reference Monitoring directly inserts the reference code to target apps' bytecode [35] [36] [37]. (2) Similar to DroidPill, [25] and [38] instrument DVM and native libraries to enforce policies via the hooking code. (3) [39] places reference code outside of target apps' processes, which is close to what Boxify [22] and NJAS [23] do. However, regarding to defensive systesms, they are either requied to modify the OS or lack of the strong reference monitoring properties.

App Repackaging In fact, DroidPill can be thought of a technique that a malware dynamically repackages target apps at runtime without need to carry the code and data. Although app repacking detection has been extensively studied in academia [40] [41] [42] [43], most focused on the detections based on static analysis, which is ineffective to screen DroidPill malware due to its lack of the code and data from target apps.

8. CONCLUSION

In this paper, we propose the App Confusion Attack, a more stealthy application-level attack than the existing malware schemes. It can simultaneously force multiple benign apps on a device to run in a virtual execution context controlled by the DroidPill malware using an exclusive app virtualization technique, instead of the native execution context provided by the Android Application Framework. Afterwards, we demonstrate five examples of how DroidPill can practically and effectively attack mobile users or app developers with two different attack vectors. Finally, we conclude with possible countermeasures to the App Confusion Attack.

9. REFERENCES

[1] A Murky Road Ahead for Android, Despite Market Dominance, http://www.nytimes.com/2015/05/28/technology/personaltech/a-murky-road-ahead-for-android-despite-market-dominance.html

[2] Number of Android Applications, http://www.appbrain.com/stats/number-of-android-apps

[3] 97% of Mobile Malware is on Android, http://www.forbes.com/sites/gordonkelly

[4] Zhou, Y., Jiang, X.: Dissecting Android Malware: Characterization and Evolution. Oakland 2012.

[5] Lindorfer, M. et al.: Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. BADGERS 2014.

[6] Google: Android Security 2014 Year in Review, https://static.googleusercontent.com/media/source.android.com/en//security/reports/Google_Android_Security_2014_Report_Final.pdf

[7] Simplocker: First Confirmed File-Encrypting Ransomeware for Android,

http://www.symantec.com/connect/blogs/simplocker-first-confirmed-file-encrypting-ransomware-android
[8] Stefanko, L.: Aggressive Android Ransomeware Spreading in the USA, http://www.welivesecurity.com/2015/09/10/aggressive-android-ransomware-spreading-in-the-usa/
[9] Ren, C. et al.: Towards Discovering and Understanding Task Hijacking in Android. USENIX Security 2015.
[10] Diao, W. et al.: No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. Oakland 2016.
[11] Bianchi, A. et al.: What the App is That? Deception and Countermeasures in the Android User Interface. Oakland 2015.
[12] Chen, Q. et al.: Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. USENIX Security 2014.
[13] Bobrov, O.: Certifi-Gate: Front Door Access to Pwning Millions of Android Devices. Blackhat USA 2015.
[14] Query String Injection: Android Provider, http://www.hpenterprisesecurity.com/vulncat/en/vulncat/java/query_string_injection_android_provider.html
[15] The Android Trojan Svpeng Now Capable of Mobile Phishing, http://securelist.com/blog/research/57301/the-android-trojan-svpeng-now-capable-of-mobile-phishing/
[16] Rasthofer, S. et al.: An Investigation of the Android/BadAccents Malware which Exploits a New Android Tapjacking Attack. Technical Report 2015.
[17] Zhou, W. et al.: Slembunk: an Evolving Android Trojan Family Targeting Users of Worldwide Banking Apps, https://www.fireeye.com/blog/threat-research/2015/12/slembunk_an_evolvin.html
[18] Jung, J. et al.: Repackaging Attack on Android Banking Applications and its Countermeasures. Wireless Personal Communications 2013.
[19] Google Bug Bounty, https://www.google.com/about/appsecurity/reward-program/
[20] HP Fortify Static Code Analyzer, http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/
[21] Masque Attack: All Your iOS Apps Belong to Us, https://www.fireeye.com/blog/threat-research/2014/11/masque-attack-all-your-ios-apps-belong-to-us.html
[22] Backes, M. et al.: Boxify: Full-Fledged App Sandboxing for Stock Android. USENIX Security 2015.
[23] Bianchi, A. et al.: NJAS: Sandboxing Unmodified Applications in Non-Rooted Devices Running Stock Android. SPSM 2015.
[24] Computer Security Technology Planning Study (Volume I), http://csrc.nist.gov/publications/history/ande72a.pdf
[25] Xu, R. et al.: Aurasium: Practical Policy Enforcement for Android Applications. USENIX Security 2012.
[26] Android Dashboards, https://developer.android.com/about/dashboards/index.html
[27] Sanz, B. et al.: PUMA: Permission Usage to Detect Malware in Android. CISIS-ICEUTE-SOCO 2012.
[28] Poeplau, S. et al.: Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. NDSS 2014.
[29] Costamagna, V., Zheng, C.: ARTDroid: A Virtual Method Hooking Framework on Android ART Runtime. IMPS 2016.
[30] AllHookInOne, https://github.com/boyliang/AllHookInOne
[31] App Licensing, http://developer.android.com/google/play/licensing/index.html
[32] Rutkowska, J.: Introducing Blue Pill, http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html
[33] Zovi, D.: Hardware Virtualization Rootkit. Blackhat USA 2006.
[34] King, S. et al.: SubVirt: Implementing Malware with Virtual Machines. Oakland 2006.
[35] Davis, B. and Chen, H.: RetroSkeleton: Retrofitting Android Apps. MobiSys 2013.
[36] Davis, B. et al.: I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. MoST 2012.
[37] Jeon, J. et al.: Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. SPSM 2012.
[38] Zhou, Y. et al.: Hybrid User-Level Sandboxing of Third-Party Android Apps. ASIACCS 2015.
[39] Russello, G. et al.: FireDroid: Hardening Security in Almost-Stock Android. ACSAC 2013.
[40] Zhou, W. et al.: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. CODASPY 2012.
[41] Shao, Y. et al.: Towards a Scalable Resource-Driven Approach for Detecting Repackaged Android Applications. ACSAC 2014.
[42] Zhou, W. et al.: Divilar: Diversifying Intermediate Language for Anti-Repackaging on Android Platform. CODASPY 2014.
[43] Chen, K. et al.: Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. USENIX Security 2015.