Screenmilker: How to Milk Your Android Screen for Secrets

Chia-Chi Lin[1], Hongyang Li[1], Xiaoyong Zhou[2], XiaoFeng Wang[2]
[1]Department of Computer Science, University of Illinois at Urbana-Champaign
[2]School of Informatics and Computing, Indiana University at Bloomington
{lin36, hli52}@illinois.edu, {zhou, xw7}@indiana.edu

***Abstract*—With the rapid increase in Android device popularity, the capabilities that the diverse user base demands from Android have significantly exceeded its original design. As a result, people have to seek ways to obtain the permissions not directly offered to ordinary users. A typical way to do that is using the Android Debug Bridge (ADB), a developer tool that has been granted permissions to use critical system resources. Apps adopting this solution have combined tens of millions of downloads on Google Play. However, we found that such ADB-level capabilities are not well guarded by Android. A prominent example we investigated is the apps that perform programmatic screenshots, a much-needed capability Android fails to support. We found that all such apps in the market inadvertently expose this ADB capability to any party with the INTERNET permission on the same device. With this exposure, a malicious app can be built to stealthily and intelligently collect sensitive user data through screenshots. To understand the threat, we built Screenmilker, an app that can detect the right moment to monitor the screen and pick up a user's password when she is typing in real time. We show that this can be done efficiently by leveraging the unique design of smartphone user interfaces and its public resources. Such an understanding also informs Android developers how to protect this screenshot capability, should they consider providing an interface to let third-party developers use it in the future, and more generally the security risks of the ADB workaround, a standard technique gaining popularity in app development. Based on the understanding, we present a mitigation mechanism that controls the exposure of the ADB capabilities only to authorized apps.**

I. Introduction

With the progress in Android smartphone and tablet technologies comes the rapid expansion of their application markets. Until October 2012, already about 700,000 applications (*app* for short) have been built for Android, with 25 billion of downloads reported by Google Play [1]. This surge of new apps puts tremendous pressure on Android's security protection: malicious apps have been reported [2] and security vulnerabilities are found on Android [3]. Fortunately, so far Android has fared well, with less 0.5% of malicious apps discovered on its market [4], thanks to its security-sensitive design that confines individual apps within their own sandboxes and mediates their access to critical system resources through a permission mechanism.

NDSS '14, 23-26 February 2014, San Diego, CA, USA
 ISBN 1-891562-35-5
http://dx.doi.org/10.14722/ndss.2014.23049

ADB-based workaround. However, app developers continue to push the envelope, coming up with new apps that challenge the limits of Android security design. Increasingly, they are unhappy with the resources provided by Android APIs and strive to acquire new capabilities that enable the apps to do what ordinary Android users cannot do, such as wireless tethering, system backup, new font adding and others. One way to get such capabilities is through phone rooting, which allows a user to attain the root privilege of her device. The problem is that rooting a smartphone typically voids its warranty and could also damage the device [5].

Given the risks of rooting phones, increasingly, people are seeking its "no-root" alternatives. The most widely used workaround for this purpose is based on the Android Debug Bridge (ADB): one can connect her smartphone to a PC to launch the ADB and through it further invoke a service running with its privilege on her phone. An app can then communicate with this service process to acquire the resources the Android APIs do not provide, even after the phone is unhooked from the PC. Note that this approach is completely legal and also pretty convenient to use with assistance of proper instructions and installation scripts [6] (see the video demos provided by app developers for such screenshot and USB tethering apps [7], [8]). Actually, many apps, such as sync and backup [9], USB tethering [10], screenshot apps [6], [11], etc., have been using this "no-root" technique. Those apps have already accumulated tens of millions of downloads on Google Play and gleaned acclaimed reviews (see [12], [6], with reviews posted as recently as this July). However, this workaround escalates a normal app's privileges and exposes the ADB-level resources. Until now, its security and privacy implications are less clear to the app developers, and apparently also to the Android designers. To better understand this potentially security-critical issue, we performed the first study on it through an in-depth analysis of Android screenshot tools, a prominent example of such ADB-based apps.

Android screenshot. Screenshot is a highly useful functionality that Android falls short of. Until Android 4.0 (the Ice Cream Sandwich release), ordinary users cannot take screenshots of their phones at all. Even though the newer versions of Android let the users do that by holding down the Power and Volume-Down buttons, it still does not provide any API to allow an app to programmatically get the images of other apps. Given

the demand of programmatic screen capture (e.g., extracting only part of the screen, triggering this function using other mechanisms than manual clicking on those two buttons) and the fact that still over 60% of Android distributions in use today are older than 4.0 [13], both app developers and phone users have to resort to the ADB-based workaround in the case that they do not want to root their phones. Indeed, all "no-root" screenshot tools on the market, such as Screenshot Free [11], Screenshot UX [6], etc., are built upon the ADB. They are extensively used by Android users [6], [11], having been downloaded for millions of times (Section II).

These tools rely on an ADB-level server to take screenshots, according to the parameters set by a client typically running as an ordinary Android app. Since Android does not support the Inter-Process Communication (IPC) between an app and a background native server, these two parties have to talk to each other through the TCP sockets opened on the same device, a standard replacement for the IPC on Android [14]. Although this local-socket channel plays an important role in Android design (which has been extensively used by OS processes such as Zygote [14]), it has not been mediated at all. As a result, it opens a new venue to the attack that has not been considered in the Android security model: we found in our research that not only the screenshot client, but *any* app with the INTERNET permission can connect to the local server and command it to take screenshots. Note that this problem is not limited to the screenshot apps, as other ADB-based apps can have the same vulnerability (Section II). Even more importantly, given that both the ADB workaround and the local-socket channel are pretty standard techniques, in the absence of in-depth understanding of their security risks, their further utilizations by the app developers and the Android designers are almost certain to bring in even more serious security hazards.

Screenmilker. In this paper, we describe our design and implementation of a malicious app, called *Screenmilker*, that intelligently and stealthily collects user secrets through this ADB channel, using nothing but its INTERNET permission. The objective of our study is to understand how serious the problem can be: once an ADB-level capability has been leaked out to an unauthorized app, the app can immediately leverage other public Android resources at its disposal to come up with a practical attack. This asks for a serious effort to control ADB-level services and their interactions with other Android apps (through the local socket connections). Also, our technique can inform Android developers how the screenshot capability should be managed once they consider providing an API for it in the future: otherwise, one who gets the permission to use such an API can do something completely unexpected.

Actually, simply getting the screenshot capability does not make collection of confidential user data trivial, particularly when you want to do it stealthily and with a minimum set of permissions. Screenshots are large, at least 8 Kbytes with an 800×480 resolution (Section III). If the adversary does not know when to take them and has to continually take many shots for a long time, her app may end up consuming a significant portion of the SD storage space (which needs a permission to write on) and a large network bandwidth for delivering images. Also, analysis of images locally using OCR (optical character recognition) tools requires a lot of CPU/memory/storage resources. Such activities can easily arouse a smartphone user's suspicion and get caught by malware detection.

Our development of Screenmilker, however, shows that there are plenty of resources already on the phone the malicious app can utilize to make those activities go unnoticed. Specifically, we can use side-channel information leaked by Android and image fingerprinting to identify the right moment for taking a shot, for example, when the user is entering her password. The content of the screenshot images can also be efficiently analyzed using the features of smartphone user interfaces, which enables the malicious apps to "milk" sensitive information from these images in a real-time manner. We implemented a screenshot keylogger that can continuously and accurately pick up the password the user types without being noticed. Our design does not use the SD storage at all and only sends a very small amount of data across the Internet to the malware owner. We further developed a simple mitigation and offer suggestions to the Android developers on how to mediate the access to the ADB capabilities and the local-socket channel, as well as how to control the programmatic screenshot interface once they decide to provide one in the future. A demo of our malware is posted online [15].[1]

Contributions. We summarize the contributions of the paper as follows:

• *Understanding of the security risks of the ADB workaround and the local-socket channel*. We preliminarily studied the practice to utilize the ADB-level capabilities to enrich the functionalities of Android apps, a legitimate, extensively used yet completely unregulated approach. This approach has been used to implement assorted apps, e.g., sync and backup [9], USB tethering [10], and screenshot apps [6], [11]. Our research reveals the security risks of this approach (using screenshot apps as an example), particularly the standard local-socket channel it uses to communicate with ordinary apps, and explores the ways to mitigate the risks and manage the channel (which has also been widely utilized for other inter-process communication on Android [14]).

• *New techniques for targeted, stealthy and real-time collection of sensitive information from screenshots*. We developed new techniques that utilize Android public resources and its unique interface design to make our malware stealthy and effective. Our study demonstrates that the threat of information collection from screenshots is realistic and serious.

• *Implementation and evaluation*. We implemented our technique [15] and evaluated its effectiveness.

Roadmap. The rest of the paper is organized as follows: Section II analyzes the existing screenshot apps on Android; Section III elaborates our design and evaluation of our screenshot malware; Section IV reports our experimental study on the malware; Section V describes the mitigation of the threat and suggestions for the Android developers; Section VI compares our work with prior related research; Section VII concludes the paper and discusses future research.

[1] We do not trace the visitor.

II. Programmatic Screenshot on Android

Whether it is a developer showing her app's user interfaces on Google Play, or a blogger presenting a tutorial on her blog, screenshots are one of the most powerful visual aids to demonstrate one's innovative idea on how to use smartphones. This important capability, however, has not been well supported by Android: as discussed before, Android users cannot take screenshots on 1.X to 3.X, which are still running on most Android phones in the market [13]. From 4.0, the capability is provided but can only be used through holding on the Power and Volume-Down buttons. There is still no programmable interface that allows users to customize the way to utilize this capability. As a result, for most of the people who do not have proper devices to accommodate the latest Android versions or are looking for advanced screenshot functionalities (e.g., consecutive shots or screenshot-taking triggered by events), they have to find a workaround to do that.

To get the programmatic screenshot capability, an app needs to have a *signature-level* permission from the system, which, in this case, requires the app to be signed by Android development team [16], a condition considered to be unrealistic for most third-party app developers. In the absence of such a permission, those developers have to resort to two standard workarounds: (i) requiring users to root their phones; (ii) leveraging an Android native executable as a *proxy* to access ADB's capabilities. Rooting the phone is oftentimes not an option since it can invalidate a user's manufacturer warranty and also could damage the phone [5]. Hence, as long as ADB has the capability that a particular app needs, majority of the developers opt for the latter workaround. Indeed, this is the case for screenshot apps.

In the rest of this section, we first provide an overview of ADB's capabilities, and describe the standard approach for apps to obtain these capabilities though a proxy. Then we detail a common vulnerability that we found on these apps, and present a measurement study on today's screenshot apps to demonstrate its prevalence.

Android permission and ADB workaround. ADB is a command-line development tool that allows developers to communicate to Android devices connected to a development system (e.g., a PC). For this purpose, it has a set of unique capabilities specified under the Android permission system, which we briefly survey as follows. Android protects system resources with permissions, which apps need to request before they can access a specific resource in a controlled way. Depending on the risk implication of each permission, Android assigns it a *protection level* so as to handle it differently from those in other levels. More specifically, most permissions have either the *normal level* or the *dangerous level*. Those in the normal level are considered to be risk-free (e.g., the permission to set an alarm), and are therefore automatically granted to requesting apps. On the other hand, permissions in the dangerous level can bring in security risks (e.g., the permissions to open network sockets), which the system only gives them to apps under the user's explicit consents (e.g., asking the user to press a confirmation button whenever the permission is requested).

For the permissions with the highest risk implications (e.g., that for accessing the frame buffer), Android classifies them into the *signature level*. Such permissions are only given to the apps signed by a trusted party (like the Android developer team). For this purpose, Android maintains a database of trusted certificates from such parties. Most third-party app developers are not among them and therefore their apps do not have this level of permissions.

TABLE I. Examples of the signature-level permissions granted to ADB in Android 4.1.1. The full set of ADB's permissions can be found in `/etc/permissions/platform.xml` in each system.

Permission	Description
BACKUP	Control backup and restore process
CHANGE_CONFIGURATION	Modify the configuration
DELETE_PACKAGES	Delete packages
DEVICE_POWER	Access low-level power management
GET_DETAILED_TASKS	Get full detailed information about tasks
INJECT_EVENTS	Inject user events to any window
INSTALL_PACKAGES	Install packages
READ_FRAME_BUFFER	Take screenshots

On the other hand, some of the signature-level permissions are valuable to users. Examples here include the one for modifying the configuration, that for controlling the backup process, and, of course, the permission for taking screenshots. An Android user who wants to use the resources such permissions guard can only do so through a *default access mechanism*, which is not open to third-party apps: for example, modifying system configuration needs to go through a system app and screenshots can only be taken through the Power and Volume-Down button combination. Some signature-level permissions considered to be useful to app development are given to ADB. Table I illustrates a few such examples for Android 4.1.1.

Among those default access mechanisms, ADB turns out to be probably the only one that the developer can use to programmatically access some signature-level resources. This system tool can run Android native executables, which can be leveraged to create a proxy for her third-party app to attain the desired signature-level permissions. Actually, such a treatment is the standard workaround for developing apps that need such permissions. It is also considered to be legitimate, given the fact that apps using this approach have already accumulated tens of millions of downloads on Google Play and no objections have been heard from Google. Given its popularity among the app developers, the security implication of the workaround, however, has never been well understood. Our study shows that this approach indeed brings in serious security risks, which could lead to devastating consequences if the developers are not aware about them.

What can go wrong. The ADB workaround consists of two steps: (i) run a customized Android native executable through ADB to acquire the desired permissions; (ii) establish a communication channel between the executable and a third-party app to access protected resources with the permissions. We here elaborate each of these steps.

As a development tool, ADB allows the developer to start a remote shell in the target Android device connected to a development PC. Through the shell, she can execute programs with ADB's signature-level permissions. Specifically, the developer first packs all methods that require signature-level permissions along with a front-end that communicates with a third-party, unprivileged app into an Android native executable, connects the target Android device to the development PC, and then triggers the executable through the ADB shell command. Once the executable is operating, the developer can disconnect the device from the machine, and the unprivileged app can use the front-end to call the packed methods, so as to access the resources protected by signature-level permissions. To simplify this process and make such a two-part app easy to use, the developers usually implement a script that automatically carries out the necessary actions. Such a script can even include the ADB binary to avoid having the user configure the Android SDK (see the video demo [7], [8]). Therefore, a user only needs to execute the script once per reboot and then she can run the unprivileged app to get signature-level permissions. This kind of apps are actually extremely popular among users, due to the additional features they provide. With proper instructions, the popularity of the apps such as Screenshot UX [6] and No Root Screenshot It [12] has demonstrated that users are able and willing to set up the ADB workaround to obtain the additional features.

A tricky issue here is the communication between the ADB-level proxy and the unprivileged app. Android blocks the system-wide Inter-Process Call (the System V IPC)[2] and instead restricts the IPC channel to delivering *intents* among the apps running in Java Virtual Machines (JVM)[3]. As a result, the interactions between a native executable and an unprivileged app need to go through other channels. Typically, this happens through local network sockets: the proxy opens a network service and the app makes TCP connections locally to exchange data with the proxy. This channel has been extensively used on Android (e.g., the Zygote process) [14].

Such communication, however, can be very problematic. What we discovered in our research is that Android does not protect communication channels other than the IPC, and as a result, there is no access control on the local-socket channel at all. Specifically, unlike the IPC, where developers can leverage the permission model to ensure that a process can only talk to an authorized party, when it comes to the local socket, developers have to implement their own security mechanism to prevent unauthorized access to the channel. Unfortunately, we found that existing security implementations in those apps are incompetent in this perspective, and routinely expose signature-level permissions to unauthorized apps.[4]

Android screenshot apps. To understand the pervasiveness of the ADB workaround and its security implications, we thoroughly studied the Android screenshot apps, a prominent example of third-party apps that need signature-level permissions. As mentioned earlier, we focused our study on no-root apps. Table II describes 8 such apps on Google Play. They are widely used and well received by the users despite their seemingly complicated installation procedures, particularly when they come with a clear guidance or scripts that help the users conveniently set them up on the phones [8]. Prominent examples here are Screenshot UX and No Root Screenshot, both of which glean acclaimed reviews as recently as this July (with average scores 4.4 and 4.3 stars out of 5, respectively) [12], [6].

TABLE II. LIST OF ALL NO-ROOT SCREENSHOT APPS ON GOOGLE PLAY. THE LIST WAS COMPILED IN FEBRUARY, 2013. ALL APPS USE ADB TO GAIN THE SIGNATURE-LEVEL PERMISSION AND ARE VULNERABLE TO SCREENMILKER'S ATTACK.

App Name	Total Installs
Screen Capture – No Rooting 2.2	1,000,000 – 5,000,000
Screenshot Free	1,000,000 – 5,000,000
Screenshot UX Trail	1,000,000 – 5,000,000
No Root Screenshot It	100,000 – 500,000
Screenshot and Draw Trail	100,000 – 500,000
Screenshot Ultimate	100,000 – 500,000
ShakeShot Trail	100,000 – 500,000
NoRoot Screenshot Lite	50,000 – 100,000

In our research, we installed and checked each of those apps to analyze how it obtains the capability of screenshot taking. We found that all these apps utilize the ADB workaround to obtain the necessary permission. This finding shows that such a workaround has become a standard approach to get signature-level permissions without rooting.

In addition, as shown in Table II, these apps have millions of total installs, as reported by Google Play, despite the extra effort required to set them up (installation of Android SDK, etc.,), although the automated installation scripts from developers have vastly simplified this process. We believe this popularity is due to the unavailability of the screenshot capability prior to Android 4.0 and the advanced screenshot features these apps provide.

Finally, we analyzed the communication channel each of these apps uses to coordinate its ADB proxy and unprivileged component, particularly the access control they enforce on such a channel. All these apps turn out to rely on local network sockets for the communication, and, unfortunately, have literally no protection on the channel at all. As a result, once the ADB proxy is activated, *any* app can request a service from it to take screenshots at anytime, without any restrictions. To demonstrate that such vulnerability can be practically exploited, we utilized tcpflow [17], a flow analyzing tool, to reverse-engineer these apps' custom communication protocols. Based upon the knowledge about the protocol format, we further built a malicious app that talks to the ADB proxy and commands it to take screenshots. The security risk here is severe, since if one of these screenshot apps is running on an Android device, all other apps that have the INTERNET permission immediately acquire the capability to take screenshots without the user's consent. In Section III, we show that with this capability exposure, the malicious app can

[2]The details are documented in bionic/libc/docs/SYSV-IPC.TXT in the Android source.

[3]Although Android allows a native-code *library* to use the IPC channel, it blocks parts of the JNI interface to prevent a native *executable* to access the channel.

[4]Note that the recent addition of the USB debug whitelist in Android 4.2.2 doesn't address this issue, since ADB here is used to launch a legitimate executable the user intends to run from her PC.

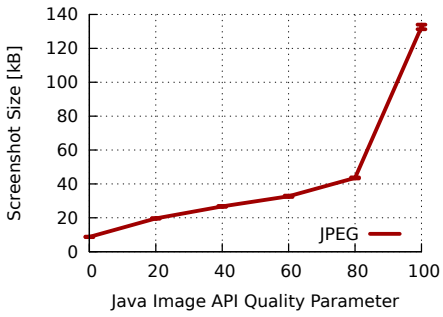

Fig. 1. The size of screenshots with an 800×480 resolution in the JPEG format. The size increases from 8 to 130 Kbytes as the quality increases. Since the size difference between colored and gray-scale screenshots is negligible, we only present the result of colored screenshots.

be made so effective that it can accurately identify and extract sensitive user data on the screen at an exceedingly low cost.

Generality of the problem. Our preliminary work, as reported in the paper, focuses on screenshot apps. However, the underlying problem, the security implications of the ADB workaround and its related local-socket channel, are much more generic. Actually, we found that in a similar way, sync and backup apps [9] and USB tethering apps [10] also inadvertently expose their ADB capabilities to unauthorized apps, which could lead to disclosure of sensitive user data or circumvention of Android firewall protection, although the whole issue needs further investigation. More specifically, backup apps send request to an ADB proxy (through the local-socket channel) that is capable of backing up the phone user's data to SD cards or online storage services. However, without proper protection, an unauthorized app with only the INTERNET permission can leverage this proxy to store private user information (e.g., contact, SMS, etc.) to the SD card and then gets access to it in the background. For the USB tethering apps, they utilize an ADB proxy to forward their TCP packets to the Internet. A risk here is that a malicious app will also have the access to the proxy (through the local-socket channel) to sneak out its data, avoiding the inspections by the firewall apps (e.g., Android Firewall [18], DroidWall [19]) on the phone. This indicates that serious efforts need to be made on Android to regulate the use of the ADB workaround and the local-socket channel.

III. Design and Implementation

Even with the screenshot capability, it is still challenging for a malicious app to stealthily collect confidential user data, due to the size of screenshots. Figure 1 illustrates the image sizes with an 800×480 resolution in the JPEG format. We adjust the parameter of the Java image API to create screenshots of different qualities. As a result, the average size varies from 8 to 130 KB.

Given this image size, a malicious app cannot afford to randomly take screenshots, hoping that some of them happen to contain sensitive user data. Consider an ordinary user's 3G/4G data plan, which is typically 2 GB per month. To keep a low profile from data-usage monitoring, the app clearly cannot use a significant portion of this limit. Suppose that it spends 5% of the limit on sending screenshots to its owner. All together, the app can take about 26 shots (130-Kbyte images) per day. If it does not know when to take the screenshots, the chance of getting the right information is rather low. Furthermore, high-value information such as password typically does not show up on the screen. To get it, the app needs to repeatedly take screenshots when the smartphone user is typing. Under the constraint of upload bandwidth for today's 3G/4G network, which is typically about 2 Mbps, only 2 screenshots can be sent out every second. This is often insufficient to catch up with the user's typing speed. Alternatively, the malicious app has to temporarily keep those images on SD card, which needs a permission and therefore may raise suspicion, making the attack less likely to succeed.

However, we show in this section that Android has already made a lot of resources available, which enables a carefully designed malicious app to grab high-value user data in an almost invisible way. In our research, we built Screenmilker, an example of such apps to understand how the attack can be done. Screenmilker can masquerade as any legitimate app that needs the INTERNET permission. Once installed, it can detect the presence of the ADB proxy, which it can use to take screenshots. During its operation, the malware monitors other apps' activities, intelligently determining when sensitive user data is on the screen before taking screenshots. In addition, Screenmilker employs a lightweight data-analysis mechanism to extract a very small set of sensitive user data from the screenshots in real time. This capability allows it to pick up whatever the user types even when the content cannot be directly observed from the screen (e.g., when the password is being entered), and still maintain its stealthiness. What Screenmilker can do strongly suggests that any future attempt to offer the programmatic screenshot interface needs to be well thought-out, to prevent it from being abused by the party that has this permission.

A. Overview

Adversary model. In our research, we consider an adversary who can disguise Screenmilker into another genuine app to trick users to install it on their devices. The malicious app only needs the INTERNET permission, which is requested by a vast majority of apps, due to the need of retrieving advertisements from the Web [20]. We assume that the target device has one of such no-root screenshot apps installed, which, however, is not known to Screenmilker *a priori*. Also, we assume that the device owner pays attention to mobile-data usage and utilizes other tools to discover the problematic behaviors of the apps on her phone, such as taking a large amount of SD-card space, memory and CPU resources.

Architectural overview. Figure 2 depicts the architecture of Screenmilker. The app consists of two main components: (i) the runtime situation detection component that identifies the right moment to take screenshots; (ii) the real-time data extraction component that analyzes the screenshot to collect the most valuable personal data.

Screenmilker is designed to effectively collect sensitive user data while maintaining its stealthiness. To this end, Screenmilker adopts a targeting strategy to control its use of CPU, memory, and network resources: it performs a lightweight surveillance on a set of apps that operate on high-value data, such as banking apps, and only takes screenshots

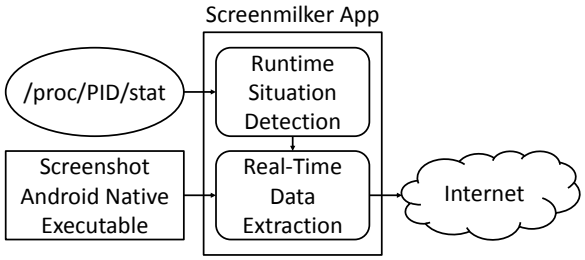

Fig. 2. The architecture of Screenmilker. Screenmilker probes `/proc/PID/stat` to detect target apps' activities. When the target apps are active, Screenmilker takes screenshots and extracts confidential user data. Then, it sends the extracted data to the adversary through the Internet.

when these apps are active and also in a state where such confidential data is being displayed on the screen. This surveillance is performed by the runtime situation detection component, which leverages Android pubic information, the target apps' CPU, memory and network activities and the fingerprints extracted from the current screen image to determine whether those apps are operating in the foreground and whether they are in the right states. Once the picture-taking moment is detected, it notifies the real-time data extraction component to milk confidential user data from the screen.

The data extraction component calls the ADB proxy to take screenshots and performs a lightweight analysis on the images. The objective here is to milk a small set of high-value data out of those images in real time. The original screenshot images can then be thrown away to limit network and memory consumption . For this purpose, Screenmilker is designed to take advantage of the unique way that smartphones display content, which is typically well-formatted and rather predictable. For example, the positions of digits on a phone number and alphabets on contacts can be relatively easy to determine on the screen, and their fonts are pretty standard (Figure 7). Therefore, our app can recognize their content through a highly efficient approach. What is more challenging here is to obtain passwords, which often do not show up in plain text on the screen. To address this issue, we show that the data extraction component can be used as a keylogger that efficiently recovers a user's password from the screenshots when she types on the Android soft keyboard.

B. Runtime Situation Detection

Before it starts collecting a user's secrets from her smartphone, Screenmilker needs to figure out what it can do on the phone and what is the best time to do that. More specifically, its situation detection component makes sure that the malware can access the screenshot capability on the phone, through an ADB-based proxy. Then, the component tries to determine whether target apps are present and if so, puts them under surveillance, detecting whether they are running in the foreground and also at the right program state where confidential user data is on display. Here we show how this can be done.

Detecting the screenshot proxy. Screenmilker employs a multitude of mechanisms to detect the exposure of the screenshot capability. It carries a database for the package names and other features of the screenshot apps listed in Table II, and after it is installed, it uses the public API `PackageManager` to get the list of apps also on the phone. From the list, Screenmilker can identify those screenshot apps and other target apps (discussed later). Furthermore, whether they are currently running can be known through the Linux command `PS`, which provides detailed information about all active processes. Note that using this command does not need any permission. Alternatively, the malware can simply check the TCP ports the ADB proxies of these screenshot apps use: since these proxies always listen on fixed ports, Screenmilker can probe these ports and once connected continue to request a screenshot. In this way, the malware can acquire the screenshot capability if it has been exposed.

Monitoring target apps. In the same way, Screenmilker identifies the presence of its target apps (such as banking, healthcare apps, etc.) through `PackageManager` and the timings when they are executed by running `PS` periodically. Moreover, the `PS` command also returns the process ID (PID) of the target app. This information enables Screenmilker to locate the process statistic files of the app under `/proc/PID/stat`, which includes the app's memory and CPU usage information. Such data can then be used to infer the app's current state: for example, when the user starts entering her password.

Specifically, to find out when a target app is receiving user inputs, Screenmilker checks whether the app is running in the foreground (discussed later), and if so, starts a continuous monitoring on the default soft keyboard app (`com.google.android.inputmetohd.latin`). After retrieving the keyboard app's PID using `PS`, Screenmilker reads the corresponding `/proc/PID/stat` file once every 100 ms to detect the change of the app's CPU usage.[5] In our research, we found that the accumulated *user CPU time* for the keyboard app is a good indicator of whether a key is being pressed on the soft keyboard. Hence, whenever Screenmilker observes that the accumulated user CPU time increases, it concludes that the app is at the state receiving the user's typing inputs, and therefore moves on to take shots to capture her keystrokes (Section III-C). Similarly, information such as the timing when the user is making a call can also be inferred from the change of accumulated user CPU time, once the user is pressing an individual's name on her contact list.

Detecting display states. Screenshots can only get the content displayed in the foreground of a smartphone's screen. This requires Screenmilker to take the right picture at the right time. To this end, our design uses a strategy that periodically grabs screenshots (see Section III-C) whenever the target app is found to be running, and quickly checks each image using a fingerprint for the app's user interface (UI).

The first step to check the image is to determine the orientation of the phone. This can be easily done by calling the Android API `getRotation` under the `Display` class. The API can be invoked by the party without any permission. They return the degrees of the rotation (0, 90, 180 or 270) from the phone's natural orientation, which are used to get the position of the phone (landscape or portrait).

[5]We monitor the accumulated *user CPU time*, which is the 16th field in the `/proc/PID/stat` file.

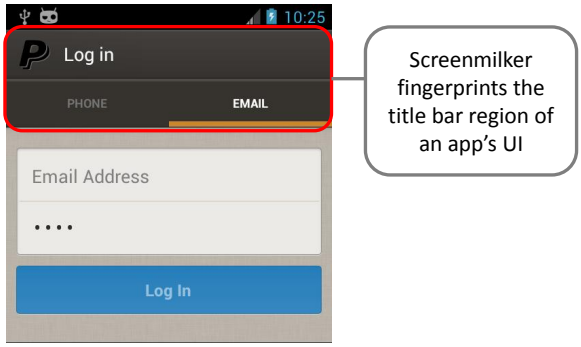

Fig. 3. An example of fingerprinting the PayPal login UI. Screenmilker calculates the CRC32 value of the title bar region, and then looks up the hash table to determine if the user is trying to log in.

Fig. 4. The keystroke animation of the default soft keyboard in Android 4.1.1. When a key is pressed, the color of the button turns blue, and a pop-up appears above the key. Then, the color of the button returns back to gray. Finally, the pop-up disappears.

Based on the orientation information, Screenmilker takes a shot from the screen and then quickly extracts part of the image to search a set of fingerprints for target apps' activities of interest. For example, if the adversary wants to collect user's PayPal password, he can instruct Screenmilker to start taking screenshots when the user is logging into her PayPal account.

Specifically, we implemented a lightweight image fingerprinting technique, which builds a hash table to map the CRC32 value of the title bar region of an app's UI to an Android activity. After taking a screenshot, Screenmilker quickly calculates the CRC32 value of that bar region on the image (which is extracted according to the orientation of the phone) and looks up the hash table to decide whether the target app is running in foreground and also in the state of performing an activity of interest. Figure 3 illustrates an example of how a screenshot of PayPal and the CPU usage of the soft keyboard are used together to fingerprint the user's login activity.

C. Real-time Data Extraction

Screenmilker uses the real-time data extraction component to efficiently collect confidential user data from screenshots and deliver it to the malware owner. This is achieved through a very lightweight manner in which only a small amount of CPU/memory resources (compared with that of an ordinary, legitimate app) and a negligible amount of mobile data are consumed during the analysis. Here we show how this can be done through two examples: an image-based keylogger for gathering passwords and a technique for picking up a phone user's contacts.

The high-level idea. The most direct way to collect confidential user data (e.g., search terms, emails, instant messages, etc.) from screenshots is to take pictures and run an OCR tool on them to extract valuable content. To do this, the malware needs to either continually stream the pictures to the adversary's server for analysis, or to run OCR locally on the phone. However, neither of these options is practical: the former, as discussed before, will end up consuming too much user network data and become easy to detect; the latter requires installation of a large software component and uses a significant amount of CPU/memory resources, making its behavior conspicuous.[6] In our research, we found that these technical challenges can actually be overcome by the malware author, who can find a much simpler solution to make those malicious activities almost invisible.

More specifically, what we discovered is that the unique features of smartphone's UI offer a shortcut to highly efficient content analysis. As discussed before, an app's UI is typically simple and stable across different devices, as long as those devices are running the same Android version, which can be found out, without any permission, from the Android API `android.os.Build.VERSION`. Given the size of a smartphone's screen (from the public API `getRealSize`), its display feature (from `getMetrics`, also a public API) and its orientation, the positions of the content on the screen, as displayed by the target app, are pretty much determined. For example, each contact shown by Android's "Contact" app (`com.android.contacts`) sits at a fixed position when the app is activated and can only be moved up and down through scrolling. As another example, every key's position on the soft keyboard's image is completely predictable. Leveraging this observation, our approach runs a high-performance checksum (CRC32) to fingerprint some predictable positions on a screenshot to recover the content of interest.

Real-time keystroke analysis. To get what a user types, we can simply grab the screenshot to analyze the image content within a text-box. This approach, however, turns out to be less effective in practice: the positions of the text being edited on the screen are quite dynamic and less predictable. More importantly, for high-value inputs, particularly passwords, their content is typically covered, as illustrated in Figure 3. A more reliable source that we can get such information from is the whole typing process, a sequence of screenshots that record the dynamics of the soft keyboard.

To get these images, Screenmilker repeatedly takes screenshots from the screen once the user is found to enter high-value inputs into a target app (which is identified using the situation detection technique elaborated in Section III-B). Our current implementation can achieve a speed of 6 shots per second without incurring noticeable performance impacts (Section IV). To avoid consuming too much resource, Screenmilker does not store these images, nor does it stream them out of the system. Instead, it performs a highly efficient analysis on them based on the features of the soft keyboard to get the value of each key between shots.

The soft keyboard is one of the main input methods on Android devices. It is a service utilized by most Android apps that handle user inputs. Once activated, the app displays a keyboard on the screen and provides a set of UI techniques to assist user's typing. More specifically, it highlights every

[6]For example, the popular Android OCR library, tess-two, requires more than 30 MB additional storage for its associated files, and takes around 400 ms to extract private information from a contact list screenshot.

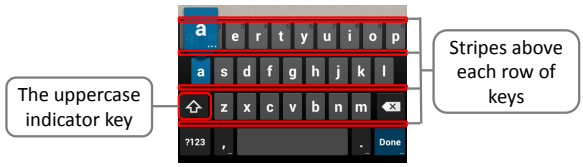

Fig. 5. Screenmilker uniquely determines the key's content with the stripes right above each row of keys and the uppercase indicator key.

pressed key by changing the color of its button and popping up a block to amplify the content of the key. Once the letter is entered, the color of the button first fades away and then the pop-up disappears. This input process, as it happens on Android 4.1.1, is illustrated in Figure 4. Screenmilker leverages the features of the animation to build a classification model that maps the screenshots to different key values.

As discussed above, our idea is to fingerprint the image of each keystroke using a checksum calculated on a minimum set of image features that characterize the key. The challenge here is that the content of the images changes during the animation, making this fingerprinting difficult. To address this issue, we need to find out a relatively stable part of the image that is also associated with the key. The most suitable candidate here is the pop-up block, which is the first to show up and the last to disappear. Also its color and other features keep unchanged in the process. Therefore, our strategy is to fingerprint the position of this box. Specifically, we look at the stripes right above each row of keys, as illustrated in Figure 5. When a key is pressed, some of these stripes overlap with the pop-up block, which results in changes of color at some positions on them. These positions are deterministically related to the key being entered, thereby uniquely characterizing the key.[7] Also their colors, lengths and shapes are stable throughout the presence of the pop-up block, which, as illustrated in Figure 4, lasts for the longest duration during the whole key-entering process. This gives Screenmilker the best opportunity to get the key.

The content of the soft keyboard can be switched between different character sets, including lowercase alphabets, uppercase alphabets, and signs-and-numbers. The state of the keyboard in terms of the selected letter set is visually marked by the uppercase indicator key and the character set indicator key (see Figure 5). This state, together with the position of the key, uniquely determines the key's content. Our implementation of Screenmilker fingerprints the state using the uppercase indicator key: when it is highlighted, the keyboard is using uppercase letters; when it is not, the keyboard accommodates lowercase letters; when its content is changed to another symbol, what is in use is the sign-and-number set.

Altogether, Screenmilker identifies a key from the stripes carved out from the top of each key row and the uppercase indicator key cropped from the keyboard image. These image fragments are then used to calculate a CRC32 value for recovering the content of the key. For this purpose, we built a hash table into Screenmilker, which maps precomputed checksum values of these features to their corresponding key. This table is very small: each CRC32 has 4 bytes and all letters, numbers and symbols together take 114 keys; only 456 bytes are needed to keep these values. Also, the hash computation and table look-up are extremely efficient (see Section IV), which enables our app to work as a capable keylogger that collects the user's keystroke inputs in real time (see our demo [15]). Figure 6 summarizes the workflow of this keystroke-extraction approach.

Real-time contact collection. The fingerprinting approach discussed above can also be applied to infer other well-format sensitive data from the smartphone screen. In our research, we implemented a technique to recover the phone user's contacts from a screenshot image. Again, Screenmilker first detects the operation of the "Contact" app through `PS` and further fingerprints its UI to detect its presence in the foreground. Then, our malware quickly analyzes the screenshot image to extract the contact information. Specifically, as illustrated in Figure 7, the horizontal position of each contact is fixed once the screen's orientation, display feature and size are known. The tricky part is its vertical position, which changes when the user scrolls up and down the list on the screen. To get the positions of the contacts, our approach picks up a vertical line of pixels from the screenshot image, which intersects with all horizontal partition lines for individual contacts (Figure 7). By inspecting the color of individual pixels on the vertical line, Screenmilker obtains the vertical position of each contact, between two neighboring partition lines. The start position of the contact's text is then identified.

The rest part of the analysis is rather straightforward. Screenmilker inspects the image fragment cropped from the position of a contact, and further segments the image into individual character blocks, according to the space between these characters. For each block, it calculates its CRC32 value and then maps the value to the content of the character using a precomputed hash table. This technique is also used to analyze the screenshot image taken when the user is making a call. From the image, our malware locates the name and phone number of the party in the call, whose positions are fixed. The above technique is then run on the image fragment with the name and the number to extract the information.

Discussion. The real-time analysis technique utilized by Screenmilker is highly efficient and effective. However, what it can do is still limited by the capability of screenshot taking on Android. Particularly, although Screenmilker can accurately detect the key being entered when the screenshot is taken in the duration of a keystroke animation, it may not work well under some circumstances. For example, a user may press keys in a rate higher than the rate of picture taking that the Android native executable supports. In this case, Screenmilker may only recover part of the user's inputs. To address this issue, Screenmilker can leverage a dictionary to reconstruct the inputs. Moreover, in the situations where the user types in the same input multiple times (e.g., entering a password), Screenmilker can aggregate what are learnt from multiple rounds of the input entering to piece together the input content. For example, in our implementation, we let Screenmilker count the number of displayed password dots to infer the positions of the extracted keystrokes and make up for the missing letters

[7]For some early Android versions, such as 2.X, when a key on the first row is pressed, the stripe on top of the row does not overlap with the pop-up block. This issue can be addressed by widening the stripe to include part of the keys, which captures the color change happening to the one being pressed in a same way.

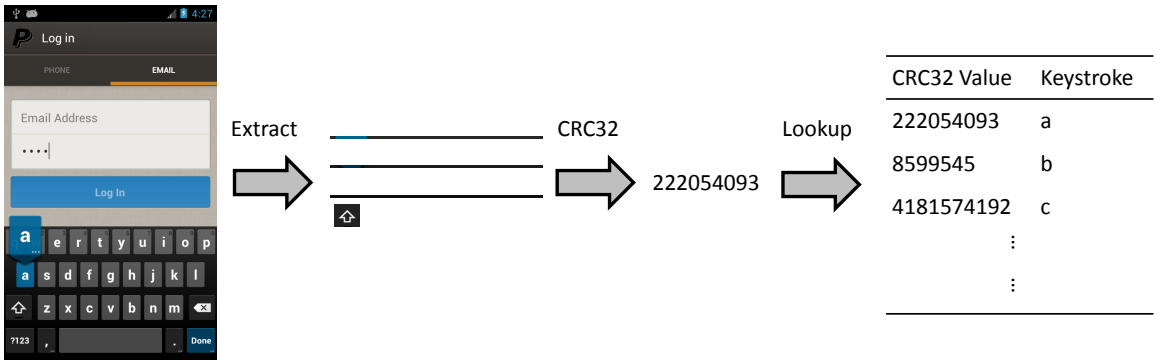

Fig. 6. The real-time keystroke extraction workflow. Screenmilker first extract the essential features from the soft keyboard screenshot. It then calculates the CRC32 value of these features and looks up the hash table to determine the keystroke.

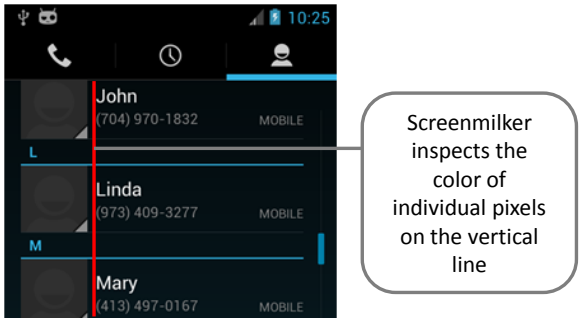

Fig. 7. Screenmilker inspects the color of individual pixels on the vertical line to obtain the vertical position of each contact.

in one round with the letters learnt from other rounds.[8]

Screenmilker analyzes the keystroke animation of soft keyboards (Figure 4) to extract user inputs, and is capable of doing so on various soft keyboards with different color schemes and layouts. To handle keyboards with different color schemes, Screenmilker can distinguish the text and background colors of the keys by examining pixels at specific locations (e.g., the text and corner regions of a key), and convert the cropped image (Figure 5) into a black-and-white image before calculating the checksum. This enables Screenmilker to handle soft keyboards with different color schemes using the same hash table. To deal with keyboards with different layouts, Screenmilker can build a hash table for each target soft keyboard, and utilize the same fingerprinting technique to determine which one is in use. Since the size of a hash table is small (456 bytes for the default Android US keyboard), the adversary is able to include a wide range of popular keyboards without significantly increasing memory consumption. An exception here is the soft keyboards with unconventional animations (e.g., Swype Keyboard [21]). Extraction of keystrokes from them needs new technique, which we leave as future work.

IV. Evaluation

In this section, we present our evaluation study on Screenmilker, which aims at understanding the effectiveness and stealthiness of the malware in collecting sensitive user data. All our experiments were performed on Nexus 4G Android phones (single-core, 1 GHz, ARM Cortex-A8 CPU, and 512 MB RAM) with Android 4.1.1 (kernel version 3.0.31).

A. Effectiveness

App monitoring. We first examined whether Screenmilker can identify the moment when a key is being pressed on the soft keyboard. For this purpose, we conducted ten 10-minute typing sessions, during which Screenmilker probed the `/proc/PID/stat` file every 100 ms to determine whether a key was being pressed. The result of each probe was recorded and was later compared with the ground truth collected through the API `TextWatcher` during the experiment. The study shows that Screenmilker can accurately determine the key press event with a 3.79% false positive rate and a 2.71% false negative rate.

Display detecting. Then, we studied whether Screenmilker can accurately determine which target app is running in the foreground. To this end, we set five banking apps, American Express US, Citi Mobile, Chase Mobile, PayPal, and Wells Fargo Mobile, as the targets and instructed our malware to monitor their operations, together with other unrelated top 50 free apps in Google Play. The experimental results demonstrate that the hash fingerprint of the app's title bar can not only always correctly identify a target app, but also unambiguously differentiate the login screen from other screens of the app.

Keystroke logging. As described in Section III-C, the rate at which Screenmilker can take screenshots is limited by the capability of the ADB proxy. Our experiment reveals that all the native executables found in Google Play (Table II) need an average of 160 ms to take a screenshot (6 screenshots per second). As a result, Screenmilker might only capture part of the user's keystroke inputs.

To understand how effective this key logging can be, we measured its *capture ratio*, i.e., the ratio of keystrokes that Screenmilker was able to get when a user was typing 100 keys consecutively. During this experiment, the rate of screenshots (shots per second) was adjusted from 1 to 5 to examine its impact on the capture ratio.

Figure 8a illustrates the result. The capture ratio increases from 0.27 to 0.76 as the screenshot rate goes up from 1

[8]The same technique enables Screenmilker to handle long-pressed backspace key. In addition, we use a similar technique to detect the location of the cursor, which allows Screenmilker to deal with insertions in the middle of a password.

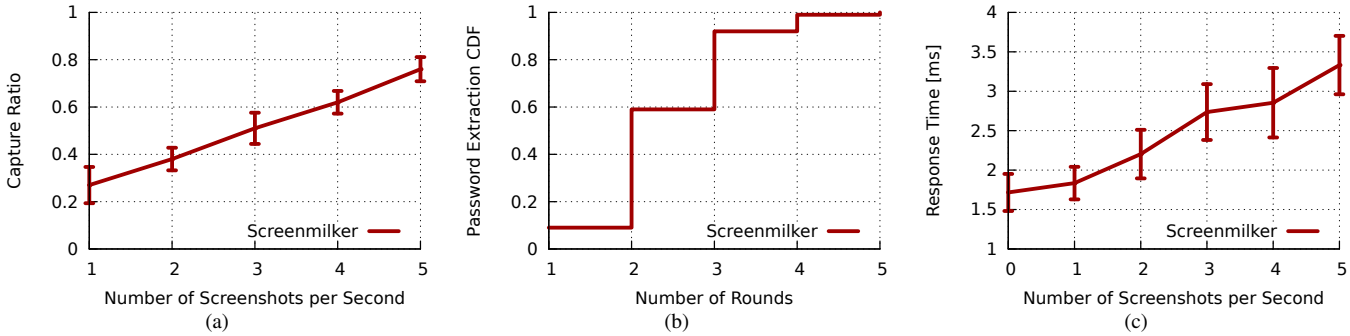

Fig. 8. (a) The capture ratio of Screenmilker to log a single keystroke. (b) The number of rounds for Screenmilker to extract an entire password. (c) The soft keyboard response time when Screenmilker is active.

TABLE III. THE AVERAGE NUMBER OF ROUNDS FOR SCREENMILKER TO EXTRACT AN ENTIRE PASSWORD FROM FIVE BANKING APPS.

App	Average Number of Rounds
American Express US	2.625
Citi Mobile	2.525
Chase Mobile	2.325
PayPal	2.75
Wells Fargo Mobile	2.45

to 5. As mentioned in Section III-C, this ratio is sufficient for a practical attack, such as password collection, when the adversary incorporates the dictionary attacks or combines multiple observations of the user typing the same content, such as her password.

Password extraction. Finally, we combined all the components and evaluated Screenmilker's effectiveness in password extraction. Here, we examined how many observations Screenmilker needs to identify an entire password. Specifically, we assigned unique 10-character passwords to five banking apps, American Express US, Citi Mobile, Chase Mobile, PayPal, and Wells Fargo Mobile. Since these banking apps do not store user passwords for security reasons, they are the perfect target for the password lifting. In the experiment, we ran Screenmilker to extract 40 passwords from each target app during multiple rounds of password entering[9], in the presence of other running apps that served as background noise. Our purpose here is to understand the number of rounds our malware needs before it can successfully identify a whole password. What we found in our study is that whenever Screenmilker caught the moment when a password character was being entered, it always accurately determined the content of the character. However, it could miss some typing moments and therefore had to wait for the user typing the password again to pick up the characters falling through the cracks. Figure 8b illustrates the experimental results: that is, the number of rounds for obtaining a complete password. As we can see here, most of the time (around 60%), our approach got the right password within 2 rounds, and it always did so within 5 rounds.

We further evaluated the effectiveness of Screenmilker in password extraction by examining the average number of rounds that our malware needs to get a complete password from each of the five banking apps. From the results summarized in Table III, we can observe that even when different apps use distinct interfaces for entering passwords, Screenmilker is agnostic to these interface designs and performs equally well on each app.

Contact collection. We evaluated the effectiveness of our approach in collecting other sensitive user data. In this case, we ran Screenmilker when browsing a contact list on Android. The malware successfully detected the operation of the contact app, took a screenshot, and further extracted the text content of the contacts on the list. The data collected was found to be identical to what was on the list.

B. Stealthiness

To avoid being detected, Screenmilker is designed to minimize its resource consumption so as to reduce its CPU, memory, and network footprints. To understand the effectiveness of this design, we measured its stealthiness in the following ways. First, we analyzed the performance overhead perceived by the smartphone user, in terms of the system's response time when Screenmilker is active. Then, we systematically assessed Screenmilker's usages of CPU, memory, and power resources.

Response time. To understand Screenmilker's impact on the system and user interactions, we recorded the response time when the user pressed a key on the soft keyboard until the moment that the system displayed the key on the screen. This time interval was measured through the API `TextWatcher`. The response time is related to the user's perception of anomaly in the system and therefore needs to be limited to keep the malware stealthy.

The result of our study is reported in Figure 8c. As we can see here, such response delays are rather minor. With the screenshot rate going up from 0 (the benchmark, when the malware was not active) to 5, the response time, as observed by the user, moves from 1.717 to 3.332 ms. Given such a small difference, merely 1.615 ms increase in the response time, we believe that the impact of our malware should go unnoticed by the user.

Resource consumption. We further conducted a series of experiments to check whether the presence of the malware can

[9]Through fingerprinting the target app (Section III-B), Screenmilker knows exactly when the user is typing her password to the same app again (Section III-C).

TABLE IV. THE AVERAGE EXECUTION TIME OF EACH DATA EXTRACTION FUNCTION IN SCREENMILKER. THE FUNCTIONS IN BOLD TEXT ARE IMPLEMENTED BY SCREENMILKER, WHILE THE SCREENSHOT FUNCTION IS PROVIDED BY THE ADB PROXY.

	Extraction Function	Time [ms]
General	**Initialize the hash table [one time]**	1.389
	Take a screenshot	161.314
Keystroke extraction	**Fingerprint the image features**	0.388
	Lookup the hash table	0.220
Contact collection	**Obtain position of the text**	3.018
	Segment and map the text	2.916

be detected by inspecting system resource usages. Specifically, we looked into the CPU usage of Screenmilker in terms of execution times for individual malware components. In the experiment, each component was invoked 10,000 times to measure its average execution time. As described in Section III-B, the situation detection component operates once every 100 ms to find out the currently running apps. We observed that at each time the component was invoked, it took less than 1 ms to finish its job. This brings in about 1% CPU overhead, which is negligible.

On the other hand, data extraction takes a longer time. Particularly, to recover password inputs, the malware needs to repeatedly take screenshots, which can have a relatively large impact on system performance. However, this component is only activated when the target app is found to operate in the foreground. Our experiment shows that Screenmilker needs 1.389 ms to initialize its hash table (a one-time cost), and then an average of 161.922 ms to grab and analyze each screenshot. Table IV provides the breakdowns of this execution time: most of it has been spent on taking the screenshot, which is determined by the implementation of the ADB proxy in different screenshot apps; analysis on the images to extract each key only needs an additional 0.608 ms. This level of resource consumption will not cause observable performance degradation, as revealed in our measurement of response time. Similarly, to recover the phone user's contacts only requires an additional 5.934 ms.

We further evaluated the memory usage of Screenmilker, comparing them with what was incurred by popular, legitimate apps, such as Clock, Calculator, Google Talk, and others (see Table V). During those apps' normal operations (e.g., playing music through Pandora Internet Radio, surfing the Web through a browser), their memory consumption was monitored by `PS` over a 5-minute period. In this period, we collected their usage data every 10 seconds, which was averaged at the end of the study. The results are presented in Table V.

As we can see from the table, when compared to popular apps, Screenmilker does not consume memory resources in any conspicuous way. Its memory usage is similar to those of the Clock and Calculator apps, at around 290 KB. Other popular apps such as the browser and Temple Run 2 need almost 50% more memory.

Finally, we ran PowerTutor [22] to examine the CPU and Wi-Fi power consumptions of each of these apps.[10] Our measurements were averaged over the data collected during a 5-minute period (once per 10 seconds), as illustrated in Table VI. Compared to those popular apps, Screenmilker consumed a very small amount of power. The situation component used only 4.1 mW, while the keystroke extraction component took 101.6 mW and the contact collection component consumed 8.3 mW. This level of power consumption is very moderate, in comparison with other apps, e.g., 155.8 mW for Instagram, 529.2 mW for Temple Run 2.

TABLE V. THE AVERAGE MEMORY USAGES OF SCREENMILKER COMPARED TO POPULAR APPS. SCREENMILKER USED RELATIVELY LOW AMOUNT OF MEMORY. WE EMPLOYED `PS` TO GATHER THE RESULT.

App	Memory [Kbyte]
Screenmilker [situation detection]	286.308
Clock	294.072
Screenmilker [contact collection]	295.279
Screenmilker [keystroke extraction]	295.364
Calculator	295.464
Google Talk	310.844
Instagram	326.244
Pandora Internet Radio	356.332
Facebook	365.384
Browser	391.912
Temple Run 2	436.712

TABLE VI. THE AVERAGE POWER USAGES OF SCREENMILKER COMPARED TO POPULAR APPS. SCREENMILKER USED VERY SMALL AMOUNT OF POWER. WE LEVERAGED POWERTUTOR TO MEASURE THE CPU AND WI-FI POWER CONSUMPTIONS OF EACH APP.

App	Power [mW]
Screenmilker [situation detection]	4.1
Screenmilker [contact collection]	8.3
Google Talk	47.8
Clock	52.1
Calculator	91.8
Screenmilker [keystroke extraction]	101.6
Instagram	155.8
Pandora Internet Radio	213.5
Browser	252.1
Facebook	374.8
Temple Run 2	529.2

V. MITIGATION AND SUGGESTIONS

The ADB workaround has become the standard way for third-party apps to acquire signature-level permissions. This approach, however, also brings in serious security risks: without proper regulation, it could expose signature-level system capabilities (illustrated in Table I) to unauthorized apps, as in the case of screenshot apps. Note that simply removing signature-level permissions from ADB or disallowing communications through local network sockets is completely im-

[10] Although PowerTutor is not able to measure accurate power usages on newer devices, it is still useful for comparing relative power consumptions which is sufficient for our purpose.

practical, since both ADB and local sockets are fundamental to Android design. ADB is the main Android development tool, which requires access to system resources protected by signature-level permissions to assist the developers in building apps. Local network socket is the major inter-process communication channel that native executables are able to utilize. In fact, many crucial system processes such as the Zygote process use local network sockets to communicate with others [14].

To mitigate this security risk, the app developers can certainly build authentication and access control into their apps, should they decide to utilize this ADB-based workaround, to prevent signature-level capabilities from getting into the wrong hand. However, given the importance of these capabilities, Android system-level protection should also be in place to ensure that security mediation on them will never be bypassed. As a first step toward this end, we describe here our preliminary design and implementation that controls the communication between the ADB proxy and a third-party app.

In addition, the design of Screenmilker demonstrates what a malicious party can do once it acquires a signature-level capability like screenshot. This indicates that any attempt to make such capabilities available to the third party (through Android APIs) should be well thought-out, an issue also discussed in the section.

Mediation of ADB resource exposure. A first step to prevent exposure of signature-level capabilities is to mediate the communication between the ADB proxy and its unprivileged app client. Although the security-enhanced Android version (SEAndroid [14]) can support such finer-grained access control, the prospect of its extensive deployment is still less clear, given the complexity in configuring its security policies, which impeded the wide adoption of SELinux it is built upon. In our research, we implemented a simple protection mechanism, which utilizes Linux `iptables` to control the interaction between a screenshot app's ADB proxy and its unprivileged component. Specifically, we set rules through `iptables` to control local-socket communication. An app by default is not allowed to make TCP connections to a server running on the same device. When a screenshot app is being installed on the device, it is supposed to explicitly ask for a new permission (which requires only a few lines of change to existing apps' manifests) for communicating with a local server through a specific port. Once the permission is granted by the user, a new service we built adds an `iptables` rule that binds the app's UID to a local TCP port specified, thereby allowing the app to talk to its ADB proxy. Other unauthorized apps still cannot make any local connections. In our research, we implemented this approach and evaluated its efficacy as well as efficiency. When configuring the system to simultaneously mediate up to 20 apps that require to access local TCP ports, we did not observe any malfunction and noticeable delay compared with the situation that the protection was not present. Although the performance of `iptables` is known to degrade as the number of rules increases, studies have shown that it is able to simultaneously support hundreds of apps [23], which is more than enough for most Android users. In addition, when adopting alternatives to `iptables` such as nf-HiPAC [24] or IP Sets [25], our design is able to mediate thousands of apps simultaneously. Note that this protection mechanism addresses not only the security risks in the screenshot apps, but also those that come with sync and backup, USB tethering apps and any other apps built upon the ADB workaround or insecure use of the local socket channel.

Interface suggestions. Given the strong demand for the programmatic screenshot capability (Section II), it is possible that Android might consider releasing an official interface to provide such a service in the future. Of course this capability is highly risky and needs a well thought-out protection to strike a careful balance between utility and user privacy. This effort can certainly benefit from the understanding of what Screenmilker can do. Here are some preliminary thoughts.

To avoid malware stealthily collecting confidential user data through the interface, Android should notify the user when an app takes a screenshot. This can be done through a system notification or visual and sound effects. Android can also issue different permissions to limit the screenshot rate, such as `LOW_RATE_SCREENSHOT` and `HIGH_RATE_SCREENSHOT`. Clearly, those allowed to take shots at a high speed are more likely to get access to the user's confidential information. Finally, Android can block the screenshot entirely when the user is working on confidential data. A way to do that is to let the user get into a privacy mode in which screenshots are not allowed. Also, an app being installed can also indicate that its operation should not be monitored, through its manifest.

VI. Related Work

Information leaks from smartphone have been extensively studied in prior research. As a prominent example, there is a line of prior work on how to infer sensitive user data through the accelerometer. Such work demonstrates what a malicious app can do once it obtains accelerometer readings. Examples include TouchLogger [26] and ACCessory [27] that work on keystroke inference, ACComplice [28] that identifies the trajectory of the user who is driving, and TapPrints [29] that recovers the positions on screen one taps from motion sensor data. A human subject study has also been reported to understand whether these attacks pose a credible threat to the smartphone users [30]. The side channels studied in these approaches all involve complex machine learning algorithms. Screenmilker, on the other hand, does not rely on such sensor-based side channels and is designed to work efficiently and stealthily on the phone.

Physical side channels such as oily residues left on the touch screen and reflection of the screen from nearby objects could also be exploited by attackers to extract private information from the victim's phone [31], [32]. Those attacks generally begin with the attacker taking a photo of the victim's phone screen, which is followed by image processing to analyze the fingerprints and infer confidential user information. They all require the attacker to be geographically close to the victim and able to take a photo of the victim's screen. The same constraint is also applied to the attacks like iSpy [33] that records a video of the victim's mobile device screen reflected from nearby objects such as victim's sunglasses, and automatically reconstructs the texts typed on the virtual keyboard of the victim's device through machine learning techniques.

Prior research also investigates how sensory malware with permissions to access camera or microphone can stealthily

collect user data and make a surprise use of it to get confidential user information. An example is Stealthy Video Capturer [34] that allows the attacker to automatically activate the built-in camera on 3G smartphones and take videos without victim's notice. It performs situation detection on Windows OS by explicitly claiming permissions to get access to related APIs. Another example, Soundcomber [35], utilizes microphones to record calls to interactive voice response systems, and then leverages voice recognition algorithms to retrieve credit card information. Compared to these prior approaches, Screenmilker only requires INTERNET permission and gets the screenshot capability from other apps. It processes the information collected and identifies its situations without using any other system permissions.

VII. Conclusion and Future Work

The ADB workaround has become the standard way to obtain signature-level permissions from an unrooted Android device. This workaround usually involves using local network sockets to establish communication between a native executable and a JVM-level app. Android, however, lacks access control on this channel. As a result, implementations of the workaround can oftentimes be exploited by an adversary to gain unauthorized signature-level permissions.

In this paper, we studied Android screenshot apps, a prominent example of the third-party apps that need signature-level permissions. We found that all the screenshot apps in Google Play employ the ADB workaround, and unfortunately, all of them fail to prevent unauthorized access from malicious apps. We further designed and implemented Screenmilker to demonstrate that through lightweight situation detection and data extraction, a malicious app can effectively and stealthily gather confidential information. To mitigate the security risk of the ADB workaround, we propose solutions to mediate the access to signature-level capabilities through ADB. In addition, we offer suggestions on the design of a secure programmatic screenshot API.

Our research is a first step towards understanding the security risks that could lead to exposing signature-level permissions. Further studies on this issue are certainly important. As examples, here we sketch two possible future directions.

Access policies for local network sockets. Local network socket is the main communication channel for many Android system processes (e.g., Zygote) and is the source for several notable vulnerabilities (e.g., CVE-2011-3918 and CVE-2012-2217). Thus, developing proper security policies for each process to access this resource and building a system to enforce these policies will greatly enhance the level of security for Android devices. These policies need to accommodate the requirements from different processes. For example, some processes ask for fast response time, while others demand high throughput. In addition, the enforcement system has to minimize its overhead. How to achieve these goals is a valuable research direction to pursue.

New screenshot APIs. Design of secure screenshot APIs is another important and challenging direction. Screenshot apps are extremely popular among Android users as demonstrated in Table II. Further development of these apps can be made easy once Android decides to provide screenshot APIs. However, those APIs, if used improperly, could pose threats to the privacy of the phone user. How to balance security and programmability is an essential question to be answered.

Protection of other communication channels. Although majority of the apps use local network sockets (a standard inter-process communication mechanism used in Android) to communicate with ADB proxies when demanding signature-level permissions, other communication channels may also be used for this purpose. For example, the app and the ADB proxy can read from and write to designated files to exchange requests and results. Due to the limited communication bandwidth, this approach probably will not be suitable for screenshot apps, but it is a viable alternative for backup and USB tethering apps. Since these additional communication channels pose the same threat to the Android security model as local network sockets do, strategies for effective meditation on them also needs to be investigated.

Countering stealthy attacks. One way to thwart private information extraction is to force the adversary to use more resources, making its malicious activities easier to detect by the user. Screenmilker presents a powerful yet lightweight attack that simple defense attempts such as randomizing the keyboard location or layout will not hinder the extraction process. More specifically, Screenmilker can employ the same strategy used for extracting contact list information to identify some features of the keyboard and then locate it on the screen, even when it is randomly placed. Thus, a more complicated defense mechanism is needed, e.g., a CAPTCHA-like procedure to obscure the keys. Developing such a system to make the lightweight attacks harder to succeed while still preserving the phone's usability is an important research direction.

Acknowledgements

We would like to thank anonymous reviewers for their insightful comments and valuable feedback. The authors with Indiana University are supported in part by the NSF CNS-1017782, CNS-1117106, and CNS-1223495. Hongyang Li is supported by Department of Energy Award Number DEOE0000097.

References

[1] Inside Mobile Apps, "Android reaches 25 billion app downloads, 675,000 total apps available," http://www.insidemobileapps.com/2012/09/26/android-reaches-25-billion-app-downloads-675000-total-apps-available/, 2012.

[2] The Next Web, "In one year, Android malware up 580%, 23 of the top 500 apps on Google Play deemed "high risk"," http://thenextweb.com/google/2012/10/25/in-one-year-android-malware-up-580-23-of-the-top-500-on-google-play-deemed-high-risk/, 2012.

[3] CVE Details, "CVE security vulnerability database," http://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html, 2013.

[4] F-Secure, "Q3 2012 mobile threat report," http://www.f-secure.com/weblog/archives/00002454.html, 2012.

[5] BullGuard, "The risk of rooting your Android phone," http://www.bullguard.com/bullguard-security-center/mobile-security/mobile-threats/android-rooting-risks.aspx, 2013.

[6] SmartUX, "Screenshot UX," https://play.google.com/store/apps/details?id=com.liveov.shotux, 2012.

[7] ClockworkMod, "How to get free tethering on any phone with ClockworkMod Tether (no root)," http://www.youtube.com/watch?v=R1Pc4RlieA0, 2013.

[8] E. Kim, ""no root screenshot it" for android," http://www.youtube.com/watch?v=zbbnu1JoyII, 2013.

[9] ClockworkMod, "Helium," https://play.google.com/store/apps/details?id=com.koushikdutta.backup, 2013.

[10] ——, "ClockworkMod Tether," https://play.google.com/store/apps/details?id=com.koushikdutta.tether, 2013.

[11] Wise Shark Software, "Screenshot Free," https://play.google.com/store/apps/details?id=com.androidscreenshotapptool.free, 2011.

[12] E. Kim, "No root screenshot it," https://play.google.com/store/apps/details?id=com.edwardkim.android.screenshotitfullnoroot, 2013.

[13] B. Levine, "Jelly bean now on 40% of android devices, google says," http://www.newsfactor.com/news/Jelly-Bean-on-40--of-Android-Devices/story.xhtml?story_id=00200072XCA0, 2013.

[14] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing flexible mac to android," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, ser. NDSS '13. San Diego, CA, USA: The Internet Society, 2013.

[15] Anonymous, "Screenmilker demo," https://sites.google.com/site/screenmilker, 2013.

[16] Android Development Team, "Permission elements," http://developer.android.com/guide/topics/manifest/permission-element.html, 2013.

[17] S. Garfinkel, "tcpflow," http://github.com/simsong/tcpflow/, 2013.

[18] jtschohl, "Android Firewall," https://play.google.com/store/apps/details?id=com.jtschohl.androidfirewall, 2013.

[19] Rodrigo ZR, "DroidWall," https://play.google.com/store/apps/details?id=com.googlecode.droidwall.free, 2011.

[20] S. Shekhar, M. Dietz, and D. S. Wallach, "AdSplit: separating smartphone advertising from applications," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security '12. Berkeley, CA, USA: USENIX Association, 2012, pp. 28–28.

[21] Nuance Communications, "Swype Keyboard," https://play.google.com/store/apps/details?id=com.nuance.swype.dtc, 2013.

[22] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the 8th IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES/ISSS '10. New York, NY, USA: ACM, 2010, pp. 105–114.

[23] J. Kadlecsik and G. Pasztor, "Netfilter performance testing," Netfilter Project, Berlin, Germany, Tech. Rep., 2004.

[24] M. Bellion and T. Heinz, "nf-hipac," http://www.hipac.org, 2005.

[25] J. Kadlecsik, "Ip sets," http://ipset.netfilter.org, 2013.

[26] 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*, ser. HotSec '11. Berkeley, CA, USA: USENIX Association, 2011, pp. 9–9.

[27] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang, "ACCessory: password inference using accelerometers on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems & Applications*, ser. HotMobile '12. New York, NY, USA: ACM, 2012, pp. 9:1–9:6.

[28] J. Han, E. Owusu, L. Nguyen, A. Perrig, and J. Zhang, "ACComplice: Location inference using accelerometers on smartphones," in *Proceedings of the 4th International Conference on Communication Systems and Networks*, ser. COMSNETS '12. New York, NY, USA: IEEE, 2012, pp. 1–9.

[29] 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*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 323–336.

[30] L. Cai and H. Chen, "On the practicality of motion based keystroke inference attack," in *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, ser. TRUST '12. Berlin, Heidelberg, Germany: Springer-Verlag, 2012, pp. 273–290.

[31] 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*, ser. WOOT '10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–7.

[32] Y. Zhang, P. Xia, J. Luo, Z. Ling, B. Liu, and X. Fu, "Fingerprint attack against touch-enabled devices," in *Proceedings of the 2nd ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '12. New York, NY, USA: ACM, 2012, pp. 57–68.

[33] R. Raguram, A. M. White, D. Goswami, F. Monrose, and J.-M. Frahm, "iSpy: automatic reconstruction of typed input from compromising reflections," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 527–536.

[34] 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 2nd ACM conference on Wireless network security*, ser. WiSec '09. New York, NY, USA: ACM, 2009, pp. 69–78.

[35] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, ser. NDSS '11. San Diego, CA, USA: The Internet Society, 2011, pp. 17–33.