How Your Phone Camera Can Be Used to Stealthily Spy on You: Transplantation Attacks against Android Camera Service

Zhongwen Zhang [1 3], Peng Liu [2], Ji Xiang [1], Jiwu Jing [1], Lingguang Lei [1]
[1]Institute of Information Engineering, CAS, Beijing, China
[2]Pennsylvania State University, University Park, PA, US
[3]University of Chinese Academy of Sciences, Beijing, China
[1]zwzhang@lois.cn [2]pliu@ist.psu.edu [1](jixiang, jing, lglei)@lois.cn

ABSTRACT

Based on the observations that spy-on-user attacks by calling Android APIs will be detected out by Android API auditing, we studied the possibility of a "transplantation attack", through which a malicious app can take privacy-harming pictures to spy on users without the Android API auditing being aware of it. Usually, to take a picture, apps need to call APIs of Android Camera Service which runs in *mediaserver* process. Transplantation attack is to transplant the picture taking code from *mediaserver* process to a malicious app process, and the malicious app can call this code to take a picture in its own address space without any IPC. As a result, the API auditing can be evaded. Our experiments confirm that transplantation attack indeed exists. Also, the transplantation attack makes the spy-on-user attack much more stealthy. The evaluation result shows that nearly a half of 69 smartphones (manufactured by 8 vendors) tested let the transplantation attack discovered by us succeed. Moreover, the attack can evade 7 Antivirus detectors, and Android Device Administration which is a set of APIs that can be used to carry out mobile device management in enterprise environments. The transplantation attack inspires us to uncover a subtle design/implementation deficiency of the Android security.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; K.4 [**Computers and Society**]: Privacy

Keywords

Android, Spy on Users, Transportation Attack, Android Camera Service

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CODASPY'15, March 2–4, 2015, San Antonio, Texas, USA.
Copyright © 2015 ACM 978-1-4503-3191-3/15/03 ...$15.00.
http://dx.doi.org/10.1145/2699026.2699103.

1. INTRODUCTION

In May, 2014, a news feature story from the USAToday Newspaper reported that users' phone camera can be used to spy on them [11]. "One segment was particularly troubling. In it, Snowden described how a hacker could potentially hijack the camera in William's pre-paid smartphone and use it to capture photos, video, and audio without his knowledge." With this news, that the camera can be used to spy on users becomes well known.

Regarding how the Camera device in a smartphone can be hijacked, a typical way is as follows.

Motivation example. To achieve stealthiness, the attacker can use a QR-code scanner app to spy on users. QR-code scanner can be used in many places, such as comparing price when shopping, downloading a coupon or an app, getting other users' business card. Because the Camera device is used when scanning, the scanner app should have the CAMERA permission. To misuse this app to spy on users, the attacker can maliciously repackage this app to take pictures. According to [36], 86% malware are repackaged version of legitimate apps.

Motivation. This work is motivated by a key observation in launching spy-on-user attacks, which is that getting the CAMERA permission does not address all the concerns of the attacker. Even with the CAMERA permission, the attack is still concern with what the defences can do. Because the repackaged scanner app has to use Android APIs to take a picture, the Framework code of Android, e.g., PackageManager Service, Camera Service, can be extended to do at least 2 things. First, it can do API auditing. API auditing will give users substantial awareness and alerting. For example, one day, users only use the QR-code scanner during 5pm to 6 pm to compare price when shopping, but the audit record shows the camera usage APIs are called every 30 minutes all day. That is a big alerting, and users will gain full awareness of being spied by attackers. Second, a simple image processor can be added (to Camera Service) to verify whether the picture just token is QR-code. If not, the Framework code (Camera Service) can reject returning the picture back. This will fail the attack.

To avoid the second defence, the attacker could use a standard picture-taking app instead of QR-code scanner app to add spy-on-user code. However, even if using a standard picture-taking app cannot confront the first defence.

Our goal is to enable the spy-on-user attack to achieve superb stealthiness. By superb stealthiness, we mean that neither the first nor the second defence way will deter the spy-on-user attack. Through the proposed *transplantation attack*, the attacker can spy on users without calling any Android API.

On Android phones, to capture a picture, attackers could call picture taking APIs provided by Android SDK. However, from the attackers' point of view, this kind of spy-on-user attack is not stealthy. It cannot evade the Android API auditing detection, which records when and which API is called.

Android API auditing is important to both enterprise environments and individual users. In enterprise environments, employees can fulfill their job responsibilities on their Android phones anywhere. Even with permission controls, abuse (by attackers) or misuse (by employees) of apps can still bring security risk to companies. Therefore, as important as Windows/Linux audit in PC world, it is critical to deploy API auditing on employee phones. No matter it is a COPE (Corporate Owned, Personally Enable) phone or a BYOD (Bring You Own Device) phone, the company must deploy Android API auditing before the phone can be used to do any business.

Besides enterprise environments, individual users face security risk as well. The news feature story indicates that users' smartphone camera could be turned on without their knowledge. Such spy-on-user news will make individual smartphone users increasingly concerned about their privacy. Therefore, more and more individual users should do API auditing on their phones.

Android API auditing helps defend the spy-on-user attack in at least three ways. First, it offers awareness to users. Company administrators and individual users can look into the audit files to check if there are camera-access APIs been called. Second, it can do intrusion detection. The API auditing can automatically do intrusion detection according to some (user-provided) detection rules. Third, based on the intrusion detection results, company administrators and individual users can uninstall the malicious apps.

Moreover, either employee phones or individual phones may contain Antivirus (AVs). One of the functionalities of AVs is antispyware, which is used to protect the user's phone from being turned into a voice or video recorder or to see what the phone's camera sees by malicious code. Regularly calling the picture taking APIs will face the threat of being detected by AVs.

In enterprise environments, besides API auditing and AVs, the MDM (Mobile Device Management) is another safety measure, which can control the usage of picture taking APIs. MDM apps can enable or disable Camera Service. Once the Camera Service is disabled, the picture taking APIs cannot work. As a result, the spy-on-user goal cannot be achieved, either.

Problem Statement. Security measures like API auditing, AVs, MDM hinder the existing spy-on-user attacks which relying on picture taking APIs. Motivated by this key observation, in this paper, we focus on a new spy-on-user attack, i.e. how to take pictures without calling any picture taking APIs.

Besides API auditing, log auditing is another way to detect abnormal behavior. However, on Android system, picture taking logs are only generated after picture taking APIs are called. As no picture taking APIs will be called in the new attack, picture taking logs will not be generated. Therefore, this new attack can also evade log auditing.

We have the following insights about this attack. (1) When an app takes a picture, the app will send a request to the *mediaserver* process, in which the Camera Service runs, via Binder IPC (Inter-process Communication). Then, the *mediaserver* process sends a request to the *systemserver* process, in which API auditing can be done. (2) The picture taking code of Camera Service run in the *mediaserver* process, and they exist in the form of *.so* libraries. (3) The camera driver could be directly accessed by the *.so* libraries without the app having any Binder IPC.

Base on the above insights, we could transplant the needed *.so* libraries from *mediaserver* process to a malicious app process, and let the malicious app access camera drivers to take pictures, directly. We denote this attack as **transplantation attack**. To make this happen, the malicious app should be able to access camera drivers. If a camera driver is globally accessible (i.e. has a Linux permission of 666 or 777), the malicious app can directly access it. Otherwise, the malicious app should become a member of *camera* group, which can be achieved by applying CAMERA permission [1]. In this case, the transplantation attack needs CAMERA permission as well. Our survey reveals that among the top 758 apps gotten from Google Play, 24.27% of them request the CAMERA permission. These apps provide a lot of chances to make transplantation attack become a real-world threat (e.g., by repackaging).

Going through a series of failures, we have finally constructed a novel spy-on-user attack. Accordingly, this spy-on-user attack has the following characteristics: 1) A malicious app can take a (potentially privacy-harming) picture at anytime without calling any picture taking APIs. 2) No Binder IPC is involved when the malicious app is taking pictures. 3) The transplantation attack can evade the Android API auditing. 4) The transplantation attack, when being applied to the Camera Service, results in stealthiest and *unnoticed* picture taking. That is, privacy-harming pictures can be taken by the malware without the Android API auditing being aware of it.

Research Contributions. The main contributions of this paper are listed as follows.

- To the best of our knowledge, this work is the first one on the transplantation attack. We have searched the CVE (Common Vulnerabilities and Exposures) list [2]. Among the 448 CVE entries that match the keyword *Android*, we found there was no such kind of attack happened before.

- We have conducted a set of real world experiments on 69 phones. 46.38% of the smartphones (manufactured by 8 vendors) tested by us let the transplantation attack succeed. We have also evaluated the *evasiveness* of the transplantation attack against 7 Antivirus detectors, and Android Device Administration (mainly used to do MDM). The evaluation results show that the transplantation attack can evade all of these defenses.

[1] That is because, Android will automatically put an app with CAMERA permission to *camera* group, which will be discussed in Section 4.1

- Transplantation attack indicates a design deficiency of the Android security. Android will automatically put an app with CAMERA permission into *camera* group, which is the primary reason that transplantation attack happens. However, our experiment shows that doing this is not necessary. Therefore, we believe putting an app into *camera* group is a design deficiency.

The remaining part of this paper is organized as follows. Section 2 gives attack overview. Section 3 describes the workflow of the Camera Service. Section 4 shows construction of transplantation attack. Section 5 shows the evaluation result. Section 6 shows the discussion. Section 7 shows related works. Section 8 shows our conclusion.

2. ATTACK OVERVIEW

In real world, camera drivers on most phones cannot be globally accessed, instead, they are assigned with a Linux permission of 660. Therefore, in most cases, the transplantation attack needs to apply CAMERA permission (to be put in the *camera* group).

To initiate transplantation attack, attackers can simply write an app with CAMERA permission. However, attacking through this app has limited coverage if this app is not downloaded by many phones. To maximize the coverage, re-packaging a wildly downloaded app is an effective way. According to [34], 5% to 13% apps in the third party markets are repackaged. According to [36], 86% malware are repackaged versions of legitimate apps. Therefore, it is quite likely that innocent users will download a repackaged app.

Our survey reveals that among the top 758 apps gotten from Google Play, 24.27% of them request the CAMERA permission. Stowaway [13] reveals that among the 940 apps getting from Google Play, 6% of them require the CAMERA permission but not use it. We also collect 19 phones from our labmates, an average of 35.3% apps on each phone have the CAMERA permission. These apps either in the wild or is widely distributed on users' phones give attackers a lot of opportunities to exploit these apps to spy on users.

We assume attackers would repackage an app that already has CAMERA permission. Therefore, our attack goal has nothing to do with obtaining the CAMERA permission. The malicious repackaging can work as flows.

First, the repackaging does not need to modify **any** existing functionalities, including functionalities using the Camera Service to take pictures, or to scan QR-codes. Second, attackers only need to add the malicious code to spy on users. In Section 4, we will tell how to construct the malicious code. Third, after repackaging is done, attackers should resign the app using *jarsigner*. At last, attackers can submit the repackaged app(s) to, e.g., third party markets to distribute it. Of course, the repackaged app can be submitted to any markets. Even the official Android Market has 1% of repackaged apps [33].

When repackaged apps are running, users' experience is exactly the same as before. User-initiated camera usage will be audited by the API auditing and appears in the audit. However, the attack goal is that the spy-on-user part of accessing the camera hardware will not appear in the audit, which is a stealthy way. Exploiting this stealthy way of accessing the camera hardware, attackers can spy on users without being detected.

3. WORKING MECHANISM OF CAMERA SERVICE

Camera Service provides the functions of using the camera device to, e.g., take a picture, record a video, etc. The workflow of Camera Service is quite different between 2.x and 4.x. In this section, we will analyze the working mechanism of the Camera Service based on Android 4.x, which according to Google's survey [16] becomes the main stream Android version nowadays.

3.1 Camera Service Workflow

Android system is running on top of a Linux kernel. The Android system is essentially a set of processes (address spaces), including daemon processes, the *mediaserver* process, the *systemserver* process, the *servicemanager* process, and the Android application processes.

3.1.1 Overview.

Camera Service belongs to *mediaserver* process, which is a native process. The Camera Service's workflow is shown in Figure 1. The workflow is illustrated using a client process and the *mediaserver* process.

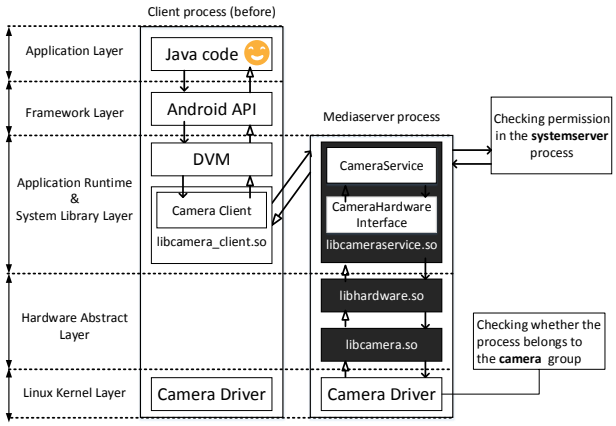

Figure 1: The workflow of the Camera Service

The client represents any app using the Camera Service. The client process has a standard 5-layer structure, which includes Application layer, Framework layer, Android Application Runtime layer (Runtime layer), Hardware Abstract Layer (HAL), and Linux Kernel layer. The Application layer and the Framework layer are written by Java code. The Runtime layer contains a Dalvik Virtual Machine (DVM) to execute Java code. This layer also includes certain native libraries to take care of the IPC needs, e.g. *libcamera_client.so*. In the HAL layer, since the client process does not need to directly interact with Camera device, this layer does not contain any code talking with camera driver. Regarding the Linux kernel layer, since the memory pages containing the kernel code are shared by every process, the client address space also contains the camera driver code.

The *mediaserver* process is a native process. Native processes do not contain Java code, therefore the *mediaserver* process does not need the DVM. The *mediaserver* process only has 3 layers (see Figure 1). The top layer only contains system libraries written in native code (*.so* libraries), therefore, we call this layer as System Library layer. The

System Library layer *.so* libraries are used to handle the request coming from a client, forward the request to the HAL layer, get the response from the HAL layer, and forward the response to the client. The HAL layer also contains *.so* libraries. Different from the System Library layer, libraries in this layer are used to talk with the camera driver. The HAL layer is doing most of the work assigned to the *mediaserver* process. The *mediaserver* process's Linux kernel layer also contains the camera diver.

3.1.2 Request Sending Workflow.

To communicate with the Camera Service, the client should firstly query the *servicemanager* process for the Camera Service's reference through binder IPC. The *servicemanager* is a native process managing a list of registered system services and their references. Querying *servicemanager* for the Camera Service's reference is a quite common IPC transaction, therefore it is not shown in the figure.

After the reference of the Camera Service is obtained, the general workflow of the Camera Service can be described as follows. If the client wants to, for example, take a picture, it should get connected with the Camera Service first. After the connection is established, the client could use all functions provided by the Camera Service. To get connected, the client first calls an Android API from its Java code. The Android API, through a system library (*libcamera_client.so*), sends the *CONNECT* binder request to the *mediaserver* process.

When the binder request is received by *mediaserver* process, the *CameraService* part (see Figure 1) of the System Library layer will parse the request type from binder data structure. If the request type is *CONNECT*, the *CameraService* part will let the *systemserver* process check the client's permission first. This step can be used to do API auditing, which will be discussed in Section 3.2. Only after the client passes permission check, the *CameraService* part will call the *CameraHardwareInterface* part of the System Library layer to initialize the camera device. The functions in the *CameraHardwareInterface* part will call the functions in the HAL layer to interact with the camera driver in the Linux kernel.

3.1.3 Image Data Transfer Workflow.

After a picture is taken, the picture (a frame of image data) will be transferred back (see the $-\triangleright$ flow in Figure 1).

In the HAL layer, there is a *notification* thread that keeps listening on the camera driver and waits for camera events, such as Camera has focused, or focus has moved. When the camera device finishes taking a picture, the camera driver will send an event to the *notification* thread. After the *notification* thread receives the event, it will transfer the image data back to the System Library layer of the *mediaserver* process from the bottom up by calling the callback functions. The image data has already been compressed (by the camera driver) to a certain picture format (e.g. *jpeg*); the *CameraService* part could directly forward the image data to the client process via Binder IPC.

In the client process, after the image data is received by the *CameraClient* in the Runtime layer, it will be forwarded to the Framework layer. Then, the Framework layer posts the image data to the screen and the user will see. The image data could be saved as a picture file in the Application layer, which could be done by developers.

3.2 Android API auditing

Android provides various APIs to access phone hardware (e.g., camera), Wifi and networks, user data, and phone settings. Some of those APIs are protected by permissions. These APIs are implemented either in framework code or in system libraries. When a protected API is called, the implementation code of the API will send a request to the *systemserver* process to check the caller's permission.

The request is sent to the *PackageManager* thread of the *systemserver* process via Binder IPC, particularly. This thread can know when and which API is called, and can get the caller's UID (user ID) and PID (process ID). Therefore, through this thread, Android API auditing can be easily done. Through the audit record, intrusion detection can be done as well.

In Camera Service, when a picture-taking request is received, the *mediaserver* process will send a request to the *PackageManager* thread to check the caller's permission. In the meanwhile, the picture taking API can be audited.

4. CONSTRUCTION OF THE TRANSPLANTATION ATTACK

In this work, our goal is to take a picture without being audited by the API auditing, which should not have IPC with the *mediaserver* process. We assume an attacker would use a repackaged app with CAMERA permission to start transplantation attack. The repackaging should add the malicious code described in this Section.

4.1 First Idea

The first idea of transplantation is to transplant the code both in the System Library layer and the HAL layer from the *mediaserver*'s address space to the malicious app's address space directly. The code in the two layers exist in the form of *.so* libraries. It should be noticed that the camera-related *.so* libraries in the *mediaserver* process do not exist in any normal Android app's address space in the real world.

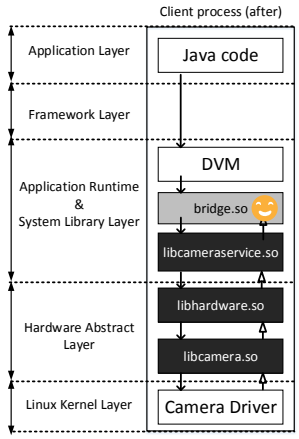

Figure 2: The address space of the malicious app

If the *.so* libraries in the *mediaserver* process can be successfully loaded into the malicious app's address space, the result will look like the one shown in Figure 2 without the *bridge.so* part. The Java code in the Application layer could get through the Framework layer and access the transplanted *.so* libraries in the Runtime layer. Then functions in the Runtime layer call the functions in the HAL layer to talk

with the camera driver in the Linux kernel layer. The execution flow of taking a picture goes from top to bottom within the malicious app's address space. The image data is transferred back from the bottom up inside the app's address space as well.

To make the above workflow work, three steps are needed. The first step is to get the capability of accessing the camera driver in the Linux kernel layer. Camera is a device file, who is assigned to the *camera* group. To get the access capability, the malicious app could become a member of *camera* group. In Android system, apps granted the CAMERA permission will be automatically put into the *camera* group [2]. Therefore, attackers should choose an app with CAMERA permission to repackage, then the repackaged malicious app will be put into *camera* group and can access camera drivers. On the condition that vendors have mistakenly configured the file access permission of camera driver as *rw-rw-rw-* (666) [35], camera driver can be globally accessed. In this case, the malicious app does not need to become a member of *camera* group. That is to say, the malicious app does not need to apply the CAMERA permission. We will discuss this case in the related work section.

The second step is to load the *.so* libraries into the malicious app's address space. There are two ways to load the system *.so* libraries into a process's address space in Android. The first one is loading all the required *.so* libraries when the process is created. This way needs the header files of the *.so* libraries. The other one is loading the required *.so* libraries when they are accessed. Using this way should call the *dlopen* and *dlsym* function, which are special functions provided by Android NDK. We have verified that both of the two ways are workable in loading *.so* libraries.

The third step is to enable the Java code in the Application layer to call the transplanted *.so* libraries in the Runtime and HAL layer. We know that Android provides Java Native Interface (JNI) for Java code to **inter-operate** with native code through DVM. Taking advantage of JNI programming, we can use native code as a bridge to connect the Java code and the transplanted *.so* libraries. The native code will be compiled to a *.so* file, and we name the *.so* file as *bridge.so*, which is shown in Figure 2.

The first and second step are easy to achieve, however, the third step is complicated. We meet several challenges in constructing the *bridge.so*: choosing a reasonable start point, setting up the execution context, and obtaining image data and save it as a picture file, which will be talked in the following sections.

4.2 Choose the Start Point of Transplantation

In step three, when we construct the transplantation attack and when we write the native code, a primary thing is to determine the start point of transplantation, i.e., which function of the transplanted *.so* library to be first called.

In the Runtime layer, the *.so* library we transplanted from the *mediaserver* process is the *libcameraservice.so*. We observed that after API auditing is finished, the first function of the *libcameraservice.so* library executed is the *connect* function. So, we try to start transplantation at this function. Before calling this function, the *bridge.so* also need to first call several *constructor code* to generate some variables needed by the *connect* function.

Till now, the native code contains the *constructor code* and the *connect* function. Then, we try to compile the native code. Unfortunately, the compilation fails. That is because some parameters, global variables, as well as *constructor code* used in the *connect* function are implemented as **private** in Object Oriented Programming. They cannot be used outside their definition class. Therefore, starting at the *connect* function does not work, and we should look for another start point.

If we move the start point upstream of the workflow, the malicious app will cannot evade API auditing, which fails our goal. In addition, we noticed that functions really working are at the downstream of the workflow and these functions are not private. Therefore, moving the start point downstream of the workflow is an option.

We find that, logically, the *libcameraservice.so* library can be broken into two parts: the *CameraService* part and the *CameraHardwareInterface* part. The primary duty of the *CameraService* part is handling Binder request; and it is the *CameraHardwareInterface* part really processing the request by calling functions in the HAL layer. The spy-on-user goal is to take pictures without using any IPC, so, we just need to jump over the *CameraService* part and start at the *CameraHardwareInterface* part, instead.

Although starting at the HAL layer also can call the camera accessing functions, it will encounter more incompatibility problems than starting at the *CameraHardwareInterface* part. That is because, the HAL layer is more closely linked with the driver than the *CameraHardwareInterface* part. According to [35], making hardware work is the primary reason for vendors to customize Android system. Therefore, the closer to the driver, the possible the code will be modified. Another reason to start at the *CameraHardwareInterface* part is that the number of functions in this part is much less than that of the HAL layer.

Starting at the *CameraHardwareInterface* part does not change the address space shown in Figure 2. We still need to transplant the *libcameraservice.so* into the malicious app's Runtime layer. The difference is that the *CameraService* part is no longer executed in the malicious app's address space.

4.3 Set up the Execution Context of Picture Taking

Without proper execution context, the function of taking a picture cannot be executed. *CameraService* part plays a significant role in setting up the execution context needed by the functions in the *CameraHardwareInterface* part. That the *CameraService* part will not be executed will lead to missing of execution context. As a consequence, functions in the *CameraHardwareInterface* part cannot be executed.

To take a picture, the *bridge.so* at least needs to call the *initialize* function (to initialize camera driver) and the *takePicture* function (to take a picture). The execution context needed by the *initialize* function is a data structure *hw_module_t*, which contains a set of camera metadata and a set of function pointers pointing to the functions in the HAL layer. To get this data structure, we need to call the *hw_get_module* function of the *libhardware.so*. After successfully obtain the data structure, the *bridge.so* can call the *initialize* function to initialize camera driver.

[2] Android has bound several permissions to some group IDs in a metadata file named *platform.xml*. In the file, CAMERA permission has been bound to camera group

Although the camera device is initialized, the execution context needed by the *takePicture* function is still missing. We notice that before taking a picture, it should start a preview first. The execution context of preview should be gotten from a data structure transferred by binder. Since there is no IPC in the malicious app's workflow, there is no binder data to use.

The execution context needed by preview is a native window variable, which is created by using a Java parameter obtained from a binder data structure. The Java parameter (*Surface* object) is created in the Application layer of the workflow shown in Figure 1. Noticing this, we could create the Java parameter using Java code and let the native code get the Java parameter by programming with JNI.

After the native window variable is created, we first start a preview, and try to take a picture two seconds later. As expected, the preview is successfully shown on the screen. And, two seconds later, the preview window is successfully frozen, which means the *takePicture* function is successfully executed.

4.4 Store Image Data as Picture File

Although picture/image data can be shown on the screen, it does not mean the picture files have been generated. We need to get the picture/image data and store the data as a picture file.

As we mentioned in Section 3.1.3, the image data is transferred through callback functions. The *CameraHardwareInterface* part also provides the *setCallbacks* function to set up callback functions. There are three callback functions: *notifyCallback, dataCallback*, and *dataCallbackTimestamp*. Their usage can be inferred from the function's name. The *notifyCallback* function tells user such things as the Camera has focused or the focus has moved. In this attack, we do not want the user to receive those notifications. Instead, we want to stealthily take a picture and send the picture to someone else. Therefore, we focus on getting the image data, which needs to implement the *dataCallback* function.

After the *dataCallback* function being added to the *bridge.so*, we rerun the malicious app. Out of our expectation, the *dataCallback* function is not called. As a result, the picture cannot be obtained.

The possible reasons are analyzed as below. First, Android may restrict that the image data only can be transferred to the *mediaserver* process only. However, since we already succeed in previewing the image data on the screen, the malicious app **can** receive the image data coming from camera driver. Therefore, this reason is not valid.

Second, the image data may exist in the kernel space since the camera driver runs in the Linux kernel, and native code running in the user space cannot get the data. However, all transplanted code is also executed in the user space of the *mediaserver* process. Since the image data can be gotten in the *mediaserver*'s user space, it also should be gotten from malicious app's process. This reason is not valid, either.

Third, there may be other missing steps in our transplantation. In the *mediaserver* process, the callback functions are set in the *CameraService* part, whose code is 3.67 times as much as the code of the *CameraHardwareInterface* part[3]. The *CameraService* part is not executed in the malicious app's address space, which may lead to other execution context not being set up or some functions not being executed.

[3]The comparison is between the *.cpp* files on Android 4.1.2 version.

To see if this reason is valid, we do a dynamical analysis. We take advantage of the *log* mechanism provided by Android to print the names of the functions executed in a picture taking workflow. To see if there are missing steps, we compare the functions executed in the *mediaserver* process and the malicious app process by analyzing the logs. The result shows that several functions in the HAL layer are not called in the malicious app's address space. For example, functions used to detect users' face, to play shutter sound, and to enable/disable an event message type to get/drop a notification are not called.

We care about the functions about notification, whose names are *enableMsgType* and *disableMsgType*. They are used in pairs. There are several message types enabled in the *mediaserver* process when taking a picture. What really matters is the event message type *CAMERA_MSG_COMPRESSED_IMAGE*. Enabling this message type will make the *dataCallback* function be called. Motivated by this special trigger condition, we let the native code call the *enableMsgType* and *disableMsgType* functions provided by the *CameraHardwareInterface* part. In addition, we let the native code call the *stopPreview* function to stop the preview and the *release* function to close the camera device. These steps make the malicious code clean and tidy.

After all the above functions are added into the malicious app's native code, we rerun the app. The good news is the *dataCallback* function is called, however, the image data remains not received. To diagnose, we do a static analysis, and we find the image data is transferred back by the *notification* thread (see Section 3.1). The *notification* thread needs to wait for image data coming from the camera driver, which is time consuming. To the contrary, the native code is executed in another thread, which takes much less time. The two threads are not synchronized. Therefore, before the image data is transferred back, the *CAMERA_MSG_COMPRESSED_IMAGE* message type may be disabled or the camera may be closed.

Based on this observation, we add synchronization for the two threads. This time, we successfully get the image data transferred by the *dataCallback* function in the native code. Moreover, the image data is successfully stored to a picture file in the *dataCallback* function.

4.5 How to Hide

To make the spy-on-user attack be stealthily carried out, we assume the attacker would need the following requirements when executing the malicious code of picture taking.

Hiding the user interface. As an app, the attacker's Java code should be implemented in the form of Android components. Among the four kinds of components (*Activity*, *Service*, *Content Provider*, and *Broadcast Receiver*), we prefer the *Service* component to hold the malicious Java code of creating the Java parameter needed in preview and calling the malicious native code. It is because this component does not contain a user interface and runs in the background. When the *Service* component is running, users will not feel any abnormal things on the screen.

Hiding the preview window. At the meanwhile of hiding user interface, we need a window view (*window* variable) to preview when taking a picture. A view occupies a rectangular area on the screen, which could be seen by the user. We have two ways to hide the preview window. The first one is minimizing the window view's size to such as 1 pixel

× 1 pixel, so that it cannot be seen by humans' eyes. What worth mentioning is that the size of the preview window does not affect the size of the picture. The other way to hide the preview window is not transferring the image data to the window view when previewing. As there is no image data received, the window view cannot be seen on the screen. However, the view still stays on the screen. If coincidentally the malicious app is taking a picture and users touch the window view area, e.g., to launch a new app, there will be no response happens, which will make users suspicious. Therefore, we prefer the first way to hide the preview window.

Hiding the flashlight and shutter sound. The flashlight and shutter sound can be controlled by the *bridge.so*. To minimize the picture taking signs, we prefer not to use them. At daytime, the nature light is enough for taking a picture. While at night, the evening light is not suitable for photographs without flashlight. Therefore, the attack should choose not to take pictures after 8pm. Without flashlight and sound, the user can hardly feel the signs of picture taking. It may be considered that a camera indicator light may be shown when taking pictures. However, all the experiment phones used by our labmates (see Table 3) do not show it. Further, provided the camera indicator light is shown, the nature light will make it hardly be noticed.

Stealthily sending the picture out. There are several ways that can be used to send pictures out, such as MMS (multimedia message), Bluetooth, 3G, and Wifi. To achieve the stealth goal, we should not cost the user's money, leave any audit records, or light on any icon on the desktop. Users care about the MMS they send, as these will cost their money, so we should not use them to send pictures. Sending data out through the Bluetooth has the shortcomings that it will leave an audit record and does not fit for long distance transfer, so we do not use it, either. Since 3G and Wifi do not suffer the two shortcomings as Bluetooth, we prefer to use them to send pictures out. To avoid lighting on their icon on the screen, we let the malicious app send pictures out **after** the 3G/Wifi is turned on by the user. Nowadays, 3G traffic is quite cheap and has no toplimit; it is not a big concern of users any more.

Do not drain the battery too quickly. If the malicious app takes pictures too frequently, the battery may be drained too quickly, which will make the user suspicious. To determine the frequency of picture taking, we measured the power consumption of taking ten pictures, playing *Angry bird* and *Sudoku* for ten minutes respectively on the same phone using *PowerTutor*. The result shows that the power consumed by playing 10 minutes of *Angry bird* can support taking 41.7 pictures, while playing 10 minutes of *Sudoku* can support taking 30.2 pictures. So, taking 36 pictures every day will not be perceived by the user. Further, we do not take pictures at night because the flashlight cannot be used, this will also save energy. To make the balance between the battery consumption and the frequency of picture taking, the attacker could set the malicious app to take a picture every 20 minutes during 8am to 8pm (36 pictures in total).

4.6 Other Things That Attackers Should Do

In the real world, the first step to launch the transplantation attack is to choose an app with CAMERA permission to repackage. For example, scan-QR-code apps need CAMERA permission to open camera and scan QR code, attackers can choose this kind of apps to repackage. A great number of scan-QR-code apps can be found on Google Play, Amazon, or other markets. Attackers can easily choose a widely downloaded one on Google Play to repackage, and then attackers could submit the repackaged app to third party markets to distribute. Also, the target app can be chosen on third party markets and submit to Google Play.

Assuming that the malicious repackaged app has been successfully installed into the user's phone, the next step is letting the app obtain the CPU cycles to run itself. Attackers would like that the malicious app is able to get the CPU cycles every 20 minutes. A possible way is letting the malicious app be started immediately after the phone has been booted up. To do this, the malicious app should receive a BOOT_COMPLETED broadcast, which is sent by the Android system when the system has been boot up. The stealthy picture taking functionality can be implemented as a service, which can be denoted as picture taking service. In the broadcast receiver, the attacker put a piece of code that denoted as *timer* and let the *timer* to start the picture taking service every 20 minutes.

Some Antivirus allow users to block apps from receiving the BOOT_COMPLETED broadcast. In this case, the stealthy picture taking can be trigged by motion sensors, such as accelerometer sensor, orientation sensor. There are plenty of open sources towards how to use motion sensors to infer position of smartphones publicly available on many websites. Therefore, attackers can easily get those code and add them into the repackaged app to make the stealthy picture taking triggered by motion sensors. Using motion sensors does not need any permission.

To send pictures out, the malicious app needs to apply the INTERNET permission. According to [19], there are 88% apps with the INTERNET permission. Most of apps use INTERNET permission to fulfill the advertisement needs. It is not a strange thing that a scan-QR-code app to apply this permission, e.g, to show advertisements.

Front and Rear Camera Nowadays, most smartphones shipped with front and rear Cameras. Sometimes, attackers want to see users' expression, and sometimes, attackers want to see the environment where users stay in. This can be archived by specifying the camera ID of front Camera or rear Camera in the *bridge.so*. Usually, the ID of front Camera is 0, and the other one is 1, which is true on the successful ones of 69 phones (See Section 5). Still, there exists a possibility that vendors may change their Camera IDs. In this case, analyzing the phone logs or brute force may be help to find the IDs.

5. EVALUATION

We evaluate the transplantation attack in the aspects of success rate on real phones, detection rate of Antivirus (AVs), and the Android Device Administration, respectively.

5.1 Success Rate

To know the effectiveness of the transplantation attack, we run the malicious app on different phones shipped with different Android versions in the real world. We choose 8 different vendors, 69 different phones, and 7 different Android versions. The result is shown in Table 1 and Table 2.

Among the tested phones, 14 of them are collected from our labmates, 62 of them are achieved from *Baidu* app test platform [1]. 7 of 62 phones have the same model and the

same Android version with some of the 14 ones. So, the total number of **different** phones is 69. On the *Baidu* app test platform, all phones used to test apps are real phones not emulators.

On our labmates' phones, we test all functionalities of the malicious app, including picture taking, preview window hiding, and picture sending. We let the malicious app send the stealthily taken pictures to another phone. If the receiver can successfully open and view the pictures, we regard the attack as successful. As long as the Wifi is available, if the malicious app can take a picture, it can always send the picture out.

On the phones provided by *Baidu* test platform, because we cannot physically access those phones, we only test the picture taking functionality on the test platform phones. For the convenience of test, we let the malicious app show the preview window on the screen. Therefore, to check whether a phone's screenshots (generated by the test platform) contain the preview window or not, we can verify whether the picture taking is successful or not.

As shown in Table 1, for version 4.1.1 phones from 5 vendors, the success rate of transplantation attack is 100%. For version 4.1.2 phones from 5 vendors, the success rate of transplantation attack is 75%. We also measure the success rate per vendor. As shown in Table 2, the success rate for Samsung phones is 52% (25 phones in total). The success rate for Huawei phones is 54.55% (11 phones in total). Overall, among the 69 phones, the overall success rate is 46.38%. This means nearly a half of the phone in real world would possibly suffer from this spy-on-user attack.

Table 1: Success rate of different Android versions

Android Version	Vender	Number	Success Rate
Android 4.0.3	HTC	4	0
	Huawei	3	
	LG	1	
Android 4.0.4	Google	1	12.5%
	Samsung	4	
	HTC	5	
	Huawei	1	
	Sony	1	
	Moto	4	
Android 4.1.1	Google	1	100%
	Samsung	1	
	HTC	4	
	Huawei	2	
	Meizu	2	
Android 4.1.2	Google	2	75%
	Samsung	16	
	HTC	1	
	Huawei	3	
	Sony	2	
Android 4.2.1	Samsung	1	0
Android 4.2.2	Huawei	1	40%
	Sony	3	
	LG	1	
Android 4.3	Samsung	3	0
	HTC	1	
	Huawei	1	
Total	8	69	46.38%

Table 2: Success rate of different vendors

Vendor	Google	Samsung	HTC	Huawei
Success Rate	75%	52%	33.33%	54.55%
Vendor	Sony	Meizu	LG	Moto
Success Rate	33.33%	100%	50%	0

5.2 Look into the phones where the attack is not successful

The experiment on version 4.0.3 is very confusing, so we analyze this version's picture taking workflow, and we find there is no difference between this version and other 4.x versions. All 4.0.3 version of phones are provided by *Baidu*. Since we cannot get the failure info (*adb log*) of phones from *Baidu* test platform, we cannot figure out the failure reason on version 4.0.3 as well as version 4.2.1 phones. However, we believe this is a side benefit of customization according to the failure reasons of the phones used by our labmates, which will be analyzed as follows.

Table 3: Experiment result on our labmates phones

Vender	Model	Version	Result
Google	Nexus S	4.0.4	√
	Nexus S	4.1.2	√
	Nexus 4G	4.1.1	√
	Galaxy Nexus	4.1.2	×
Samsung	SHV-E160S (Galaxy Note)	4.0.4	×
	GT-I8268	4.1.2	√
	GT-I9300 (Galaxy S3)	4.1.2	√
	SCH-I879	4.1.2	×
	SCH-I959 (Galaxy S4)	4.2.2	×
	SCH-N719 (Galaxy NoteII)	4.3	×
HTC	T528t (One ST)	4.0.4	×
Sony	LT29i (Xperia TX)	4.1.2	√
MeiZu	M040 (MX2)	4.1.1	√
Huawei	P6-C00 (Ascend P6)	4.2.2	√

We only analyze the failure reason phones used by our labmates. The experiment result is shown in Table 3.

The transplantation attack does not succeed on 6 phones. The failure reasons are different for each phone. To analyze each phone's failure reason, we obtain their *adb log* of taking a picture generated by the *Camera* app shipped on the test phone and by the malicious app. Here, we summarize our findings.

For Google Galaxy Nexus, we analyze its source code of picture taking workflow since this phone support AOSP (Android Open Source Project). We find that before opening the camera device, the *mediaserver* process will open the device */dev/rproc_user* first, which belongs to the *drmrpc* group. Unlike the *camera* group ID, the *drmrpc* group ID cannot be obtained by apps. Therefore, as apps cannot become a member of the *drmrpc* group, they cannot open the */dev/rproc_user* device. As a result, the attack fails.

For Samsung SHV-E160S, the difference between the *Camera* app and the malicious app is that the malicious app is "unable to find matching camera info" for the given camera ID, but the *Camera* app can. As a result, the malicious app cannot open the camera, and the error number is -1, which means "operation not permitted". As we cannot get the source code of this phone, we could only infer that it may be the vendor has modified the workflow of opening the camera device.

For Samsung SCH-I879, the error message and error number are the same as Samsung SHV-E160S, which still is "operation not permitted". Since Samsung SCH-I879's Android version is the same as Samsung GT-I8268 and Samsung GT-I9300, we compare the adb logs obtained from the three phones. However, as they use different hardware and different internal logic of picture taking, the comparison between the three phones does not help.

For Samsung SCH-I959, the log shows that the camera device is successfully initialized. But taking pictures on this phone needs to write a file (*CameraID.txt*) into the *data* par-

tition. Otherwise, the preview cannot successfully start. As a result, the *takePicture* function fails, whose error number is -38. This error number is not defined by the AOSP, and we believe it is defined by the vendor. Since the malicious app does not have privilege to write the *data* partition, the transplantation attack cannot succeed.

For Samsung SCH-N719, when initializing the camera device, it should open the device */dev/video0*, which belongs to the *camera* group. As the malicious app is already set as a member of the *camera* group. Theoretically, it could access the device. Surprisingly, the experiment fails. To diagnose, we analyze the *adb log* and a metadata file (*ueventd.smdk4x12.rc*), and we find the camera hardware of Samsung SCH-N719 is the same as Samsung GT-I9300 on which phone the attack is successful. The difference between the two phones is that Samsung SCH-N719's Android version is 4.3 and the SEAndroid is enabled. We will discuss the SEAndroid in Section 6.

The last one is HTC-T528t, whose error message is "cannot open OpenMAX registry file /tmp/.omxregister". The failure reason is like the Samsung SCH-I959, and this one is the malicious app does not have privilege to access the registry file. But the error number is different. On HTC-T528t, the error number is 0x80001000, which we believe is defined by the vendor as well.

Among the 6 failed phones, 3 of them (Galaxy Nexus, Samsung SCH-I959, and HTC-T528t) fail because extra device and files are involved in their picture taking workflow. We wonder if the malicious app can access the device and files, could the transplantation attack succeed? To answer this question, we do an experiment on the Galaxy Nexus phone, also because this phone supports AOSP.

We know the failure reason of this phone is that the malicious app cannot be assigned the *drmrpc* group ID. Learning from the binding between the CAMERA permission and *camera* group ID, we make the *drmrpc* group ID available to apps by binding it to a permission, e.g., binding it to the CAMERA permission. Binding them needs to modify a metadata file (*platform.xml*), which needs root privilege. After binding, the malicious app can get the *drmrpc* group ID. Then, we rerun the malicious app, and this time the malicious app succeeds in taking a picture and sending the picture out.

According to this experiment, we can infer that if the malicious app can access to the devices and files, the success rate of transplantation attack will be increased from 57.14% to 78.57% in Table 3.

5.3 Evading Antivirus

Employee phones or individual phones may be installed Antivirus (AVs). To test whether the malicious app could evade detection under AVs monitoring, we choose the top 7 **free** AVs according to the [28]'s comparison result. Among the protections they provide, we care about the Scans Phone Apps protection, the Real-Time Protection (discover harmful threads immediately), the Antispyware protection (protect users from being spied), and the Quarantine Section protection (store the suspicious thread in an isolation area). These protections could be classified into two categories: the installation time protection and the runtime protection.

In order to know whether the malicious app could evade detection at installation time, we first install the 7 AVs into 3 experiment phones, then install the malicious app. The 7 AVs' scanning result claims the app is clean, as a result, the app is successfully installed into the 3 phones. Then we run the app under the 7 AVs' monitoring to test whether the app could evade runtime detection. As expected, none of the 7 AVs detect the malicious app out. The result is shown in Table 4.

The result shows that the malicious app can successfully evade detection at both installation time and runtime. Two possible reasons are as follows. The first one is that the installation time scanning only scans the signature of an app. As the malicious app is new, it does not have signatures, yet. In addition, the AVs do not regard loading external *.so* libraries is malicious. Therefore, when the native code thread is running, the AVs do not regard the running thread as evil, either.

5.4 Evading Enterprise Device Administration

Device Administration is usually enforced in the enterprise, in which the phones are owned by companies or organizations. Android framework provides several special Android APIs, which are called Device Administration APIs. They could be used by device admin apps to configure a phone. Administrators could install the admin app on the employee's phones to, for example, disable or enable the camera device. It is designed that once the camera device is disabled by the Device Administration APIs, apps cannot access the camera device anymore.

To understand whether the transplantation attack could sustain the camera device disabling, we run the *Camera* app and the malicious app after the camera device is disabled, respectively. As expected, the *Camera* app fails in taking a picture after the camera device is disabled. However, the malicious app sustains the disabling and successfully takes a picture.

That is because the *Camera* app relies on the *mediaserver* process to take pictures. When the camera device is disabled, the value of a flag (*sys.secpolicy.camera.disabled*) is set to 1 (true). Once the flag's value is 1, the *mediaserver* process will reject any request of accessing the camera device. What is to say, all picture taking APIs cannot be called. In the transplantation attack, the malicious app does not rely on the *mediaserver* process, does not need to call those APIs, and does not execute the flag checking code in its address space, either. Therefore, the transplantation attack can evade the Device Administration as well.

6. DISCUSSION

6.1 Design Deficiency

The transplantation attack indicates that there is a subtle design/implementation deficiency of Android security management.

Since it is the *mediaserver* process in charge of talking with the camera driver, and since an app does not, we wonder whether it is necessary for Android system to put an app into the *camera* group. To answer this question, we did an experiment by removing the apps with CAMERA permission from the *camera* group. We found that the *Camera* app as well as apps relying on it (by sending an *Intent*) can take pictures even if they are not running with the *camera* group ID. The result shows that binding the CAMERA per-

Table 4: Antivirus Testing Result

Protections \ AVs	Lookout	McAfee	Kaspersky	ESET	Trend	F-Secure	NetQin
Scan Phone Apps	√	√	√	√	√	√	√
Real-Time	√	√	√	√	√	√	√
Antispyware	√	√	√	NA	NA	√	√
Quarantine Section	√	NA	√	√	NA	√	NA

mission with the camera group could be a design deficiency, since this will allow the transplantation attack to succeed.

6.2 How to Defend

There may be several ways to defend the transplantation attack, but some of them may not work out. For example, forbidding the usage of system libraries may sound a good idea to defend the attack. However, as apps can ship their own copy of the required system library, this way may not work out. Here, we discuss two possible ways as follows.

6.2.1 *Break the Binding Between Permissions and Group IDs.*

To start transplantation attack, a malicious app should get the capability of accessing a hardware device. To gain this capability, the malicious app should be assigned with the hardware's group ID by applying corresponding permission. Noticing this, a defence is that we could break the binding between permission strings and group IDs. Taking Camera device as an example, breaking the binding between CAMERA permission and *camera* group ID will not affect the normal apps to take pictures. That is because Camera device has a daemon process (*mediaserver* process, in which Camera Service runs) in charge of taking pictures. Apps just need to send request to the daemon process, and the process will handle the picture taking work.

One weakness of this defence is that when the hardware has zero daemon process (e.g., there is no daemon process for Sdcard) or more than one daemon processes, it is possible to result in denying of services.

6.2.2 *Using SEAndroid Policy.*

SEAndroid enforces mandatory access control to every process (user) under a fine-grained access control policy. Every process belongs to a domain (type). Here, third-party apps are classified into the *untrusted_app* domain, which will be blocked when directly access the camera driver.

Although SEAndroid can block accessing the camera driver, it has a rather limited enforcement range. SEAndroid [27] is merged into AOSP since version 4.3 and enforced since version 4.4. According to Google's survey [16], the phones shipped with version 4.3 and 4.4 each accounts for 8.5% of the total at the beginning of May, 2014. That a phone shipped with 4.3 version of Android does not mean that the SEAndroid is enforced. So, nearly 90% of the Android phones in the wild are however not protected by SEAndroid. Among the phones used by our labmates, 93% of them without SEAndroid. It may take a long period of time before SEAndroid can be widely deployed in the wild. During this period of time, many many users may suffer from the spy-on-user attack.

Besides the distribution range limitation, SEAndroid has weakness as well. Pau Oliva shows 3 weaknesses of SEAndroid and gives out 4 ways to bypass SEAndroid [29]. We did an experiment, in which we change SEAndroid from enforce mode to permissive mode via PC terminals. The same principle could be applied to apps. The experiment shows that SEAndroid can indeed be bypassed.

7. RELATED WORKS

A Similar Attack. Xiaoyong et. al. [35] illustrate an end-to-end attack similar to transplantation attack. Their attack can take a picture without applying CAMERA permission, and can evade API auditing as well. However, their attack has a critical premise that the camera device node (*/dev/video*) should be set as publicly readable and writable (a Linux permission of 666), which means the phone is mistakenly configured. Otherwise, their attack cannot succeed. In case of fulfilling the premise, the transplantation attack can succeed without applying CAMERA permission, too. In addition, transplantation attack can work on nearly a half of well configured phones.

Xiaoyong et. al. have implemented the whole HAL layer within their app, which is quite different from us. We just implemented a *bridge.so*, and let the *bridge.so* call the transplanted *.so* libraries of System Library layer and HAL layer. Our attack and their attack are constructed in very different ways. Obviously, our attack has much less lines of code.

Xiaoyong's work and our work share different focus, too. They mainly focus on understanding security risks in vendors' customization process; and their attack is inspired by a security flaw that vendors have mistakenly configured the camera device node with permission of 666. However, at the beginning, our focus is to construct a transplantation attack to evade API auditing.

Other Spy-on-user Attacks. Besides Camera, many ways can be applied to spy on user. TouchLogger [7], TapLogger [31], TapPrints [20], ACCessory [24], ACComplice [18], Soundcomber [26], and Screenmilker use motion sensor readings, microphone, screenshots, to work as a keylogger to spy on users' input, location hacker to trace users' driving, or sound trojan to infer users' credit card accounts, etc. Stealthy Video Capturer [30] stealthily record video to compromise users' privacy. Besides the side channels on smartphones, physical side channels like shoulder surfing, reflection of the screen from sunglass [25], and oil fingerprint [32] [4] also can be used to spy on users.

Although these works also focus on spying on user, none of their methods is carried out by transplantation attack, and they may be faced the risk of being detected out by API auditing.

Permission Escalation Attacks. Another well studied attack of Android system is permission escalation attack. To detect whether an app has unprotected interfaces that can be exploited to escalate permissions, a number of detection tools have been proposed [8] [12] [13] [3] [15]. These static analysis tools are likely to be incomplete, as they cannot

completely predict the actual permission escalation attack occurring at runtime. To address this issue, some framework extension solutions [14] [10] [6] [17] [5] have been proposed.

Root Exploits Attacks. According to [36], attacks exploiting root privilege play a significant role in compromising Android security. Among the root exploiting malware, the *DroidKungFu* [22] is a typical example. Attacks exploiting root privilege could break the boundary of Android sandbox and could access resources without applying permissions. The root exploits attacks could be blocked by SEAndroid [27]. By introducing SEAndroid, processes even running with root privilege cannot access the protected files and devices.

Framework Enhancements. A large number of solutions [21] [37] [23] [9] focus on enhancing runtime permission control to restrict app's permission at runtime. These solutions aim at providing a fine-grained access control for IPC. However, the transplantation attack does not involve IPC. Therefore, the attack fails these solutions.

8. CONCLUSION

In this paper, we propose the transplantation attack, an attack which enables a malicious app to take (potentially privacy-harming) pictures at anytime without being audited by the Android API auditing. The transplantation attack, when being applied to achieve the spy-on-user goal, results in stealthiest and unnoticed picture taking. We have conducted a set of real world attacking experiments. 46.38% of the 69 smartphones (manufactured by 8 vendors) tested by us let the transplantation attack succeed. The transplantation attack also uncovers a subtle design/implementation deficiency of the Android system. Our recent analysis reveals that although on AOSP version 4.4 the deficiency is fixed, most of vendors have not fix this problem yet.

9. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable comments. This work is supported by Strategy Pilot Project of Chinese Academy of Sciences under Grant XDA06010702; National High Technology Research and Development Program of China (863 Program) under Grant 2013AA01A214, 2012AA013104. Peng Liu is supported by Army Research Office W911NF-09-1-0525, W911NF-13-1-0421; National Science Foundation CCF-1320605, SBE-1422215.

10. REFERENCES

[1] Baidu Mobile Test Center. Available at `http://mtc.baidu.com/`.

[2] Common vulnerabilities and exposures. `http://cve.mitre.org/`.

[3] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *ACM CCS*, 2012.

[4] A. J. Aviv, K. Gibson, E. Mossop, M. Blaze, and J. M. Smith. Smudge attacks on smartphone touch screens. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[5] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XMandroid: a new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*.

[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *19th NDSS*, 2012.

[7] L. Cai and H. Chen. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. In *Proceedings of the 6th USENIX Conference on Hot Topics in Security*, HotSec'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *9th MobiSys*, 2011.

[9] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for android. In *Information Security*. 2011.

[10] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security*, 2011.

[11] T. Donegan. How your phone camera can be used to spy on you. `http://cameras.reviewed.com/features/how-your-smartphone-camera-can-be-used-to-spy-on-you`, 5 2014.

[12] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *16th ACM CCS*, pages 235–245. ACM, 2009.

[13] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *18th ACM CCS*, pages 627–638. ACM, 2011.

[14] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[15] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland*, 2009.

[16] Google. Dashboards. `http://developer.android.com/about/dashboards/index.html?utm_source=ausdroid.net\#Platform`, 2014.03.

[17] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *19th NDSS*, 2012.

[18] J. Han, E. Owusu, L. Nguyen, A. Perrig, and J. Zhang. Accomplice: Location inference using accelerometers on smartphones. In *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on*, pages 1–9, Jan 2012.

[19] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *18th ACM CCS*, 2011.

[20] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury. Tapprints: Your finger taps have fingerprints. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 323–336, New York, NY, USA, 2012. ACM.

[21] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *5th ACM CCS*, 2010.

[22] NC State University. Security alert: New sophisticated android malware droidkungfu found in alternative

chinese app markets. Available at http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html, 2011.

[23] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 2012.

[24] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang. Accessory: Password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12, pages 9:1–9:6, New York, NY, USA, 2012. ACM.

[25] R. Raguram, A. M. W. 0002, D. Goswami, F. Monrose, and J.-M. Frahm. ispy: automatic reconstruction of typed input from compromising reflections. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 527–536. ACM, 2011.

[26] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[27] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS*, 2013.

[28] Toptenreviews. 2014 Best Mobile Security Software Comparisons and Reviews. Available at http://mobile-security-software-review.toptenreviews.com/, 2014.

[29] viaForensics. Defeating SEAndroid ÍC DEFCON 21 Presentation. Available at https://viaforensics.com/mobile-security/implementing-seandroid-defcon-21-presentation.html, 8/3/2013.

[30] N. Xu, F. Zhang, Y. Luo, W. Jia, D. Xuan, and J. Teng. Stealthy video capturer: A new video-based spyware in 3g smartphones. In *Proceedings of the Second ACM Conference on Wireless Network Security*, WiSec '09, pages 69–78, New York, NY, USA, 2009. ACM.

[31] Z. Xu, K. Bai, and S. Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 113–124, New York, NY, USA, 2012. ACM.

[32] Y. Zhang, P. Xia, J. Luo, Z. Ling, B. Liu, and X. Fu. Fingerprint attack against touch-enabled devices. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 57–68, New York, NY, USA, 2012. ACM.

[33] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of "piggybacked" mobile applications. In *3rd CODASPY*, 2013.

[34] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *2nd CODASPY*, pages 317–326. ACM, 2012.

[35] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *IEEE Symposium on Security and Privacy*, 2014.

[36] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP)*, pages 95–109. IEEE, 2012.

[37] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing*. 2011.