Journal of Computer Security 28 (2020) 269–293
DOI 10.3233/JCS-191325
IOS Press

Do not jail my app: Detecting the Android plugin environments by time lag contradiction

Yifang Wu [a,b], Jianjun Huang [a,b,*], Bin Liang [a,b] and Wenchang Shi [a,b]
[a] *School of Information, Renmin University of China, Beijing, China*
[b] *Key Laboratory of DEKE, Renmin University of China, MOE, China*
E-mails: lizhiwuyi@ruc.edu.cn, hjj@ruc.edu.cn, liangb@ruc.edu.cn, wenchang@ruc.edu.cn

Abstract. Many Android apps today face problems such as the large application package (APK) size, frequent updates, and so on. The Android plugin technology provides a solution for app developers, allowing a running app to dynamically load and execute a separate APK file without installing it in the system. These dynamically loaded APKs are called plugins. In Android app markets, many multi-instance apps abuse this technology to load normal social apps as plugins. While satisfying the users' demand for logging into multiple accounts simultaneously, it brings new security threats to the legitimate apps. Sensitive API invocations can be hijacked and private data becomes accessible to malicious multi-instance apps. Therefore, identifying the running environments becomes necessary. In this paper, we propose a novel detection mechanism, named PluginAssassin, to identify whether an app is running as a plugin. PluginAssassin uses the time ratio of different activity launching procedures to determine the running environment, conforming to the observed time lag contradiction phenomenon. We also present a mitigation mechanism for the ΔT attack specific to our approach. We collect 50 multi-instance apps from two app markets and implement PluginAssassin in five popular social apps. We assess the effectiveness on three devices and the experimental results show that PluginAssassin can detect plugin environments effectively.

Keywords: Android, plugin environment, time lag contradiction, time ratio, ΔT attack

1. Introduction

With the rapid development of the Android platform, people are heavily dependent on mobile applications (a.k.a., apps), which bring convenience to our daily life, study and work. In the meantime, apps tend to embody more and more functionalities, enlarging the size of the installation files (e.g., the APK files in Android). The plugin technology was proposed a few years ago, originally for reducing the size of installation files and providing an easier way for functionality update.[1] Although the dynamic code loading techniques [16,26,27,47] support dynamic loading of code snippets during an app's running, it has some limitations in resource update and components addition. The plugin technology allows an independent app or APK to be loaded within another app, skipping the typical installation process on devices. With this technology, modules can be decoupled from a giant app and developed in form of smaller apps. Each modular app is "installed" or "uninstalled" within the scope of the main app which

*Corresponding author. E-mail: hjj@ruc.edu.cn.
[1]Google I/O 2018 introduced the Android App Bundle [1] that can do the same job, but the open source plugin framework VirtualApp [39] was released even two years earlier.

consists of only the basic functionalities according to actual usage scenarios. Take a popular Android app, Baidu [7], as an example. As an app supporting mobile online search, the main APK file only contains basic functionalities such as searching, news push, user management and so on. Some marginal functionalities like novel reader, game center, wallet and music are enclosed in individual APK files and loaded on demand. By this means, the vendors can quickly update the functionalities or fix bugs in those plugged apps silently, instead of transferring a huge full-featured APK file and asking the users to install the updates frequently.

While app development benefits from the plugin technology, the abuse of this technology has been found in apps, which are referred to as *multi-instance* apps. A multi-instance app runs as a host app in the system and loads another legitimate app as a plugin by utilizing the plugin technology. This makes multiple instances of the legitimate app exist in the system at the same time. Users can leverage multi-instance apps to login to different accounts for the same social app on one device, eliminating the trouble of login and logout for accounts switching. Such usage scenario becomes popular recently. As we can see later in Table 5 in Section 4, multi-instance apps are popular and have attracted a large number of users. More specifically, Parallel Space [24] (#2) is one of the most popular in the Google Play store, with hundreds of millions installations. It's worth noting that the cost of developing a multi-instance app is low. There are two open-source plugin technology frameworks, DroidPlugin [13] and VirtualApp [39], which provide the basic support of loading arbitrary apps as plugins. The commercial version of VirtualApp is even compatible with the latest Android 10, i.e., being capable to bypass the restrictions on non-SDK interfaces starting in Android 9 [28]. Based on the capability of the technology, developers can implement other functionalities as well. For example, some multi-instance apps offer the function of mocking device information for plugin apps, such as the location and phone models.

Along with the convenience, being loaded as a plugin threatens the security of the legitimate apps. As a popular social app, Twitter [37] has already suffered from a variety of phishing attacks [31], which aim at stealing users' accounts and credentials. The researchers also found a new kind of malware, DualTwitter [35], which embeds a plugin technology framework, launching the normal Twitter app as a plugin. Because the plugin technology grants the ability to hijack all API calls in the plugin app to the host app, DualTwitter can steal the user's inputs by hijacking corresponding APIs and extract the login information. Besides, a multi-instance app can access all the files created by the plugin app, which eases the host app to acquire the private data that should be isolated by Android's security policy. To further illustrate the security threats to legitimate apps, researchers conduct several demo attacks to popular social apps [45]. Social apps usually store login tokens of user accounts in their private directories. By loading a social app as a plugin, adversaries can easily dump all the files under its private directory and then install the same social app on an emulator or another smartphone with all the private files of the victim app. Without explicitly obtaining the credential information such as the username and password, adversaries can login as the victim and take control of the account. In addition to the login tokens, chatting history or email content can also be leaked in a similar attack way. The researchers also point out that attackers can forge emails by loading the Gmail application to achieve more malicious purposes.

Adversaries may achieve the same purposes by repackaging target apps, but with the development of various code protection methods like code obfuscation [6,21] and packing techniques [14,46], it becomes more and more difficult. Compared with repackaged apps, of which users are concerned about the authenticity, a multi-instance app requires no change to the plugin apps and indeed attracts the users to load whatever they want. In other words, adversaries can do something evil more stealthily.

Considering these potential security threats, it is necessary for an app to detect whether it is running as a plugin. Plugin-Killer [23], to the best of our knowledge, is the only feasible solution for normal

Android apps to detect the plugin environments at present. Unfortunately, as we will elaborate in Section 2.3, Plugin-Killer can be bypassed easily because a multi-instance app can take full control of the plugin app. Hence, a more robust detection method to protect legitimate apps from being loaded as plugins is deem required.

In this paper, we propose a novel detection mechanism, named PluginAssassin, integrated to normal Android apps to detect the plugin environments. Our insight is that the plugin technology needs to interpose the Android component launching operation to bypass system restrictions, and this interference prolongs the time consumption. We focus on only one type of Android components and observe that launching an activity under two scenarios, in the same process and in a new process, has disproportionate time overhead when an app is running in plugin environments, resulting in a phenomenon that we call the "*time lag contradiction*". Based on this observation, PluginAssassin collects the elapsed time and computes the ratio of the activity launching under two scenarios. The ratio is compared with a predefined threshold to judge the type of current environment. Compared with Plugin-Killer, PluginAssassin involves much fewer APIs that do not directly describe an app's features, leaving less chance for plugin environments to bypass. However, we discuss the opportunity for a plugin environment to evade the detection and present the ΔT attack. PluginAssassin mitigates the challenge by probing the launching procedure to ensure the time authenticity and enhance the approach's reliability.

We evaluate PluginAssassin on five popular social apps and 50 real-world multi-instance apps. Three different Android smartphones are used in the experiments. We compute the threshold with a few randomly selected samples and apply it to all the other test cases. The results show that, except those cases in which the multi-instance apps do not support the functionalities required by the guest apps, PluginAssassin successfully identifies all plugin environments without false alarms or false negatives, demonstrating the effectiveness of our approach.

Our work makes the following major contributions:

- We present PluginAssassin, which detects the running environments based on the time lag contradiction of activity launching in different scenarios. PluginAssassin labels the plugin environments by comparing the computed time ratio with a given threshold.
- We discuss a potential challenge specific to PluginAssassin, i.e., the ΔT attack, and propose a mitigation mechanism which probes the existence of an attack along with the activity launching.
- We evaluate PluginAssassin on real-world social apps, multi-instance apps and devices. The results show that PluginAssassin accurately discovers the plugin environments.

The rest of this paper is structured as follows. Section 2 provides the technical background of the Android plugin technology, and gives a look at Plugin-Killer. In Section 3, we present our approach in detail, including a potential threat and its mitigation. Afterwards, Section 4 evaluates our approach and discusses the results. We discuss some limitations in Section 5 and survey the related work in Section 6. Finally, we conclude this paper in Section 7.

2. Background

In this section, we first demystify the Android plugin technology, presenting a high level overview. Then without losing generality, we describe how a host app utilizes the plugin technology to launch Android activities. At last, we talk about Plugin-Killer [23], a lightweight defense mechanism that prevents Android apps from being plugin apps running in a plugin environment. We also show how its defense solutions can be easily bypassed.

First of all, we clarify some terms that are commonly used in this paper. The *plugin environment* denotes a set of programs and resources which are necessary to execute another independent program (i.e., an Android app). A plugin environment can either be merely a framework for secondary development or a *host app* that already embodies the framework. The host apps are also called *multi-instance* apps. The *plugin environment* and the *host app* are interchangeably used in this paper. A *plugin app* is an Android app that runs within a host app. It is also used as *guest app* sometimes. We call launching a component within the same process an *intra-process* launching, while in an *inter-process* scenario, the component is started in a new process.

2.1. Basis for the plugin technology

Android apps typically contain many components such as *Activity*, *Service*, *Broadcast Receiver* and *Content Provider*, and run on top of the Android framework [3]. When an Android app (or an APK file) is installed on a device, the Android system maintains some records about the app and the registered components defined in `AndroidManifest.xml`. Such records are useful when the framework launches the app or one of its component, for instance, starting a new activity for editing emails in a mail app. Without installing an app means the corresponding records are missing and the Android framework is not able to discover the components within the app as they are not registered. The plugin technology addresses this issue, making it possible to run the apps not installed on the Android system, even if the Android framework does not know the existence of the guest app.

We examined the two open-source Android plugin frameworks, DroidPlugin [13] and VirtualApp [39], to figure out how they play tricks on the Android system and successfully execute guest apps. Figure 1 demonstrates the overview of the plugin technology. In general, the host app, which embeds a plugin framework, is installed on a device as a normal app. The embedded plugin framework provides the ability to manage the plugin apps and hook necessary API invocations. Besides, the plugin framework contains a special `AndroidManifest.xml`, which defines lots of stub components. The stub components are registered to the system as the host app installation. When launching an app or a component, the plugin framework hooks the interactions between the guest app and the Android framework, requesting the system to launch a stub component instead. The system checks succeed as the stub component has been

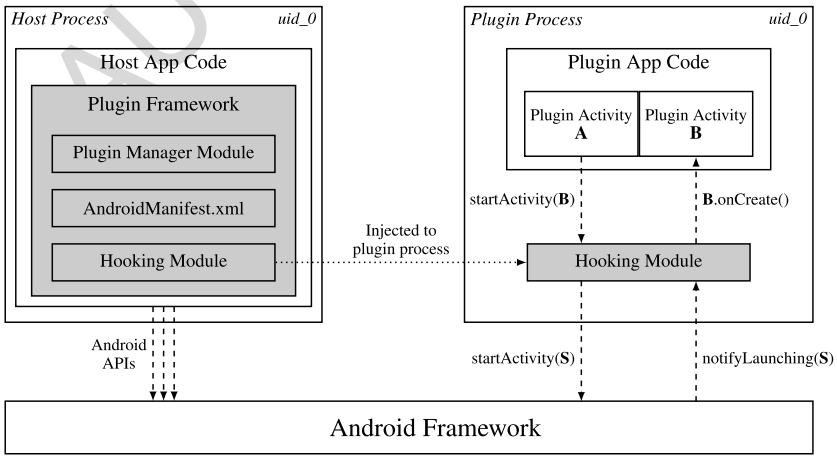

Fig. 1. Plugin framework overview.

registered, but subsequent operations are interfered by the plugin framework via hooking, replacing the stub component with the target component.

For example, as shown in Fig. 1, when an activity *A* in the plugin app is trying to start another activity *B* via `startActivity()`, the invocation falls into the hooking module, which requests the Android framework to launch a stub activity *S*. The Android framework performs some checks and then notifies the plugin process to instantiate and start *S*. The hooking module hijacks the response, and launches the original target activity *B* through the *reflection* mechanism. The Android framework does not interact with the plugin app directly, but with the help of the plugin framework, the graphical user interface associated with *A* is successfully transited to that for *B*.

In order to avoid the interference among different guest apps, the host app and each guest app usually run in different processes. However, since the Android sandbox mechanism prevents users – in most cases, each app is assigned an unique *uid* – from accessing resources of other users [4], in order for the plugin framework to interleave the execution of a guest app, they share the same *uid*. Hence, the sandbox mechanism is evaded and this allows the host app to access all resources and information associated with the guest apps, either public or private. Risks appear as adversaries can leverage this feature to steal the user's privacy data, as done in DualTwitter [35], or perform many other unexpected behaviors [45].

2.2. *Launching plugin activities*

We omit the implementation details of how the plugin environment loads specific classes and resources from a plugin app [23]. For our paper, we describe the conceptual workflow of launching an activity, which motivates our solution for detecting the plugin environments.

Figure 2 illustrates the normal flow of launching an activity in Android. When an activity *A* tries to launch another activity *B*, it invokes an API like `startActivity()` and sends an `Intent` message to `Activity Manager Service` (a.k.a., AMS) via `Binder`, the Android-specific inter-process communication mechanism [2]. Once the request arrives, AMS, running in the process *system_server*, performs a series of checks, for example, checking whether the target activity *B* has been registered to the system. AMS also deals with the management of activity stacks and tasks. Typically, after pausing the top activity *A*, AMS sends a LAUNCH_ACTIVITY message to the app process. A callback `handleLaunchActivity()` responds the message. The app process instantiates the target activity *B* and executes the corresponding functions, switching the user interface. It is notable that, if *B* is declared to

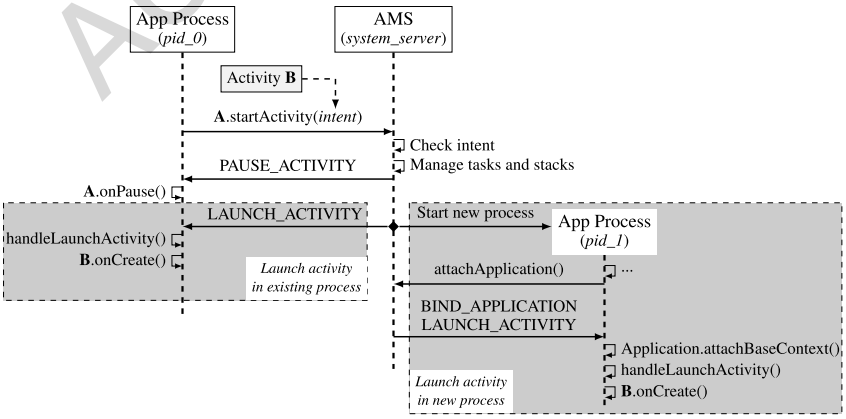

Fig. 2. Launching an activity in Android.

be launched in a new process (with "`android:process`" property in its declaration), AMS notifies the system to fork a new process. The app's *Application* is then initialized within the new process, and afterwards, similar activity instantiation and execution are done.

The above workflow fails when the app is indeed a guest app running in a plugin environment as the app is not installed and the target activity *B* is not known to the system. AMS checks fail if the `Intent` message is directly sent to AMS. The plugin technology introduces "component replacement" to bypass the AMS checks. As mentioned earlier, the plugin environment defines many stub components in `AndroidManifest.xml` and registers them to the system when installing the host app. When a plugin component is about to be launched, the host app creates a new `Intent` wrapping a stub component, which is sent to AMS. Without loss of generality, we depict below how a host app launches an activity in the plugin app.

In real-world Android apps, most activities are launched in the same process. Figure 3 illustrates how to launch a plugin activity *B* within the same process. We aggregate the injected operations into two grey areas (① and ②). The hooking module, once receiving a request for launching an activity, replaces *B* with a stub one, say *Stub01*, which is defined in `AndroidManifest.xml` of the host app, and wraps a new `Intent` to AMS. After AMS checks succeed, the hijacked callback provided by the plugin environment replaces the wrapped `Intent` with the original one and launches *B*.

Figure 4 shows the flow of launching a plugin activity *C* in a new process. The procedure before AMS responds is nearly the same as that in Fig. 3. But when AMS checks succeed, a new process (*pid_3*) is created, within which the guest app's application needs to be initialized. However, as the system cannot

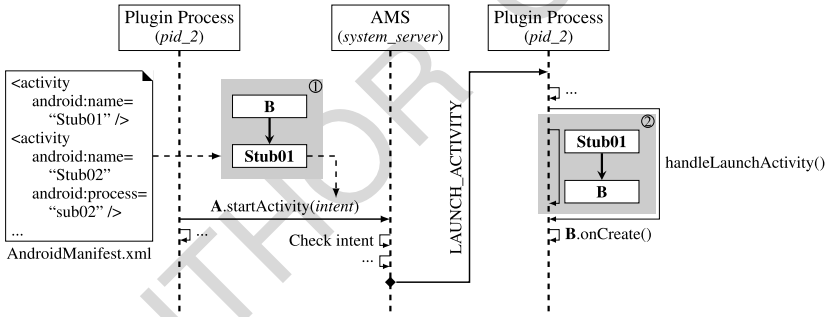

Fig. 3. Launching an activity in the same process (extra code in grey).

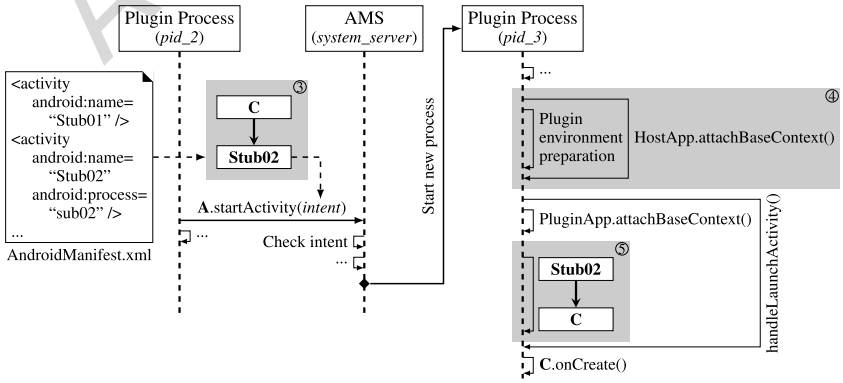

Fig. 4. Launching an activity in a new process (extra code in grey).

see the plugin app, it initializes the host app's application, which prepares the plugin environment for the new process. Then the plugin environment initializes the guest application, unwraps the replaced `Intent` and uses the original one to start the actual target *C*. Essentially, three extra phases are injected to the flow (③, ④ and ⑤).

2.3. Plugin-Killer

Luo *et al.* present Plugin-Killer [23], a lightweight defense mechanism, to prevent Android apps from being launched as plugin apps. Embedded within normal apps, it can detect the existence of plugin environments and terminate the enclosing apps if needed. The intuition behind Plugin-Killer is that, when an app is running as a plugin, there are definitely some differences in the running environment with the one created by system. Hence, Plugin-Killer collects and compares certain information with the counterpart under system environment. Any inconsistency indicates that the app is running in plugin environments. Below we will discuss the methods proposed by Plugin-Killer and show why they are not resistant to attacks.

By introspecting the code of Plugin-Killer, we find four types of information are collected in detection, the process name, the component name, the private directory path and the permission information (see the first column in Table 1).

For the process name, Plugin-Killer obtains all the running processes that share the same *uid* with itself, and checks their names. Generally, a process's name contains the package name of the corresponding app. However, in plugin environment, the host app process is also running with the same *uid*, so Plugin-Killer can detect the existence of the host app that has a different process name. The component name checking is conducted in a similar way. In plugin environments, the name of a plugin component will be the stub component name defined in the host app. For example, a service `AntiCheckMainService` launched by Plugin-Killer will have a name of `ServiceStub$StubP01$P00` due to the "component replacement" introduced by the DroidPlugin framework. Plugin-Killer considers the running environment as a plugin environment if it finds a wrong name of a component. In addition, since a plugin app is not installed by the system, it does not have its own private directory which is generally located in */data/data/{package_name}/*. All the file accesses will be redirected to another directory owned by the host app. For example, DroidPlugin creates a separate directory located in */data/data/{host_app_package_name}/Plugin/{plugin_app_package_name}/data/* as the private directory of each plugin app. If the private file path does not start with "*/data/data/{package_name}/*", Plugin-Killer terminates the app's running. Moreover, in Android, apps need to declare specific permissions to do some sensitive operations, such as sending SMS. To successfully execute arbitrary guest apps, host apps tend to declare quite a lot of permissions, generally much more than what a plugin app requires. To detect plugin environments, Plugin-Killer checks if an undeclared permission is granted at runtime. If so, it indicates that the app is running as a guest because the host app have declared the unexpected permission.

While the proposed approaches can be effective to some extent, they can be easily bypassed by the plugin environments. To collect such running environment information, some specific Android APIs are invoked by Plugin-Killer. Table 1 lists the APIs to obtain the information under the system environment and the DroidPlugin environment. Suppose the package name of the host app and the plugin app are *com.example.hostapp* and *com.example.testapp*, respectively. As described in Section 2.2, the `startActivity()` API is hijacked by the plugin environment to launch the plugin's activity. Similarly, the APIs in Table 1 can also be hijacked. When Plugin-Killer invokes these APIs, the plugin environment modifies their return values and replays them with the values under the system environment. Take

Table 1
Information collected by Plugin-Killer with related APIs and values under different running environments

Information type	Related API	Information value	
		System environment	DroidPlugin environment
Process name	ActivityManager.getRunningAppProcesses()	com.example.testapp	com.example.testapp com.example.hostapp
Component name	ActivityManager.getRunningServices()	com.example.testapp.AntiCheckMainSerivice	com.morgoo.droidplugin.stub.ServiceStub$StubP01$P00
Private directory filepath	PackageManager.getApplicationInfo()	/data/data/com.example.testapp/	/data/data/com.example.hostapp/Plugin/com.example.testapp/data/
Permission information	PackageManager.checkPermission(android.permission.SEND_SMS, ...)	PackageManager.PERMISSION_DENIED	PackageManager.PERMISSION_GRANTED

the component name as an example. After the plugin environment gets the actual return value of `ActivityManager.getRunningServices()`, it extracts the stub component name and finds the original target component of the plugin app. Then, the plugin environment initializes a new object based on the target component and returns it to the plugin app. Plugin-Killer cannot distinguish the system environment and the plugin environment if all the information in Table 1 are replayed in such ways. In other words, Plugin-Killer is not robust enough to defend the real-world plugin technology. We also empirically evaluated the attacks with an implementation on top of DroidPlugin and all of the defense techniques provided by Plugin-Killer are successfully bypassed.

3. Approach: PluginAssassin

In this section, we propose PluginAssassin based on the observation of *time lag contradiction* between the intra-process and inter-process activity launching in plugin environments. We will first present the plugin environment capability assumptions, after which we discuss our observations and why some thoughts do not work. Then we detail our approach, a potential threat and the corresponding mitigation. At the end, we summarize how PluginAssassin is generally implemented.

3.1. Threat model

PluginAssassin is designed to detect whether an app is running in a plugin environment. Considering the nature of attack-defense countermeasures, we cannot expect to solve the problem once and for all. Therefore, we assume the plugin environment has the following capabilities:

- The host app can hook all the API invocations in a plugin app and modify the actual arguments and results. In the meantime, the guest app's utility must be preserved.
- The host app can only run as an ordinary user in Android, i.e., it is installed as a third-party app instead of a system or privileged app.

The first requirement relaxes the restriction, which invalidates the Plugin-Killer solutions as we discussed in Section 2.3. Though it gives the plugin environments the best ability to interfere the guest app, we think the major purpose of a plugin environment is to run Android apps as usual instead of running into crashes or malfunction. As a consequence, in this paper, we require the plugin environments to

preserve the utilities of guest apps. The second assumption limits the ability of the host app to interpose other apps and the kernel's running.

3.2. Road to success

Recall the flow of launching a plugin activity in Fig. 3 and Fig. 4. Apparently, the extra embedded functionalities, such as component replacement for both scenarios (①, ③, ④ and ⑤) and plugin environment preparation for inter-process launching (②) cost additional time compared with the normal flows of launching an activity. The keyword in our observation is "*time*" – the time is a potential property to distinguish the system environment and the plugin environment.

At first, we try to get clues from the *absolute time* an interesting operation consumes. We represent the time period from invoking `startActivity()` to entering `onCreate()` in the target activity as T_{sp} and T_{np}, respectively for intra-process and inter-process scenarios. We use T_i ($1 \leqslant i \leqslant 5$) to denote the time cost of the ith numbered block in Fig. 3 and Fig. 4. Moreover, $T_{\mathcal{S}}$ and $T_{\mathcal{N}}$, separately for the two types of activity launching, express the sum of the time required by all other events than the embedded code, such as executing the plugin app's own code, process scheduling, and so on. We have the following equations:

$$T_{sp} = \sum_{1 \leqslant i \leqslant 2} T_i + T_{\mathcal{S}}, \qquad T_{np} = \sum_{3 \leqslant i \leqslant 5} T_i + T_{\mathcal{N}}. \tag{1}$$

We are able to measure T_{sp} and T_{np} via APIs like `currentTimeMillis()`. However, we do not have oracles about the precise and tight ranges of $T_{\mathcal{S}}$ and $T_{\mathcal{N}}$. Many factors can affect their values, for example, different hardwares, different Android releases, and even different system states at runtime. The uncertainty of time measurements makes it impossible to devise a technique utilizing the absolutely elapsed time of the procedures.

Later, we discovered that APIs like `clock_gettime()` allow apps to obtain the real CPU time for current process (e.g., the new process) or even a thread in the current process (e.g., the main thread launching an activity). Even though scheduling, AMS and any other out-of-process effects are excluded, the problem still exists as we cannot determine a proper threshold which limits how long a target operation or procedure can take at most. We know that even the same piece of code can cost different time in different situations. One threshold does not support the detection in arbitrary devices but app developers are unable to learn many thresholds for all situations.

While the absolute time doesn't help, we turn our mind to the *relative time*. Our research question emits: "*Can we compute such a time ratio of two procedures, satisfying that: (1) at least one procedure involves the code injected by the host apps in plugin environments, i.e., activity launching, and (2) the ratios for different situations are well distributed such that a simple threshold can easily separate those for system environments and plugin environments?*"

To answer the question, we conducted some pilot studies and collected amounts of data, which shed some light on the situation, enlightening us to propose an approach based on the observation of the *time lag contradiction*.

3.3. Time lag contradiction

As we discussed previously, the plugin environment introduces extra code for both intra-process and inter-process activity launching during the guest app's execution, which prolongs the time consumption.

We observed that the plugin environments do not impose proportional time overhead on the two activity launching scenarios. Formally, we have:

$$\frac{T_{sp}(\text{plugin environment})}{T_{sp}(\text{system environment})} \not\approx \frac{T_{np}(\text{plugin environment})}{T_{np}(\text{system environment})}. \tag{2}$$

We call it the *time lag contradiction*, which can be used as an indicator of the environment identification. Given the fact that an app cannot measure the time for both system environment and plugin environment in the meantime, we convert the inequation to:

$$\frac{T_{np}(\text{system environment})}{T_{sp}(\text{system environment})} \not\approx \frac{T_{np}(\text{plugin environment})}{T_{sp}(\text{plugin environment})}. \tag{3}$$

Now, either side denotes a time ratio and is measurable during one execution.

For intra-process launching, we use $\mathcal{T}_{sp}$ to denote the CPU time for merely the app's main thread, from invoking `startActivity()` to entering `onCreate()`, covering both T_1 and T_2. We are unable to intercept the procedure, as from the guest app's point of view, only these two locations are visible for gathering the timestamps. For inter-process launching, `startActivity()` is invoked in process *pid_2* while `onCreate()` is executed in process *pid_3*, as shown in Fig. 4. To simplify the time data collection, we record the elapsed CPU time when the app is initialized in the new process, namely, within the constructor of the app's *Application* class, notated as $\mathcal{T}_{np}$, merely T_4 for the embedded functionalities included. Only some process initialization and the plugin environment preparation affect the value. By this means, we exclude the influence of different app implementations, only taking into account the environmental part that the app developers cannot control. We compute the ratio $\mathcal{R}$ as follows:

$$\mathcal{R} = \frac{\mathcal{T}_{np}}{\mathcal{T}_{sp}} \tag{4}$$

which, according to our observations, shows significant difference under plugin environments compared to that under system environments.

Table 2 shows the experimental results of running a simple demo app in a system environment and two plugin environments powered by DroidPlugin and VirtualApp separately, three times for each. The experiments are all conducted on an OPPO A37m smartphone with Android 5.1 running. The observed results clearly satisfy our anticipation of the time lag contradiction. $\mathcal{T}_{sp}$ in plugin environments exceeds that in system environment by at most 42%, and the optimization in VirtualApp makes the difference much smaller. However, $\mathcal{T}_{np}$ is at least seven times bigger in plugin environments than in system environment. Using $\mathcal{T}^{\diamond}$ to represent the time in the plugin environments and $\mathcal{T}^{\star}$ to denote the time in the

Table 2
Time costs of launching an activity in different environments (time unit: ns)

System environment			DroidPlugin			VirtualApp		
$\mathcal{T}_{sp}$	$\mathcal{T}_{np}$	$\mathcal{R}$	$\mathcal{T}_{sp}$	$\mathcal{T}_{np}$	$\mathcal{R}$	$\mathcal{T}_{sp}$	$\mathcal{T}_{np}$	$\mathcal{R}$
73 970 399	51 412 918	0.695	95 306 310	368 260 387	3.864	74 159 693	431 600 699	5.820
67 412 461	45 844 460	0.680	76 372 079	357 651 841	4.683	71 673 463	421 793 686	5.885
69 121 615	50 122 961	0.725	89 919 231	382 009 538	4.248	71 229 309	424 999 235	5.967

system environment, for all the observed $(\mathcal{T}_{sp}^{\diamond}, \mathcal{T}_{np}^{\diamond})$ and $(\mathcal{T}_{sp}^{\star}, \mathcal{T}_{np}^{\star})$, the following formula always holds:

$$\exists \alpha \geqslant 6, \beta \leqslant 0.5, \quad \begin{cases} \mathcal{T}_{np}^{\diamond} > \mathcal{T}_{np}^{\star} + \alpha \mathcal{T}_{np}^{\star}, \\ \mathcal{T}_{sp}^{\diamond} < \mathcal{T}_{sp}^{\star} + \beta \mathcal{T}_{sp}^{\star} \end{cases} \Rightarrow \quad \frac{\mathcal{T}_{np}^{\diamond}}{\mathcal{T}_{sp}^{\diamond}} > \frac{\mathcal{T}_{np}^{\star} + \alpha \mathcal{T}_{np}^{\star}}{\mathcal{T}_{sp}^{\star} + \beta \mathcal{T}_{sp}^{\star}} > \frac{\mathcal{T}_{np}^{\star}}{\mathcal{T}_{sp}^{\star}}. \tag{5}$$

Our goal is to choose a desirable threshold $\mathcal{R}_b$ that satisfies:

$$\forall \left(\mathcal{T}_{sp}^{\diamond}, \mathcal{T}_{np}^{\diamond}\right), \left(\mathcal{T}_{sp}^{\star}, \mathcal{T}_{np}^{\star}\right), \quad \frac{\mathcal{T}_{np}^{\diamond}}{\mathcal{T}_{sp}^{\diamond}} > \mathcal{R}_b > \frac{\mathcal{T}_{np}^{\star}}{\mathcal{T}_{sp}^{\star}}. \tag{6}$$

According to Table 2, we can simply set $\mathcal{R}_b = 1$, which perfectly differentiates the system environment from two plugin environments. We will show in Section 4.3 how to scientifically determine $\mathcal{R}_b$ for real-world Android apps and evaluate its effectiveness in other devices as well as in many other plugin environments.

3.4. ΔT attack and mitigation

While the preliminary results show the feasibility of our time lag contradiction approach, it suffers from a simple attack that can be easily implemented. Due to the hooking capability, the hooking module provided by plugin environment can modify the return value of time acquisition APIs after a hooked functionality is done, by subtracting the time consumed by the injected code from the actual value. The plugin environment developers are able to measure T_i that stands for the increment of the execution time. We denote the corresponding increments as $\Delta T_{sp} = T_1 + T_2$ and $\Delta T_{np} = T_4$. If a plugin environment collects these values and interferes the APIs, we get in the plugin environment that:

$$\mathcal{R}' = \frac{\mathcal{T}'_{np}}{\mathcal{T}'_{sp}} = \frac{\mathcal{T}_{np}^{\diamond} - \Delta T_{np}}{\mathcal{T}_{sp}^{\diamond} - \Delta T_{sp}} \approx \frac{\mathcal{T}_{np}^{\star}}{\mathcal{T}_{sp}^{\star}}, \tag{7}$$

where $\mathcal{T}'$ denotes the intervened time value, $\mathcal{T}^{\diamond}$ represents the actual time not modified in the plugin environment and $\mathcal{T}^{\star}$ expresses the corresponding time in the system environment. As a plugin activity launching consists of the normal launching procedure plus the injected behaviors, it is reasonable to presume that $\mathcal{T}^{\star} \approx \mathcal{T}^{\diamond} - \Delta T$. As a result, the above formula indicates that with a simple interference, the plugin environments can bypass our detection because the ratio in plugin environments approximates that in system environments. We call this threat the ΔT *attack*. The left part of Fig. 5 conceptually shows how a ΔT attack is performed in a plugin environment.

To mitigate the ΔT attack, we adopt a probing strategy to verify the authenticity of $\mathcal{T}_{sp}$ and $\mathcal{T}_{np}$. If either is considered fabricated, we say a ΔT attack happens, which further infers that the app is currently running as a guest app and should be terminated immediately. Our ΔT mitigation mechanism assumes that each ΔT is only visible to the process that calculates it. Namely, it won't be transferred to other processes. Otherwise, the ΔT attack can be devised elaborately and intricately to be more difficult to resist.

As shown in Fig. 5, a probe thread, is started before `startActivity()`, running along with the testing thread that launches an activity until the elapsed time is collected. The probe thread launches a

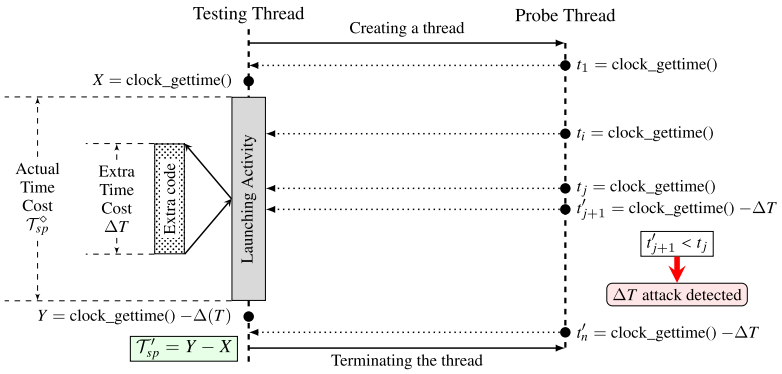

Fig. 5. Intra-process ΔT attack and detection.

series of requests to acquire the CPU time of the testing thread via `clock_gettime()`. The interval $\mathcal{H}$ between every two adjacent requests is smaller than a typical ΔT. Generally, we claim that

$$t_{i+1} = t_i + \mathcal{H}, \tag{8}$$

where t_i denotes the ith time request. Once a ΔT attack occurs, i.e., the plugin environment camouflages the time value by subtracting ΔT from the actual time value, the following formula is satisfied given that $\mathcal{H} < \Delta T$:

$$\exists j, \quad t'_{j+1} = t_{j+1} - \Delta T = t_j + \mathcal{H} - \Delta T < t_j \tag{9}$$

in which t' represents the interfered time and t denotes the actual value. Both t_{j+1} and t_j are observed by the probe thread. Such a case violates the natural law in a timeline, so upon its occurrence, we can directly report the existence of a ΔT attack in intra-process launching and terminate the running app.

However, this time-probing method fails in the inter-process launching scenario, because the system call `clock_gettime()` does not support to acquire the CPU time of other processes. A feasible method to judge the validity of $\mathcal{T}_{np}$ is to compare $\mathcal{T}_{np}$ with the execution time collected from the `proc` file system. Given a process ID, a specific record file located at "*/proc/{pid}/stat*" provides two fields, `utime` and `stime`, illustrating the scheduled time of the process in user mode and kernel mode respectively. The sum of these two values approximates the execution time of the process. It is notable that `utime` and `stime` are measured in *jiffies* (or clock ticks), so before the comparison, we convert the tick numbers to general time values that are measured in nanoseconds using the following transforming relationship:

$$\mathcal{T}_{\text{proc}} = \frac{\mathcal{N}_{\text{user}} + \mathcal{N}_{\text{sys}}}{HZ} \times 1\,000\,000\,000, \tag{10}$$

where $\mathcal{T}_{\text{proc}}$ stands for estimated CPU time of the process, $\mathcal{N}_{\text{user}}$ and $\mathcal{N}_{\text{sys}}$ denote the clock ticks in `utime` and `stime`. Moreover, HZ is a constant value representing the number of ticks per second, which is 100 in Android.

The probe thread behaves differently to an inter-process scenario. It is created in the main process right after `startActivity()` and iteratively queries the system for the PID of the new process based on its process name that is predefined in `AndroidManifest.xml`. Although this phase involves some

Android APIs that can also be hijacked, in order to preserve the utility of the guest app, the actual PID is returned generally. Once the PID is obtained, the estimated execution time $\mathcal{T}_{\text{proc}}$ of the new process can be calculated based on the observed `utime` and `stime`. At this moment, the new process has not reached the point of collecting $\mathcal{T}_{np}$, leading to a smaller $\mathcal{T}_{\text{proc}}$ than $\mathcal{T}_{np}$. However, it serves as a reasonable indicator to verify $\mathcal{T}_{np}$ as our experiments and inspection of the plugin environment implementations show that most jobs have been finished before the plugin environment replaces the stub activity process name with the target activity process name. The replacement lets the probe thread be able to query the PID with the process name and soon after that, the point of collecting $\mathcal{T}_{np}$ is reached. Thus, $\mathcal{T}_{\text{proc}}$ is generally slightly smaller than an unmodified $\mathcal{T}_{np}$.

Because of the invisibility of ΔT_{np} to the main process, the plugin environment cannot forge $\mathcal{T}_{\text{proc}}$ into a rational value. If a plugin environment implements the ΔT attack in the new process, a huge increment (a.k.a. ΔT_{np}) is subtracted from $\mathcal{T}_{np}$, based on the observations of Table 2. A satisfiable relationship $\mathcal{T}'_{np} < \mathcal{T}_{\text{proc}}$ indicates the existence of a ΔT attack in the inter-process launching.

If neither ΔT attack is detected, we consider the collected time values, $\mathcal{T}_{sp}$ and $\mathcal{T}_{np}$, not counterfeited and then apply them for plugin environment detection.

So far, we have not found any plugin environment implementing the ΔT attack. To verify the validity of both the attack and defense approaches, we conducted two pilot experiments and present the results in Section 4.1.

3.5. *PluginAssassin*

We present PluginAssassin to implement the time lag contradiction detection technique, with the ΔT mitigation mechanism equipped. According to above discussions, we show the complete flow diagram in Fig. 6.

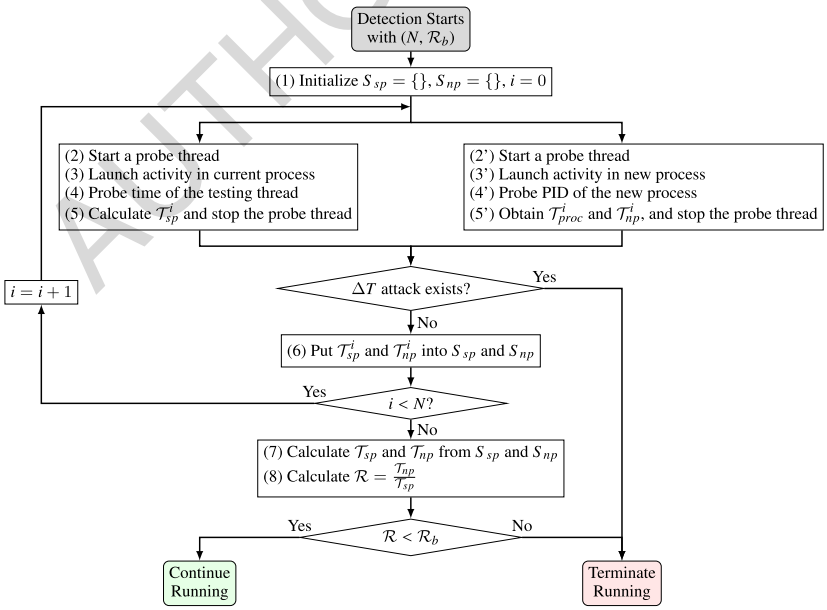

Fig. 6. Flow of PluginAssassin.

Conceptually, the intra-process and inter-process procedures can run in parallel. To reduce the impact of any noise as much as possible, we choose to collect multiple time values and finally compute the ratio with their averages. We define in the diagram the iterations of collecting time data as N and the ratio threshold as $\mathcal{R}_b$. And at the beginning, we initialize two empty sets S_{sp} and S_{np} for keeping the multiple records of time values, for intra-process and inter-process scenarios separately.

For intra-process launching ($2 \sim 5$), a probe thread is created, followed by the activity launching. After the activity is started, PluginAssassin calculates $\mathcal{T}_{sp}^{i}$, the consumed time of the ith collection. If any ΔT attach is detected according to our defense mechanism, the app is directly terminated. Otherwise, PluginAssassin puts $\mathcal{T}_{sp}^{i}$ into S_{sp}.

For inter-process launching ($2' \sim 5'$), we start a probe thread and try to start an activity in a new process. Within the probe thread, we iteratively query the PID of the new process, which allows PluginAssassin to obtain the jiffies of the new process and calculate $\mathcal{T}_{\text{proc}}^{i}$. $\mathcal{T}_{np}^{i}$ is collected in the new process and the probe thread is stopped. If no ΔT attack is found, $\mathcal{T}_{np}^{i}$ is inserted into S_{np}.

The above steps are repeated for N rounds and the averages of S_{sp} and S_{np} are computed as $\mathcal{T}_{sp}$ and $\mathcal{T}_{np}$ respectively. Then PluginAssassin calculates the ratio $\mathcal{R}$ and compare it with the given threshold $\mathcal{R}_b$. A larger $\mathcal{R}$ indicates a plugin environment, so the app is terminated. Otherwise, we think the app is not running as a plugin app and thus the app continues running.

4. Evaluation

We conducted all our experiments on three real smartphones to investigate the effectiveness of PluginAssassin across devices. The detail configurations of these smartphones are listed in Table 3.

We discuss below first the pilot experiments for the ΔT attack, followed by the description of the real-world apps we used to evaluate PluginAssassin. Then we show how to choose a proper threshold and present the evaluation results.

4.1. ΔT experiments

As mentioned in Section 3.4, we conducted two preliminary experiments on some demo apps with the OPPO smartphone to show the power of a ΔT attack and the effectiveness of the defense mechanism.

In the first experiment, we prepare two plugin environments. Both are powered by DroidPlugin, one unchanged and the other roughly instrumented with the ΔT attack technique. The demo app is equipped with the naive time lag contradiction detection but without the previously mentioned ΔT defense mechanism. From Table 4, we see that without the ΔT attack implemented in DroidPlugin, the system environment and the plugin environment can be easily differentiated with the ratio comparison (0.681 versus 4.205), similar with the results in Section 3.3. But when DroidPlugin is augmented with the ΔT bypassing mechanism, the demo app generates a relatively low ratio (0.765), close to that for the system

Table 3
Configurations of smartphones used in the experiments

Model	Android version	CPU	Memory
OPPO A37m	5.1	MT6750	2 GB
HUAWEI CRR-UL00	6.0	Hisilicon Kirin 935	3 GB
HUAWEI Mate 10	8.0	Hisilicon Kirin 970	6 GB

Table 4
Validating ΔT attack

Environment	Values		
	$\mathcal{T}_{sp}$ (unit: ns)	$\mathcal{T}_{np}$ (unit: ns)	$\mathcal{R}$
System environment	65 802 385	44 787 925	0.681
DroidPlugin (without ΔT attack)	82 273 470	345 948 923	4.205
DroidPlugin (with ΔT attack)	70 049 002	53 597 387	0.765

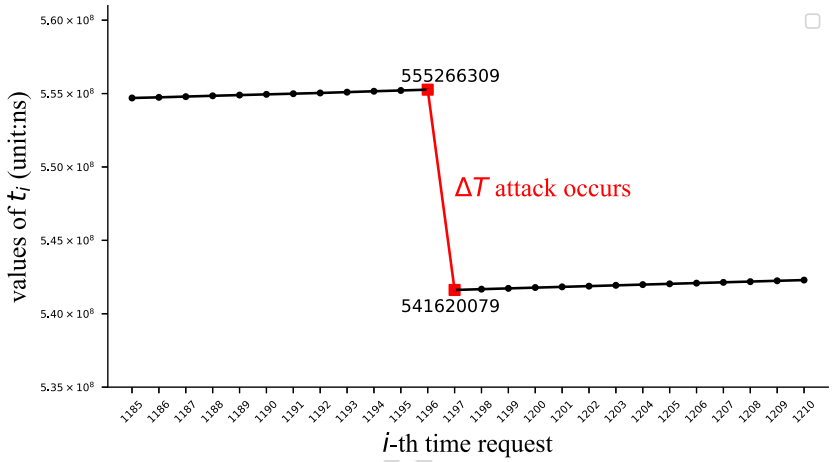

Fig. 7. Preliminary ΔT attack results.

environment (0.681) but far away from the one for the pure plugin environment (4.205). We are not confident to distinguish the plugin environment from the system environment in this case.

The second experiment tests whether the ΔT defense can be effective. We equip the demo app with the defense technique and run it in the instrumented DroidPlugin. For the intra-process scenario, we simulate the interval $\mathcal{H}$ with an empty 10 000-loop, about 50 microseconds in the experiment. Figure 7 shows the time values of the testing thread acquired by the probe thread. We see an obvious declining, more than 13 milliseconds, from t_{1196} to t_{1197}, indicating the existence of a ΔT attack. For inter-process activity launching, we dump $\mathcal{T}_{\text{proc}}$ and $\mathcal{T}_{np}$ acquired in the main process and new process separately. In this experiment, when a ΔT attack happens in the new process, the fetched `utime` and `stime` are 32 and 7, meaning that the collected $\mathcal{T}_{np}$ should be around and in most cases at least 390 milliseconds. However, the obtained value in the new process is about 50 milliseconds, resulting in an apparent inconsistency between $\mathcal{T}_{\text{proc}}$ and $\mathcal{T}_{np}$.

By far, the effectiveness of the ΔT attack and the corresponding mitigation approach are well demonstrated.

4.2. Experimental settings

To evaluate the capability of PluginAssassin to detect plugin environments, we chose five popular social apps, WeChat [41], Twitter [37], Instagram [18], WhatsApp [43], and LINE [22], to deploy PluginAssassin, and ran them in plugin environments. These social apps have hundreds of millions of down-

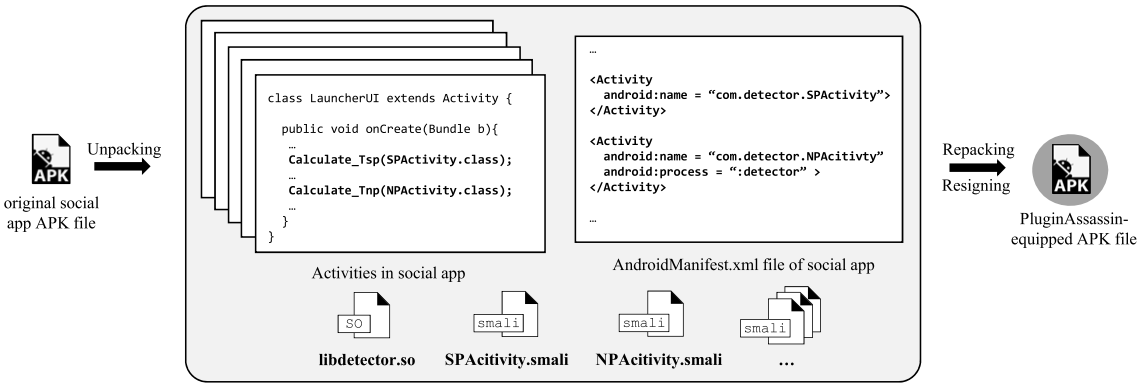

Fig. 8. Repacking social apps with PluginAssassin.

loads on Android app stores. However, because of the limitation of simultaneous login to multiple user accounts in one device, they become the first targets of multi-instance apps.

We repackaged the social apps with PluginAssassin. Figure 8 shows how PluginAssassin is integrated into social apps. First, for each social app, we obtain its original APK file and unpack it with `ApkTool` [5], generating `smali` files for all classes. We declare two additional activities `SPActivity` and `NPActivity` in its `AndroidManifest.xml`, for intra-process and inter-process launching separately. In addition, we inject a shared library file (*libdetector.so*) to implement the timestamp acquisition and time overhead calculation in native code. Because PluginAssassin requires N iterations to balance the collected time values (see Section 3.5), we disperse the operations of starting the testing activities into several existing activities of the social apps. We set the number N of iterations to 5 for all following experiments, as we have observed in earlier experiments on the demo app that, five iterations will not introduce high overhead but are enough to capture the timing characteristics of different types of activity launches, even after eliminating the outlier data.The app developers can choose an app-specific N when adopting our approach. During each test for a social app, for the reason that missed cache of the target activity can result in higher time cost for the first launch than the rest in the demo app experiments, we discard a pair of outliers, typically a maximum value and a minimum value, from S_{sp} and S_{np}, individually, and then average the more balanced remaining values for $\mathcal{T}_{sp}$ and $\mathcal{T}_{np}$. In next section, we will discuss how to determine a practical threshold $\mathcal{R}_b$. In order not to affect the social app's normal running, `SPActivity` and `NPActivity` are declared with transparent themes, making our detection operations invisible to users, and call `finish()` to explicitly terminate their instances after the time overhead is calculated. After we repack and resign all these files, an APK file equipped with PluginAssassin is generated.

We collected 29 multi-instance apps from the Google Play store and 21 multi-instance apps from the YingYongBao store (a third party Android app store in China) by searching with the keywords such as "multiple accounts". They provide the plugin environments in our experiments. In general, these apps scan the installed social apps in system and load the ones upon user's choice. Some of them support to load external APK files on device. We also found that a number of the multi-instance apps display advertisements before launching guest apps. We installed these multi-instance apps in the testing smartphones as unprivileged host apps, which will load the five security-enhanced social apps as plugins to evaluate PluginAssassin.

4.3. Threshold determination

PluginAssassin collects $\mathcal{T}_{sp}$ and $\mathcal{T}_{np}$ at runtime, computes the time ratio $\mathcal{R}$ and judges whether the running environment is a plugin environment or not based on the ratio. Thus, a predefined threshold $\mathcal{R}_b$ is required to perform a comparison. We show in Section 3.3 that simply setting $\mathcal{R}_b = 1$ works well for the demo experiments to distinguish plugin environments from a system environment. In practice, distinctive hardware and software configurations can result in varying degrees of time cost for the same operations across devices. Probably the threshold we computed based on merely the experimental data on OPPO cannot differentiate the running environments on other smartphones. Moreover, the specific implementations of the plugin environment may also influence the value of $\mathcal{R}$ at runtime.

We propose a straightforward calculation for $\mathcal{R}_b$. Our experimental observations show that, given a number of collected time values (averages for each test), the ratio $\mathcal{R}^{\diamond}$ for starting activities in a plugin environment is always greater than the ratio $\mathcal{R}^{\star}$ for launching activities within a system environment. Putting the time values in a two-dimensional coordinate system, X-axis for $\mathcal{T}_{sp}$ and Y-axis for $\mathcal{T}_{np}$, the slope of the line connecting a point and the origin O represents a time ratio $\mathcal{R}$. We pick the point with the minimum slope $\widetilde{\mathcal{R}}^{\diamond}$ and the one with the maximum slope $\widetilde{\mathcal{R}}^{\star}$, corresponding to point $S(\widetilde{\mathcal{T}}^{\diamond}_{sp}, \widetilde{\mathcal{T}}^{\diamond}_{np})$ and point $P(\widetilde{\mathcal{T}}^{\star}_{sp}, \widetilde{\mathcal{T}}^{\star}_{np})$, respectively. The angular bisector equally divides $\angle SOP$. Hence, its slope is a good choice of the threshold. Mathematically, the slope of the angular bisector is computed as:

$$\mathcal{R}_b = \frac{\widetilde{\mathcal{T}}^{\diamond}_{np}\sqrt{(\widetilde{\mathcal{T}}^{\star}_{sp})^2 + (\widetilde{\mathcal{T}}^{\star}_{np})^2} + \widetilde{\mathcal{T}}^{\star}_{np}\sqrt{(\widetilde{\mathcal{T}}^{\diamond}_{sp})^2 + (\widetilde{\mathcal{T}}^{\diamond}_{np})^2}}{\widetilde{\mathcal{T}}^{\diamond}_{sp}\sqrt{(\widetilde{\mathcal{T}}^{\star}_{sp})^2 + (\widetilde{\mathcal{T}}^{\star}_{np})^2} + \widetilde{\mathcal{T}}^{\star}_{sp}\sqrt{(\widetilde{\mathcal{T}}^{\diamond}_{sp})^2 + (\widetilde{\mathcal{T}}^{\diamond}_{np})^2}}. \tag{11}$$

We randomly selected five multi-instance apps and two social apps (WeChat and Instagram), ran them on two controlled devices, (i.e., the OPPO A37m and HUAWEI CRR-UL00 smartphones) and computed $\mathcal{R}_b$ based on the obtained time values. Figure 9 shows the distribution of the points, four stars for the system environments and twenty diamonds for the plugin environments. $\widetilde{\mathcal{R}}^{\diamond} = 2.31$ when WeChat ran as a guest app on the OPPO A37m smartphone, and $\widetilde{\mathcal{R}}^{\star} = 1.21$ corresponds to a test of Instagram on the HUAWEI CRR-UL00 smartphone. With Eq. (11), we get $\mathcal{R}_b = 1.63$ and draw the angular bisector in dashed and black. Notice that, the visual indistinctness of the equal division is caused due to more compact Y-axis compared to the X-axis. We use this $\mathcal{R}_b$ for all following experiments.

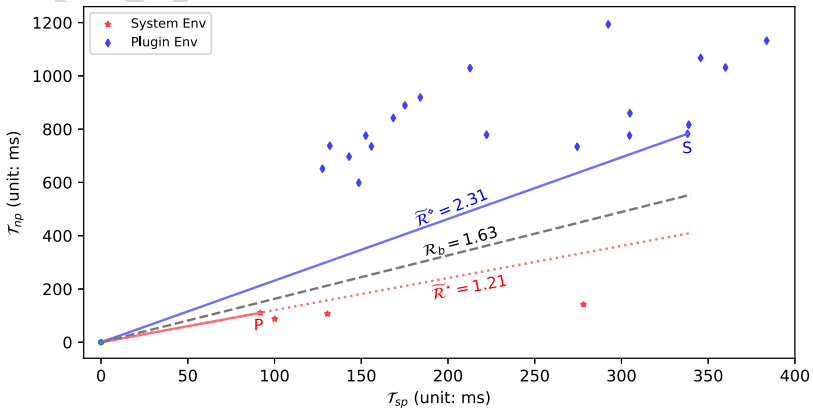

Fig. 9. Sample ratio distribution and threshold determination.

4.4. Detection results and analysis

We conducted experiments for all the social apps and multi-instance apps, on all the three smartphones. We show the results in Table 5. We use the MD5 values of the multi-instance apps to represent the 50 plugin environments and the abbreviations, WX, IN, TW, WA and LI, to denote the five social apps, WeChat, Instagram, Twitter, WhatsApp and LINE, respectively. "$*$" indicates the combination of a social app, a plugin environment and the smartphone is used in the threshold determination phase and the detection of plugin environments succeeds. "$\bigcirc$" means a successful detection as well, with $\mathcal{R}_b = 1.63$. "–", in contrast, corresponds to a failing test.

We unfold the failures as follows:

- The plugin environments #29, #31, #38 and #41 fail to load Twitter as a plugin. Twitter uses an Android mechanism `JobScheduler` to schedule various operations under specified conditions. Android introduced some new features of this mechanism in version 8.0 [20]. However, the four multi-instance apps did not update themselves to handle new features, leading to the Twitter failures.
- The failures for WhatsApp in 12 plugin environments (#4, #12, #16, #26, #28, #29, #33, #37, #38, #39, #45 and #48) are raised because of their faulty handling of the content provider component. Registering an observer for a content provider by invoking `registerContentObserver()` needs extra permissions in Android 8.0 [11]. Unfortunately, these multi-instance apps do not pay attention to the permissions, resulting in a permission denial exception.
- LINE utilizes *dex2oat* to optimize an external JAR file after startup. However, two multi-instance apps (#11 and #39) do not handle the file path properly, resulting in additional failures.

According to above discussions, none of the failures is induced by our modifications in the social apps or remarks the existence of indistinguishable time ratios, i.e., all computed $\mathcal{R}^{\diamond}$ are greater than $\mathcal{R}_b$. In fact, the minimum $\mathcal{R}^{\diamond}$ for plugin environments in all tests is 2.05, when Twitter runs in multi-instance app #9 on HUAWEI CRR-UL00. The maximum $\mathcal{R}^{\star}$ for the system environments is 1.35, introduced by LINE running directly on HUAWEI CRR-UL00. They both show sufficient gaps between the threshold and the real-world time ratios. We suspect the learned threshold 1.63 can be applied to more plugin environments and devices. So far, the research question presented in Section 3.2 is well answered.

Figure 10 shows all the $\mathcal{R}$ values obtained under system environments. As we can see, LINE produces close time ratios on all three devices, even if the actual time values differ a lot. For example, $\mathcal{T}_{sp}$ and $\mathcal{T}_{np}$ collected in LINE on OPPO are 86.4 and 115.5 milliseconds, but on HUAWEI Mate 10 they are 44.9 and 58.9 milliseconds respectively. The other four apps have unstable ratios across devices. Potential causes are the optimizations in the system and framework by certain smartphone vendors, or those done by the app developers for specific smartphones. Noises would also affect the results. Nonetheless, they have little influence on the experiment results.

We also observed that, WhatsApp usually holds relatively small time ratios (typically smaller than 0.6) while the others may produce bigger ratios. Due to this fact and that we determine the threshold based on very few tests on only two controlled devices, we suggest the app developers to exercise their apps, with PluginAssassin augmented, on more available devices and make the decision with an app-specific threshold, instead of using one applicable for many different situations as done in our experiments. For example, WhatsApp may set a smaller threshold (e.g., one) and LINE might requires a bigger value.

Considering the plugin technology has been applied to implement malicious purposes by real-word malwares, such as DualTwitter, we uploaded these 50 multi-instance apps to VirusTotal [40] to check whether they are malicious. Results show that 44 out of 50 are marked as malware or potential malware

Table 5
Detection results for 50 multi-instance apps

ID	Plugin environment	# downloads	OPPO A37m					HUAWEI CRR-UL00					HUAWEI Mate 10				
			WX	IN	TW	WA	LI	WX	IN	TW	WA	LI	WX	IN	TW	WA	LI
1	603698049a810683f7e00030f8f86e10	1M+	∗	∗	○	○	○	∗	∗	○	○	○	○	○	○	○	○
2	18dde4f77aa71b18d61fe604adbb9786	100M+	∗	∗	○	○	○	∗	∗	○	○	○	○	○	○	○	○
3	54bc9bb35425069053ddffebe97d2a97	1M+	∗	∗	○	○	○	∗	∗	○	○	○	○	○	○	○	○
4	42b60ecab5abc4d8d2e73e044faa3182	1M+	∗	∗	○	○	○	∗	∗	○	○	○	○	○	○	○	○
5	3f0901ce1206118c71e2ab9bd888e65e	10K+	∗	∗	○	○	○	∗	∗	○	○	○	○	○	○	○	○
6	7ed21d4ad1938ed62e4f488066398503	50K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
7	15117792ef50d87c00b0746a5c710265	10K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
8	3e9d12add39047a09368a8900a97de0d	10K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
9	5a25da8207d70fbff32071a7b00171be	5M+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
10	933c1758e368c85780d6d0a4f6eec374	100K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
11	8ba71aa17877a8130e60820b6d65f8fb	5M+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	–
12	f430c5635094bbe36f2be5ff737d7658	500K+	○	○	○	○	○	○	○	○	○	○	○	○	○	–	○
13	c5bbdca983e33341d85b747a9d98b539	500K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
14	739477533a8ec019793b58efd6fd0b3d	1M+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
15	a5dbf899b83622bcce24656555e80b38	10M+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
16	9fc8f214ddcd2a6ea3c3aed201e2ece7	100K+	○	○	○	○	○	○	○	○	○	○	○	○	○	–	○
17	f6767aef0f6ab7db3f9aa4810c92d525	100K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
18	13be2b122cf40ba8cc154f05a539fdab	5M+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
19	f7d8a25ef6356729387196cfdee90fea	1M+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
20	f256fb4571b9fae036976f08e002198a	500K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
21	e4d4a4b7ae621440ef5a16bbea43089d	1K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
22	2502dda623aa0b279de256c4c1c0e6a9	1M+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
23	e1873e9cb94c8e51f02d9d2e64d9b4ca	100K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
24	5435db809a2b7fe50c1be6e113963ddd	100K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
25	cd5e6584fc7c6490d03458bca2c80edb	10K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
26	b7deeda1f067bbfd3ab469ca3f316852	50K+	○	○	○	○	○	○	○	○	○	○	○	○	○	–	○
27	d29034a25712b7eb593d0f32c8c447bb	10K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
28	0694e76ebb283ea97a6c7376b8dd281f	1K+	○	○	○	○	○	○	○	○	○	○	○	○	○	–	○
29	479373f0caaad9afd9d91f40dd7b8dd5	50K+	○	○	○	○	○	○	○	○	○	○	○	○	–	–	○
30	5d5c5a33c8db79976142bd4c05c0c17a	45M+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
31	b4cf55b18f78476d5dc0131d1f8775ea	44M+	○	○	○	○	○	○	○	○	○	○	○	○	–	○	○
32	b45395529db076b98b8e046a3ed13565	1M+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
33	550d885e4543ac3c46055f99ad1a6746	150K+	○	○	○	○	○	○	○	○	○	○	○	○	○	–	○
34	5ef57251e3c51120fc6d3fc2fe8cfd4d	960K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
35	0cda386900b7e7c6457340745b5defee	12M+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
36	52e3b419156fea3946365fd367ebe92c	760K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
37	9f0cbe7fd099b78aab2a48d3b3246e41	110K+	○	○	○	○	○	○	○	○	○	○	○	○	○	–	○
38	7a5eefd4e32e4bf204757039c8771ed1	8M+	○	○	○	○	○	○	○	○	○	○	○	○	–	–	○
39	05beaf1e679b76b719b8ef3e0fdfec91	570K+	○	○	○	○	○	○	○	○	○	○	○	○	○	–	–
40	5a43c5efdbdb1cd34f6a05cddeded131	2M+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
41	c0b9a33b04704a5201783a75ec0cd00e	410K+	○	○	○	○	○	○	○	○	○	○	○	○	–	○	○
42	e513fee2700cadb8cac3c1a43960b52a	3M+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
43	a4c0011db865289438886e9cb468f47d	4M+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
44	c350a1e56e3cfea6d9bf557813568381	200K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
45	46260dbf9b058b0314ec2db77b9a16cd	170K+	○	○	○	○	○	○	○	○	○	○	○	○	○	–	○
46	3090e182fcb669a9325c7d2c344e78b8	26K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
47	13c052d371f1d74d873025b4bf3491c0	64K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
48	501495c5d4cf3c031570ccc1f6313f23	770K+	○	○	○	○	○	○	○	○	○	○	○	○	○	–	○
49	b8a985e859930de624d4f4c6a81728d0	21K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
50	9dd70637b8714264234089d676479b30	11K+	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○

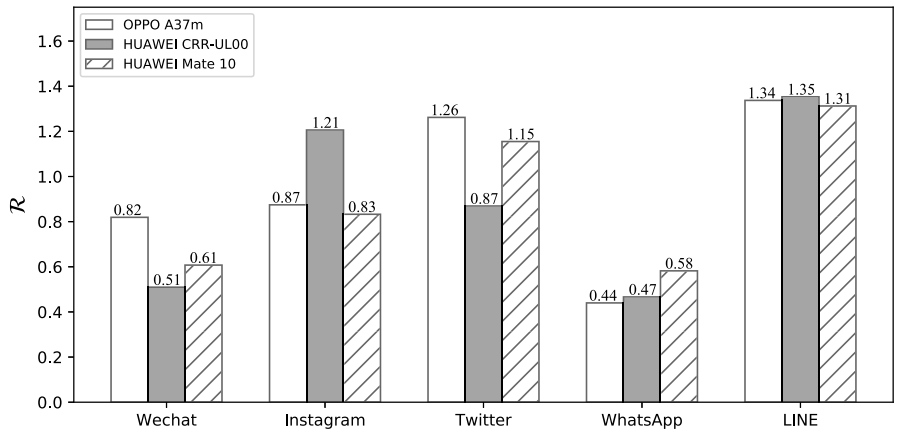

Fig. 10. Time ratios under system environments.

by the antivirus engines hosted by VirusTotal. According to the analysis reports, we found 10 multi-instance apps intend to execute the "*su*" shell command to check whether the device is rooted. Besides, 15 multi-instance apps are reported to use the Android packing technique. Antivirus engines label these samples as potential malware because they may hide malicious payloads and dynamically load them during running to evade analysis.

As we mentioned earlier, multi-instance apps with malicious behaviors may threaten the security of legitimate apps. Fortunately, PluginAssassin has a satisfactory performance on identifying the plugin environments by the time lag contradiction detection approach. From the experimental results, we can faithfully claim that PluginAssassin achieves *zero* false positive rate. For all cases where both the plugin environments and the guest apps do not crash, PluginAssassin doesn't leave any multi-instance app unreported, namely, *zero* false negatives.

4.5. *Runtime overhead*

Since PluginAssassin injects code for extra activity launching, it inevitably introduces some runtime overhead and we will briefly discuss its upper bound in this section. According to our discussion in Section 2 and Section 3, it makes sense for us to simulate the runtime overhead with the observed $\mathcal{T}_{sp}$ and $\mathcal{T}_{np}$ in the system environment. For a single pair of an intra-process launching and an inter-process launching, the overhead $\mathcal{O}$ is bounded as:

$$\mathcal{O} \leqslant 2 \times \mathcal{T}_{sp} + \mathcal{T}_{np}. \tag{12}$$

An app containing N pairs has the overall overhead bounded by $\sum_{1 \leqslant i \leqslant N} \mathcal{O}_i$. In our experiments, $N = 5$. We show all the estimated upper bounds of the overhead in Table 6. From the results, we can see that except the three cases on HUAWEI CRR-UL00, all other tests have the overall overhead lower than two seconds, while on HUAWEI Mate 10, a more powerful device, the overhead for four apps is even lower than one second. Though the running environments and test noise may affect the results, an individual pair of intra- and inter-process launching introduces the overhead of no more than 0.7 seconds on average. In our experiments, we disperse the operations into different activities along the app's running, so we believe the low overhead will not influence the user experience.

Table 6
Estimated upper bounds of runtime overhead (unit: seconds)

Model	WeChat	Instagram	Twitter	WhatsApp	LINE
OPPO A37m	1.84	1.44	1.28	1.72	1.44
HUAWEI CRR-UL00	3.49	1.47	2.06	2.02	1.56
HUAWEI Mate 10	1.24	0.95	0.68	0.94	0.74

5. Limitations and discussion

The proposed mitigation mechanism for ΔT attacks may lose its power on single-core devices, as the probe thread may not have a chance to probe the one that might suffer from a ΔT attack before being terminated or the process scheduling balances the ΔT changes in the obtained time values. Nevertheless, the approach is still possible to detect ΔT attacks successfully in a long-run situation while the probability is lowered on single-core smartphones. We also argue that in nowadays, most, if not all, smartphones are powered with multiple CPU cores, well supporting the defense technique.

Besides the ΔT attack, PluginAssassin also suffers from other carefully devised attacks when the detection principle is public to adversaries. For example, adversaries may develop a plugin environment targeting at a specific app, in which case they have the knowledge of the detection-relevant activities and the operations of measuring the time cost. The app-specific plugin environment can hijack the targeted API calls, forge the timestamps to some extent and thereby bypass the detection. Though in this paper we mainly focus on propose an approach to detecting those general-purposed plugin environments, we deem it not too difficult to integrate some engineering effort to mitigate the app-specific attacks. For instance, app developers can obfuscate the activities and randomize the measurements in different ones along each app update. Besides, app developers are encouraged to collect the time cost of those activities necessary in the app's lifecycle and use the timestamps in other code than only the plugin environment detection. In such cases, forging the timestamps may cause unexpected behaviors and thus be recognized by the app. In addition, verifying the collected values with many other environments from the app users may also screen out the fake ones.

Even though app developers can choose an app-specific threshold, some unforeseen plugin environments may still have a chance not to be identified, or in some cases, a system environment can be mistakenly marked. An adaptive threshold determination algorithm may help in the former situation, using the feedbacks from many other daily use smartphones. For the latter case, the developers can either choose to skip the detection in certain extreme conditions or decide a higher threshold to prune any potential false positives while letting off some plugin environments.

PluginAssassin inevitably increases the overall performance overhead. Nevertheless, as we discussed in Section 4.5, the overhead for most cases is not higher than two seconds in our experiments. App developers can even further reduce the overhead by measuring the time cost of launching those activities necessary in the app's lifecycle, instead of injecting unnecessary but detection-oriented ones as done in our experiments. Moreover, the powerful hardware in nowadays smartphones lessens the potential side-effect, as Table 6 shows, and we believe the approach will not affect the user experience.

App developers need to trade off between the security and the overhead when adopting PluginAssassin in their apps. As mentioned above, developers can reduce the overhead by not injecting additional activities. However, we must be careful that collecting the time cost of existing activities may result in serious privacy leak before PluginAssassin completes the detection. The developers can ease such a situation by performing the detection soon after the app is started. Because the detection finishes before the user interacts with the app, no sensitive data can be leaked to the malicious plugin environments.

6. Related work

To the best of our knowledge, Plugin-Killer [23] is the only existing solution for detecting plugin environments. As we discussed in Section 2.3, it employs some more straightforward thoughts by comparing certain information concerning the app itself and the plugin environment with the counterpart under normal system environments. The effectiveness heavily relies on whether the plugin environments hijack the corresponding APIs used by Plugin-Killer. PluginAssassin, in contrast, fully acknowledges the capabilities of adversaries and leverages the *time* consumed by operations, which are more unpredictable, to perform the detection. Considerations and mitigations of specific countermeasures also make PluginAssassin more robust.

Although the security threats from plugin environments have not been taken seriously by most app developers, the running environment is a primary consideration for security-sensitive applications, especially for malware to conceal itself and evade analysis. Dynamic analysis systems [9,15,32,42,44] observe the app behaviors, obtain more useful information at runtime and deduct the nature of the app samples. However, an intelligent malware may redact its behaviors to evade such detection if it recognizes itself under analysis, making the analysis systems fail to catch any suspicious behaviors. For example, several tools [29,34,36] are deployed in virtualized environments, such as the QEMU emulators, but such virtualized environments can be differentiated from real devices by various anti-analysis techniques. Timothy *et al.* [38] showed the differences in several aspects between emulators and real devices. For instance, the Android APIs obtaining system artifacts, such as the system build property or the phone data, return specific values under virtualized environments. The network configuration is also an indicator to distinguish emulators and real devices. Besides, the CPU and graphical performance can efficiently identify malware sandboxes. Petsas *et al.* [25] summarized the anti-analysis heuristics into three categories, static, dynamic, and hypervisor-related. Static heuristics utilize the aforementioned features of emulators. Dynamic solutions observe the sensor outputs as emulators generate same values for the sensors at same time intervals. Hypervisor-related heuristics characterize the QEMU scheduling and executions and conduct detections based on the results. While most anti-analysis methods are discovered from vast malicious apps, Jing *et al.* [19] proposed a framework, Morpheus, to automatically generate heuristic rules from observable artifacts for apps. More than 10 000 rules generated by Morpheus can be used to detect emulators. Some analysis systems do not depend on emulator environments, but they can also be recognized from normal usage scenarios. Dial *et al.* [12] pointed out the high frequency of user events, such as touchscreen tapping or swiping, gives the app an insight into dynamic testing tools (e.g., the Monkey). Moreover, an app can create a special UI widget, which is invisible to users but valid to analysis tools. If the trap widget is triggered, it presumably indicates that the app is under an analysis environment. Some malware try to root the devices and of course a number of methods have been proposed to detect whether an app is running in a rooted environment. Sun *et al.* [33] evaluated several rooting detection methods by introspecting hundreds of applications, such as checking installed packages, related files, system properties or outputs of specific shell commands, pointing out that, all the inspected methods are ineffective and can be evaded successfully.

Unlike the analysis tools, plugin environments are designed to run guest apps on real devices and do their best to preserve the guest app's utilities. Even if some plugin environments slightly interferes the graphical user interfaces (e.g., presenting an advertisement), in general, they do not prevent the guest apps running. The behavior characteristics, either benign or malicious, of guest apps are out of their concerns. While some anti-analysis techniques are similar to what Plugin-Killer uses, most of them do not work to detect the existence of plugin environments. PluginAssassin turns out to be a heuristic

approach built upon our observation of the time lag contradiction, attributing the detection to observable and comparable time ratios.

Other researchers have also adopted timing-based methods to enforce untampered code executions or detect virtualized systems. Pioneer [30] computes the time cost of the checksum computation in an external trusted entity and verifies the cost of the same execution in an untrusted platform to detect whether the execution has been manipulated by an adversary. While PluginAssassin can benefit from such ideas to treat all platforms from the app users as potentially trusted and mutually verify the time cost, our approach in this paper computes the time ratio locally and provides good detection and runtime performance. Brengel *et al.* [8] utilized the execution time of instructions to identify hardware-virtualized systems. They proposed a similar thought of the relative time, which calculates a ratio of the time cost of the `cpuid` instruction to that of another instruction, such as `nop`, `add`, or `lea`, and compares it with a predefined threshold. Virtualized environment is detected if the ratio exceeds the threshold. Their method focuses on a very low level, computing the time of CPU instructions, as the virtualized systems may intervene the execution of single instruction. Plugin environments hijack higher level API invocations instead, but we can still get inspired from their idea. Instead of using the x86-special instructions (e.g., `rdstc` used in their approach) or existing functions like `clock_gettime()` in our approach, we may start a thread executing a counter written in assembly to time the activity launching in our future work, in case that the functions are hijacked by the plugin environments. Ho *et al.* [17] devised a method for browsers to detect virtual machines by comparing the time ratio of a target operation (e.g., I/O, thread and graphics operations) to a baseline operation (DOM node writing and memory allocation) with a given threshold. Though similar to PluginAssassin, their solution didn't consider the authenticity of the acquired time value from JavaScript APIs like `getTime()`. A powerful but malicious virtual machine can deceive the upper level system and launches similar ΔT attacks. Chen *et al.* [10] proposed a remote detection method to determine if a host is running in a virtual machine. They calculated an indicator value based on the timestamps and frequency of TCP packets, and compared it with a theoretical value. Huge inconsistency between these two values indicates a remote virtualized environment. Fake TCP timestamps can erase the characteristic of clock skew behavior in VMs. Determining the theoretical value can be tedious as plugin environments are much easier to develop, which leads to many more host apps than VMs. Furthermore, if Android apps communicate with remote services that detect whether the apps are running in plugin environments, much additional cellular network traffic may be consumed, causing extra phone charges to users. The above techniques inspired us to present PluginAssassin, addressing issues specific to Android apps and distinguishing the running environments while maintaining good performance.

7. Conclusion

The popularity of multi-instance apps brings new security threats to legitimate apps, which makes legitimate apps run as plugins and allows the host apps to do evil in a stealthier way. While existing detection method is not robust enough to resist attacks, we present PluginAssassin through a deep understanding and observation of the Android plugin technology. PluginAssassin works on top of the *time lag contradiction* phenomenon of launching Android activities in different scenarios. Elapsed time is collected and a corresponding ratio is calculated. PluginAssassin compares the ratio with a predefined one to judge whether the app is running in a plugin environment. We also discuss a potential ΔT attack which can be easily adopted by adversaries to bypass our detection. A mitigation with probing phases is

then introduced as an enhancement. We deploy PluginAssassin in five popular social apps and evaluate its efficiency on 50 real-world multi-instance apps on three smartphones. Experimental results show that, without affecting an app's normal execution, PluginAssassin distinguishes all the plugin environments accurately.

Acknowledgments

This work is supported in part by National Natural Science Foundation of China (NSFC) under grants U1836209 and 61802413, the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China under grant 19XNLG02. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] Android App Bundle, https://developer.android.com/platform/technology/app-bundle.
[2] Android Binder, https://www.nds.ruhr-uni-bochum.de/media/attachments/files/2011/10/main.pdf.
[3] Android Platform Architecture, https://developer.android.com/guide/platform.
[4] Android Sandbox, https://source.android.com/security/app-sandbox.
[5] ApkTool: A tool for reverse engineering Android apk files, https://ibotpeaches.github.io/Apktool.
[6] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet and F. Mercaldo, Detection of obfuscation techniques in Android applications, in: *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ACM, 2018, p. 57.
[7] Baidu, http://mo.baidu.com.
[8] M. Brengel, M. Backes and C. Rossow, Detecting hardware-assisted virtualization, in: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2016, pp. 207–227. doi:10.1007/978-3-319-40667-1_11.
[9] I. Burguera, U. Zurutuza and S. Nadjm-Tehrani, Crowdroid: Behavior-based malware detection system for Android, in: *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ACM, 2011, pp. 15–26.
[10] X. Chen, J. Andersen, Z.M. Mao, M. Bailey and J. Nazario, Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware, in: *2008 IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN)*, IEEE, 2008, pp. 177–186. doi:10.1109/DSN.2008.4630086.
[11] Content change notifications in Android Oreo, https://developer.android.com/about/versions/oreo/android-8.0-changes#ccn.
[12] W. Diao, X. Liu, Z. Li and K. Zhang, Evading Android runtime analysis through detecting programmed interactions, in: *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ACM, 2016, pp. 159–164. doi:10.1145/2939918.2939926.
[13] DroidPlugin framework, https://github.com/Qihoo360/DroidPlugin.
[14] Y. Duan, M. Zhang, A.V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang and X. Wang, Things you may not know about Android (un) packers: A systematic study based on whole-system emulation, in: *25th Annual Network and Distributed System Security Symposium, NDSS*, 2018, pp. 18–21.
[15] W. Enck, P. Gilbert, S. Han, V. Tendulkar, 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, *ACM Transactions on Computer Systems* **32**(2) (2014), 5. doi:10.1145/2619091.
[16] L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna and F. Maggi, Grab'n run: Secure and practical dynamic code loading for Android applications, in: *Proceedings of the 31st Annual Computer Security Applications Conference*, ACM, 2015, pp. 201–210.
[17] G. Ho, D. Boneh, L. Ballard and N. Provos, Tick tock: Building browser red pills from timing side channels, in: *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, 2014.
[18] Instagram, https://www.instagram.com.
[19] Y. Jing, Z. Zhao, G.-J. Ahn and H. Hu, Morpheus: Automatically generating heuristics to detect Android emulators, in: *Proceedings of the 30th Annual Computer Security Applications Conference*, ACM, 2014, pp. 216–225.

[20] JobScheduler improvements in Android Oreo, https://developer.android.com/about/versions/oreo/android-8.0#jobscheduler.
[21] A. Kovacheva, Efficient code obfuscation for Android, in: *International Conference on Advances in Information Technology*, Springer, 2013, pp. 104–119. doi:10.1007/978-3-319-03783-7_10.
[22] LINE, https://line.me/en.
[23] T. Luo, C. Zheng, Z. Xu and X. Ouyang, Anti-plugin: Don't let your app play as an Android plugin, in: *Proceedings of Blackhat Asia*, 2017.
[24] Parallel Space, https://play.google.com/store/apps/details?id=com.lbe.parallel.intl.
[25] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis and S. Ioannidis, Rage against the virtual machine: Hindering dynamic analysis of Android malware, in: *Proceedings of the Seventh European Workshop on System Security*, ACM, 2014, p. 5.
[26] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel and G. Vigna, Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications, in: *NDSS '14*, 2014, pp. 23–26.
[27] Z. Qu, S. Alam, Y. Chen, X. Zhou, W. Hong and R. Riley, DyDroid: Measuring dynamic code loading and its security implications in Android applications, in: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 2017, pp. 415–426. doi:10.1109/DSN.2017.14.
[28] Restrictions on non-SDK interfaces, https://developer.android.com/distribute/best-practices/develop/restrictions-non-sdk-interfaces.
[29] SandDroid, http://sanddroid.xjtu.edu.cn.
[30] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn and P. Khosla, Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems, in: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP '05*, ACM, 2005, pp. 1–16. ISBN 1-59593-079-5. doi:10.1145/1095810.1095812.
[31] H. Shahriar, T. Klintic and V. Clincy, Mobile phishing attacks and mitigation techniques, *Journal of Information Security* **6**(3) (2015), 206–212. doi:10.4236/jis.2015.63021.
[32] M. Sun, T. Wei and J. Lui, Taintart: A practical multi-level information-flow tracking system for Android runtime, in: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 331–342.
[33] S.-T. Sun, A. Cuadros and K. Beznosov, Android rooting: Methods, detection, and evasion, in: *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, ACM, 2015, pp. 3–14.
[34] K. Tam, S.J. Khan, A. Fattori and L. Cavallaro, CopperDroid: Automatic reconstruction of Android malware behaviors, in: *NDSS*, 2015.
[35] Threat Intelligence Team, Malware posing as dual instance app steals users' Twitter credentials, https://blog.avast.com/malware-posing-as-dual-instance-app-steals-users-twitter-credentials.
[36] TraceDroid, http://tracedroid.few.vu.nl.
[37] Twitter, https://twitter.com.
[38] T. Vidas and N. Christin, Evading Android runtime analysis via sandbox detection, in: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ACM, 2014, pp. 447–458.
[39] VirtualApp framework, https://github.com/asLody/VirtualApp.
[40] VirusTotal, https://www.virustotal.com.
[41] WeChat, https://weixin.qq.com.
[42] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen and C. Platzer, Andrubis: Android malware under the magnifying glass, Technical Report TR-ISECLAB-0414-001, Vienna University of Technology, 2014.
[43] WhatsApp, https://www.whatsapp.com.
[44] L.K. Yan and H. Yin, DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis, in: *Presented as Part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 569–584.
[45] L. Zhang, Z. Yang, Y. He, M. Li, S. Yang, M. Yang, Y. Zhang and Z. Qian, App in the middle: Demystify application virtualization in Android and its security threats, *Proceedings of the ACM on Measurement and Analysis of Computing Systems* **3**(1) (2019), 17.
[46] Y. Zhang, X. Luo and H. Yin, DexHunter: Toward extracting hidden code from packed Android applications, in: *European Symposium on Research in Computer Security*, Springer, 2015, pp. 293–311.
[47] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo and F. Massacci, StaDynA: Addressing the problem of dynamic code updates in the security analysis of Android applications, in: *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, ACM, 2015, pp. 37–48.