Android Rooting: Methods, Detection, and Evasion

San-Tsai Sun
University of British Columbia
Vancouver, Canada
santsais@ece.ubc.ca

Andrea Cuadros[*]
Universitat PolitÃĺcnica de Catalunya
Barcelona, Spain
andreac@ece.ubc.ca

Konstantin Beznosov
University of British Columbia
Vancouver, Canada
beznosov@ece.ubc.ca

ABSTRACT

Android rooting enables device owners to freely customize their own devices and run useful apps that require root privileges. While useful, rooting weakens the security of Android devices and opens the door for malware to obtain privileged access easily. Thus, several rooting prevention mechanisms have been introduced by vendors, and sensitive or high-value mobile apps perform rooting detection to mitigate potential security exposures on rooted devices. However, there is a lack of understanding whether existing rooting prevention and detection methods are effective. To fill this knowledge gap, we studied existing Android rooting methods and performed manual and dynamic analysis on 182 selected apps, in order to identify current rooting detection methods and evaluate their effectiveness. Our results suggest that these methods are ineffective. We conclude that reliable methods for detecting rooting must come from integrity-protected kernels or trusted execution environments, which are difficult to bypass.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls, Invasive software

General Terms

Security

Keywords

Android Rooting; Privileged Malware

1. INTRODUCTION

Android is the most popular mobile operating system, accounting for about 80% of global mobile devices market in 2014 [25]. Android security controls restrict users and applications (a.k.a. "apps") from obtaining root privileges (i.e., full device administrative access), which could be abused or misused to compromise the security of the entire system. In order to freely explore or customize full system functionalities, many Android device owners voluntarily perform *rooting* to remove root access constraints placed by stock Android devices [27, 20]. Rooting is a process of allowing device users to attain a *persistent* privileged control (i.e., "root" access) to the device. The de-facto way to achieve this persistent root access is by installing a custom *su* binary (known as "switch user", "super user", or "substitute user") that allows any app on the device to perform privileged operations as root (details in Section 3). Once the device has been rooted (i.e., custom `su` installed), the device user can remove restrictions placed by carriers and hardware manufacturers, alter or remove system applications, run paid apps for free, or enjoy useful root apps (i.e., apps that requires root privileges), such as backup and restore, firewall, tethering, or full anti-malware functionality.

In this paper, by "rooting a device" we refer to the installation of a modified `su` binary on a device. This work focuses on the study of those apps that detect widely deployed `su` distributions installed voluntarily by the device owners. The detection and prevention of the malware that exploits system vulnerabilities, in order to obtain and persist root access, are out of this paper's scope.

Although useful to device owners, rooting weakens the security of Android devices. Without rooting, malware must exploit a system or kernel vulnerability present in the system in order to gain root access, which could be technically challenging. However, on a rooted device, any app could simply ask the user for root access with one-line of code (e.g., Runtime.exec("su")). The security of a rooted device relies solely on the device user regulating root access properly. Yet, the research shows that many users ignore security warnings due to habituation or lack of contextual information [23, 35]. A usability study of Android permissions also showed that most participants were not able to properly understand and grant Android permissions [24]. Once root access is inadvertently granted, malware could gain unauthorized access to any sensitive data stored on the device, intercept user inputs, tamper with runtime code (e.g., circumvent security controls, intercept file IO and network communication), and manipulate inter-app communications.

Rooted devices are prevalent. According to a recent Android security report [27], Google Verify Apps detected *rooting apps* (i.e., apps that root the devices via privilege-

[*] The author was visiting the University of British Columbia when this work was conducted.

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.

SPSM'15, October 12, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3819-6/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2808117.2808126.

escalation vulnerabilities) installed on approximately 2.5M devices, and particularly about 3-4% of Chinese devices have a rooting application installed. Note that the number does not include other rooting methods (e.g., unlockable bootloader, bootable SD card, OEM flash utilities). In addition, Verify Apps found that there are numerous applications from major Chinese corporations that include rooting exploits to provide functionality that is unavailable through the official Android API. Moreover, in China, 80% of cheap Android phones are shipped with customized system images that allow root access by default [20]. Furthermore, popular community-built system images (e.g., CyanogenMod [2]) and *root apps* (i.e., apps that require root access such as Titanium Backup [15], Root Explorer [12]) each has over 10 million downloads.

Several rooting prevention mechanisms (e.g., mount system partition with `nosuid` option [28], drop setuid call capability [28], SEAndrod [34]) have been introduced by Google in Android versions 4.3 and later. Yet, as we show in this paper, some rooting techniques bypass these countermeasures.

To reduce the risks of potential security exposures on rooted devices, mobile apps with sensitive or high-value functions therefore perform rooting detection and disable their functions, if the device appears to be rooted. Unfortunately, it is still common for the developers of root cloakers (i.e., apps that evade rooting detection) to engage into arms race with the rooting detection methods [22].

This raises an immediate concern regarding the effectiveness of rooting prevention and detection methods employed. Clearly, reliable rooting detection is desirable. There is, however, a lack of understanding of how various rooting methods work and what rooting detection methods could be difficult to evade. Our work aims to answer the following research questions to bridge this knowledge gap:

- RQ1: What methods are currently available for device users to perform Android rooting, and how do those rooting methods circumvent the rooting prevention mechanisms employed?
- RQ2 What are traits exhibited on a rooted Android device?
- RQ3: What methods are used by existing mobile apps to detect rooting traits?
- RQ4: How effective existing rooting detection methods are?

To answer these research questions, we first studied and tested existing Android rooting methods to understand how they work and what traits distinguish a rooted device. Next, we conducted an exploratory study by reverse-engineering 30 apps that contain rooting detection logic, to gain an overall understanding of existing root detection techniques. Manual code inspection, however, is pronged to human errors and hard to scale, because detection logic could be obfuscated or written in native code (C/C++). In addition, it would not allow us to evaluate the effectiveness of the detection techniques.

Informed by the insights obtained from the exploratory study, we developed a tool for dynamic analysis of rooting detection methods. Named *RDAnalyzer*, it hooks a set of APIs, both in Java and native code. For each hooked API, our analyzer logs the input parameters, and manipulates the output of the API attempting to evade the detection. We tested RDAnalyzer against apps examined in the exploratory study, and revised it whenever a new rooting detection method was uncovered. After this formative study, we conducted a confirmatory study by analyzing 152 apps downloaded from Google Play.

Our investigation of Android rooting methods found that existing mechanisms for preventing rooting can be circumvented by an advanced `su` daemon started at boot time. The `su` daemon could be installed through pre-boot rooting methods (e.g., unlockable bootloader, bootable SD card), and those rooting options would most likely remain available for device users, so long as "freedom of customization" is embraced by device vendors. In addition, we found that due to the confinement of privileged system daemons by SEAndroid, vendor-specific kernels and device drivers would be the next main targets of those rooting apps that rely on privilege-escalation vulnerabilities for rooting.

For rooting detection, we found a wide variety of techniques used by the studied apps. Besides obvious rooting traits (e.g., `su` binary, SuperSU app), various Java/C APIs and shell commands are leveraged by apps to find a wide range of rooting traits exhibited in different parts of the system. These rooting traits range from files and packages installed during and after rooting, to directory permissions, suspicious processes, background tasks, system properties, and even to the logo of a target app or over-the-air (OTA) update certificates.

Unfortunately, current methods for detecting rooting are not effective; they can be evaded, as we demonstrate with our RDAnalyzer. As such, existing root cloakers could be easily improved to evade all current rooting detection methods. Our study also suggests that the "arms race" between the developers of rooting detectors and cloakers is heavily asymmetric—the detectors are sandboxed while the cloakers are armed with root privileges. Ultimately, mobile apps need a reliable rooting detection method provided by the mobile OS. Based on the results of our investigation, we call for a reliable rooting detection API provided by Android OS.

To summarize, this work makes the following contributions:

- We studied Android rooting methods and developed their taxonomy that systemizes scattered knowledge about them. We also investigated how existing rooting prevention controls are circumvented.
- We conducted a first empirical study to understand existing rooting detection techniques on Android.
- We evaluated the effectiveness of the existing rooting detection techniques with a dynamic analysis tool that we built. Our results suggest that all rooting detection methods we found can be evaded.

The rest of the paper is organized as follows: We provide background and describe related work in Section 2. Section 3 presents the methodology and results of our investigation of rooting methods, and Section 4 discusses existing detection methods and evasion techniques. The implications of our findings and limitations are discussed in Section 5. We conclude in Section 6.

2. BACKGROUND AND RELATED WORK

Root access restrictions. Android security prevents users and apps from obtaining root privileges [36]. First, Android Open Source Project (AOSP) releases and stock Android devices allow only a small subset of core system services to run with root privileges. Second, each app installed on an Android device is constrained by Android's security sandbox. Upon installation, each app is assigned a unique Linux-level user id (uid), and its privileges are restricted within the uid-based process boundary through Linux's Discretionary Access Control (DAC). Thus, an app can only access resources within its own sandbox (i.e., app local storage) and cannot interact with other apps by default. To gain system resources access (e.g., GPS, network, contacts), apps must declare their required permissions which need to be granted by users during installation.[1] Note that the default `su` binary shipped with Android Engineer (*eng*) build allows only *root* and *shell* users to invoke it.

Persistent rooting mechanism. The de-facto mechanism for device users to obtain a *persistent* root access is by installing a custom `su` binary [19, 30, 32] that can be executed by all apps and commands on the device. When the custom `su` is invoked, it checks whether the calling app has been granted root access by the user via a root privilege management app [19, 30, 32] (denoted as "SuperSU" in this paper). If the access has been granted, `su` calls `setuid(0)` system call to switch the current process user to *root* and then executes the input command under root privileges. We found that, for Android versions 4.3 and later, due to the rooting prevention mechanisms introduced by Google, `su` does not call `setuid` system API anymore. Instead, `su` forwards the received command to an `su` daemon for execution (see Section 3.3.2 for more details).

Risks of rooting. Vidas et al. [36] survey privileged access attacks in the early Android platform (2011), including rooting. Our work presents the current state of rooting methods and analyzes how `su` has been advanced to circumvent current rooting prevention mechanisms.

Vulnerabilities in the `su` binary and SuperSU app could be exploited by malware to gain root access as well. Several implementation bugs in `su` were uncovered (CVE-2013-6774, CVE-2013-6775, CVE-2013-6770), which allow any app execute commands as root without a user permission [18]. In addition, Shao et al. [33] demonstrate that SuperSU's policy database and the local socket file could be attacked by malware to obtain root privileges. Moreover, Zhang et al. [38] show that privileged malware can retain its escalated permissions even after the user unroots the device. Furthermore, Zhou et al. [39] show that 37% of malware leverage root-level exploits to fully compromise Android.

Rooting Prevention and Protection. SEAndroid provides mandatory access control (MAC) to enforce sandboxing on Android system [34] and middleware [17] levels. Smalley et al. [34] demonstrate that SEAndroid is able to stop critical steps of a privilege-escalation exploit and prevent abusing root privileges even if system daemons are compromised. However, SEAndroid could be circumvented through pre-boot rooting methods or kernel vulnerabilities (e.g., futex by TowelRoot). According to 2014 Android security report [27], about 0.6% of devices had SELinux fully disabled and 0.3% of devices had SELinux configured in permissive mode. Our investigation uncovers how SEAndroid is circumvented (details in Section 3.3.2).

Verified Boot [29] establishes a chain of trust from the bootloader to the system image, and thus it is able to prevent unauthorized alterations to boot and system partitions (e.g., rooting). However, Verified Boot has not been deployed by most Android devices, as it requires changes to the OTA update mechanism (e.g., switching from file-level to block-level updates).

RootGuard [33] is an enhanced root-management system that provides fine-grain control (rather than "all" or "nothing") for users to grant the requested permissions to an app. The permission policy is based on system calls and parameters invoked by an app. Similarly to SE Android, RootGuard monitors system calls made by root apps to detect and prevent abnormal behavior of apps (i.e., malware) with root privileges. Nevertheless, as acknowledged by the authors, attackers might employ kernel rootkits or exploit kernel or device driver vulnerabilities to circumvent RootGuard.

Usability of root management. Current root-management model relies on users to grant root access to apps, but it's well known among usable security researchers that average users are either uncapable or unmotivated to consider carefully and make informed decisions about granting root privileges to an app. Many users ignore security warnings due to habituation or lack of contextual information [23, 35]. Felt et al. [24] show that only 3% of Internet savvy and 24% of laboratory study participants are able to successfully understand and grant permissions, while 42% of participants are unaware of permissions at all. Kelley et al. [31] performed twenty semi-structured interviews to explore Android users' feelings about and understanding of permissions, and their findings are aligned with the results of Felt et al. [24].

3. INVESTIGATING ROOTING METHODS

We started with a qualitative investigation of the methods employed for rooting Android devices and getting around various countermeasures that are supposed to make rooting difficult, if not impossible. We systemized our understanding in a form of a taxonomy of those methods.

3.1 Methodology

To understand what methods are available for device users to root their Android devices (RQ1) and what traits are exhibited on a rooted device (RQ2), we studied and tested existing Android rooting methods from various sources. In particular, we used XDA Developers website [37], a popular source for rooting related-information. We studied forums dedicated to particular models of Android devices, to understand various rooting-related topics, e.g., rooting tools, instructions, trouble-shooting, custom ROMs, drivers. We found that the information in those forums rich but not comprehensible, and sometimes, confusing. Further investigation revealed that there are typically multiple ways to obtain temporary root access for a given device, and there might be dependencies among those temporary root access mechanisms. In addition, for a given temporary root access mechanism, there are multiple rooting packages available in different forms. To build a complete view, we analyzed each rooting method found, and developed a taxonomy to clarify the relationships between various rooting-related components.

We also studied `su` source code and its over-the-air (OTA) update packages from three primary `su` distributions [19, 30,

[1] Please note that Android platform has been expanded in version *Android M* to include runtime permissions.

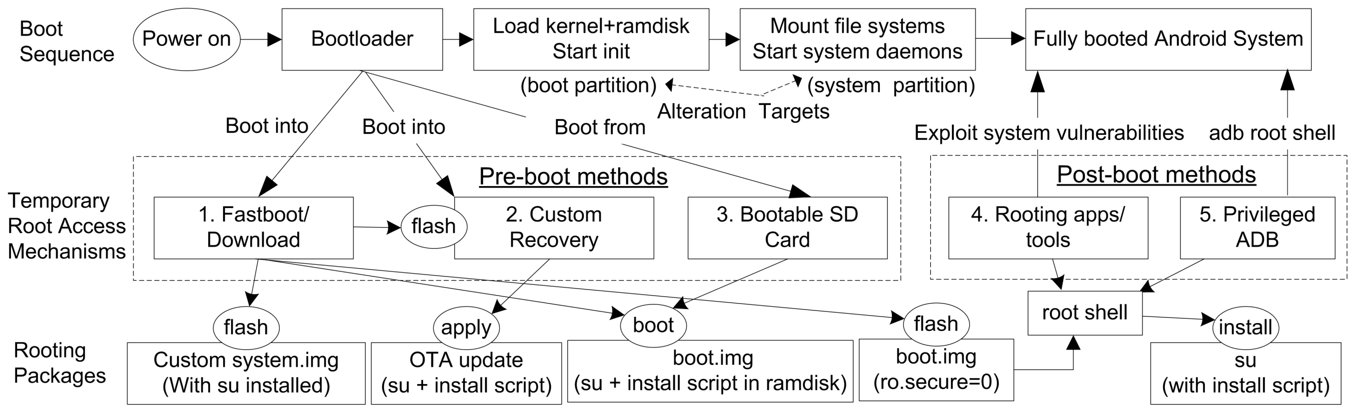

Figure 1: Taxonomy of Android Rooting Methods: The relationships between boot sequence, temporary root access mechanisms and su installation packages.

32]. This allowed us to understand how su works and how it overcomes the security protection mechanisms placed by Google. We found that only Chainfire's su distribution [30] works on devices with SEAndroid [34] in enforcing mode, but the distribution is closed source.

3.2 Taxonomy of Android Rooting Methods

The de-facto way for device users to attain a persistent root access is by installing a custom su binary that allows any app on the device to perform privileged operations as root. However, users cannot simply download or copy su over on to their devices. First, the system partition needs to be remounted as read-writable in order to install the su binary and its related files on to system folders (e.g., /system/xbin/). Second, any program that calls the setuid system call is required to have its SUID bit set (e.g., via chmod u+s command), which indicates to the operating system that the program is allowed to escalate its runtime privileges to root. Furthermore, for Android versions 4.3 and later, several system files need to be altered in order for the su daemon to be launched at boot time (further discussed in Section 3.3.2). All these operations require temporary root access. The ways of obtaining such access can be divided into two groups, depending on whether the access is gained during or after the boot process.

When an Android device is powered on and starts to boot, a low level, hardware specific program called "bootloader" is executed. The main purpose of a bootloader is to find and start the Android OS, or alternatively, the recovery OS. Many Android devices (e.g., Nexus, HTC, Motorola XOOM, Android One, Samsung) are shipped with an "unlockable" bootloader, which allows device users to flash (i.e., write raw partition image to the device's persistent storage) custom *system*, *recovery* or *boot* images, or boot from transient *boot* or *recovery* images, without flashing them to the device. On the other hand, some Android devices do not have an unlockable bootloader. Only those boot and recovery images signed by the manufacturer can be loaded by locked bootloaders.

As illustrated in Figure 1, we have identified five temporary root access mechanisms that device users could use to install a custom su binary on their devices. Based on whether the bootloader is unlockable or not, they are categorized into "pre-boot" or "post-boot" groups. Methods in "pre-boot" group, require making the device's bootloader to boot into (1) a special bootloader mode or (2) a custom recovery OS, or (3) boot from an external bootable SD card. Alternatively, "post-boot" rooting methods are employed after the device has been fully booted, and all of them require first to obtain a root shell. They do so through either (4) an exploitation of privilege-escalation vulnerabilities, or (5) a privileged Android Debug Bridge (ADB).

Depending on which temporary root access mechanism is used, there are different types of su packages available for device users to facilitate the process of installing and configuring su (e.g., OTA update package, custom system or boot images, bundled with the rooting apps). The main alteration targets of rooting methods are the boot partition (e.g., alter configurations in the RAM disk) and the system partition (i.e., install su). We describe existing rooting methods in the rest of this section.

1. Fastboot/Download mode: An unlockable bootloader typically supports a special mode (e.g., *fastboot* [5] or *download* [10] mode) that allows users to unlock the bootloader and flash custom images using tools such as fastboot [5] or ODIN [10]/heimdall [6] (for Samsung devices), through a USB connection from a host machine. For security reasons, "unlocking" a bootloader will typically trigger device factory reset (i.e., reformat the *userdata* and *cache* partitions) to ensure that a malicious OS image cannot get access to the existing user data. Once the bootloader has been unlocked, there are several ways to install su:

Replace system partition: One easy way to enable persistent root access is to flash a custom *system* image that has su already installed on it (e.g., CyanogenMod [2]). Alternatively, if the user wants to use the existing system image on the device, he/she could simply pull the stock system image from the device (e.g., using dd command) onto a host machine, install and configure su, repackage it, and then flash the new system image back to the device.

Alter boot partition to enable root shell: The user id under which an ADB shell runs on the device depends on the value of system property ro.secure. If ro.secure equals 0, an ADB shell will run as *root*; otherwise, the shell will run as a non-privileged user (e.g., *shell*). The value of this property is set at boot time, based on the `default.prop` file located in the boot partition. By unpacking and repacking a boot image with `ro.secure=0`, one can make the ADB daemon run as root and enable root access via a root shell.

Boot from a transient boot image: An Android boot image consists of a Linux kernel and a RAM disk. The RAM disk is a small partition image that contains a root filesystem (i.e., `rootfs`) mounted as read-only by the kernel at boot time. Once the `rootfs` is mounted, the first Linux process `init` will be started to mount the rest of file systems from the devices, and then perform initialization procedures based on the configuration files (e.g., `init.rc`). By creating a boot image that contains customized commands and scripts, one can therefore install `su` into the system partition, or even flash custom system images, by booting the device from a custom boot image.

Flash a custom recovery OS: Recovery OS is a minimal Linux OS that consists of a Linux kernel, a RAM disk with various low-level executables and configuration files, and a recovery program. A stock recovery OS is used to apply post-ship system updates delivered in a form of over-the-air (OTA) update packages. OTA packages include updated system files and a script that applies the updates. Stock recovery OS typically only allows applying OTA update packages signed by the device manufacturer. Nevertheless, since the recovery OS is stored in a partition similar to `system` and `data` partitions, device users can replace it with a custom recovery OS, once a temporary root access has been obtained. All existing `su` distributions provide installation packages in a form of OTA update packages.

2. Custom recovery. A popular and flexible way to install `su` is via an OTA update package applied by a custom recovery OS (e.g., ClockworkMod Recovery [1], Team Win Recovery Project [14]). An OTA update package can add or alter system files without replacing the entire system image, and thus the installed `su` can coexist with the stock OS on the device. Note that the updater script for the OTA update is usually written in a special `edify` script language [26]. However, we found that all `su` OTA packages use traditional UNIX shell, so that the same installation script can be reused by other rooting methods (e.g., bootable SD card, privileged ADB, one-click rooting apps). A snippet of `su` installation script from Chainfire's distribution [30] is shown below.

```
mount -o rw,remount /system /* remount system as  writable */
---snip--- /* place files and set permissions*/
cp_perm 0 0 06755 <src>/su /system/xbin/su
cp_perm 0 0 0755 <src>/su /system/xbin/daemonsu
cp_perm 0 0 0644 <src>/Superuser.apk /system/app/Superuser.apk
cp_perm 0 0 0755 <src>/supolicy /system/xbin/supolicy
cp_perm 0 0 0755 <src>/install-recovery.sh
                        /system/etc/install-recovery.sh
/* content of  install-recovery.sh */
#!/system/bin/sh
/system/xbin/daemonsu --auto-daemon &
```

The update script first mounts the system and data partitions in read-write mode, and then copies the included files to their intended locations in the file system via the `cp_perm` helper function. Function `cp_perm` is a shell function that copies (`cp`) a source file to a target destination, and then (1) changes the owner (`chown`) to root, (2) sets permissions (`chmod`), including `SUID` bit of `su`, and (3) configures SELinux security label of the target file (labeled as `system_file`). Note that in addition to `su` and SuperSU app, a script called by *init* to install OTA updates (i.e., `install-recovery.sh`) is overwritten by the update script, which allows `daemonsu` to be invoked by `init` at boot time. Furthermore, the tool (`supolicy`) used by `daemonsu` to patch SELinuc policy at boot time is copied as well (details are in Section 3.3.2).

3. Bootable SD Card: Bootloaders of some devices (e.g., Nook Color, WonderMedia tablets) support booting from an external bootable SD card. By creating a bootable SD card that contains a customized RAM disk (e.g., accompanied with the `su` binary and an update script similar to the one shown above), one could install `su` into the system partition by booting the device from the SD card.

4. Rooting apps or tools. For devices that do not have an unlockable bootloader, root access can be obtained by exploiting a system or kernel vulnerability. A privilege escalation exploit, typically packaged into "one-click" rooting apps or scripts, allows an app or script to start a root shell to install `su` or modify system configurations. Many popular one-click rooting tools exploit one or more vulnerabilities found in the kernel (e.g., futex by TowelRoot, mem_wrtite by Mempodroid), device drivers (e.g., PowerVR SGX driver by levitator, Qualcomm diagnostic driver by diagroot), or system daemons (e.g., adbd by RAtC and bin4ry, zygote by Zimperlich, init by psneuter and KINTO, vold by Zergrush and GingerBreak) to install `su` for a persistent root access.

5. Privileged ADB. The `ro.secure` system property configured in `default.prop` determines the process uid under which an ADB shell is running. When the value is set to 1 (i.e., secure mode), the `adbd` daemon process, which initially runs as root, drops all capabilities from its capability bounding set, except CAP_SETUID and CAP_SETGID, and then changes its UID to AID_SHELL (UID=2000) before spawning an ADB shell. Certain manufacturers, and Android engineering (`eng`) builds, do not set `ro.secure` to secure mode, which allows users to have a shell that can execute any program as root. With a root shell from ADB, one can upload `su` and its installation script to a temporary folder on the device, and then execute the script from that folder to persist `su`.

3.3 Privileged Operations via su

Once `su` and SuperSU app have been properly installed, any app on the device can invoke `su` to run arbitrary command with root privileges. In this section, we discuss how `su` is used by apps to perform privileged operations. This knowledge allows us to analyze rooting detection methods currently employed by various apps. It could also shed light on future rooting detection and prevention techniques.

3.3.1 Android OS v4.2 and Earlier

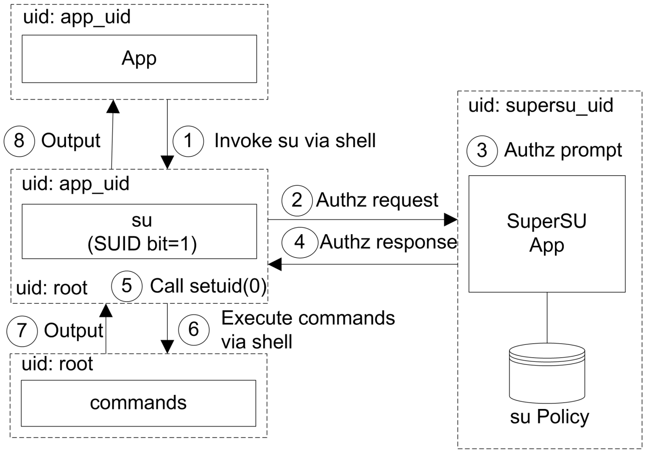

Figure 2: The sequence of invoking su by an app to perform privileged operations on Android versions 4.2 and earlier. Dashed boxes denote process boundaries.

Figure 2 illustrates the data flow when `su` is invoked by an app in Android versions 4.2 and earlier:

1. An app invokes `su` binary through a shell. The input command can be provided to `su` in the command line directly (e.g., `su -c cmd`), or through standard input piped from the calling app. Note that the app process is always running under a unique uid assigned during installation of the app.
2. `su` performs a permission check by sending an authorization request. It starts with creating a local socket under the *SuperSU's* local folder, and sets the permissions on the socket so that only *SuperSU* is allowed to access it. Then, `su` sends an *intent* message to *SuperSU*, containing the path to this socket. Once *SuperSU* connects to this socket, `su` uses it to send an authorization request that contains the package name and UID of the requesting app.
3. *SuperSU* checks its policy database to determine whether or not the requesting app has been granted root access by the user. If the root permission has not been granted, *SuperSU* prompts the user for granting root permission to the app.
4. *SuperSU* returns to `su` the authorization decision, either granting or denying the request.
5. If the root permission is granted, `su` calls `setuid(0)` and `setgid(0)` to switch the uid of the current process from the requesting app's uid to *root*.
6. `su` forks a new shell to execute the input command under *root* uid.

7 & 8. The output of the executed command is returned back to the app through a standard output.

3.3.2 Android OS v4.3 and Later

For Android versions 4.3 and later, several security enhancements have been introduced to disallow apps from executing, or installing, the `su` binary. The security enhancements include (1) mounting `system` partition with `nosuid` option, which disables programs located on that partition to call setuid API [28], (2) dropping all capabilities, including `call_setuid`, from the capability bounding set for all Android apps forked by the *zygote* process [28], and (3) confining privileged processes with SEAndroid [34] mandatory access control (MAC) policy.

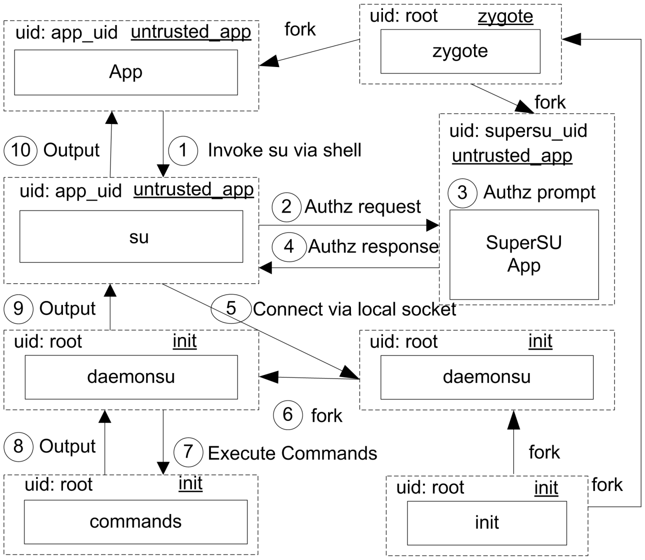

Figure 3: The sequence of invoking su by an app to perform privileged operations on Android versions 4.3 and later. Dashed boxes denote process boundaries, and underlined labels denote security context (domain).

Under SEAndroid (Android port of SELinux), each process in the system is labeled with a security context (called *domain*). When a process tries to perform an action on a resource (called *object* in SEAndroid terminology), an object manager in kernel intercepts the requested action, and checks the request against a centrally managed MAC policy, in order to determine whether the request should be allowed or not. Thus, on Android versions with SELinux in enforcing mode, executing a process with root UID does not change its security context, and such a process is still constrained by the SEAndroid MAC policy.

Due to these new security constraints, persisting root access is not as simple as copying an SUID binary to the device. Although replacing the boot or system partitions allows these security controls to be disabled and any security mitigation to be reverted, such a radical approach is not desirable by the `su` developers. Thus, `su` has been re-designed to coexist with the new stock Android systems.

As illustrated in Figure 3, the original `su` has been separated into an `su` and a su daemon (denoted as `daemonsu`). The `su` binary does not call `setuid` API any more. Instead, it forwards all the requests received from apps to `daemonsu`, which runs as a daemon process with *root* uid and *init* security context. To enable `daemonsu` running under *init* domain, the `su` installation script modifies the system in such a way that the `daemonsu` is invoked by *init* so that it can inherit the domain of *init* (e.g., modify `/system/etc/install-recovery.sh`, symlink `/system/bin/app_process` to `/system/xbin/daemonsu`, add an init script to `/system/etc/init.d/`). In addition, `daemonsu` patches the MAC policy during boot time to unconfine the *init* domain and to allow su to connect to the local socket created by the daemonsu. Note that the new su architecture is backward compatible, and thus existing root apps can work with both versions of su without any modifications.

The data flow of an app performing privileged operations via su in v4.3 and later is similar to the flow of v.4.2 and earlier (see Figure 2). The main difference is in how the domains of related processes are labeled. At boot time, the first process, *init*, sets its own domain as "*init*". When *init* launches *zygote* process, *zygote*'s security context is labeled automatically, according to the automatic domain transition rule defined in the policy (e.g., type_transition init zygote_exec:process zygote), and the type of *zygote* executable labeled during the system build process (e.g., labeled as zygote_exec). All Android apps are forked by *zygote* process, and when an app is launched, *zygote* assigns the app to *untrusted_app* domain, based on the app's UID assigned during installation and a system configuration file (i.e., seapp_contexts). Note that when su is invoked by an app, su inherits the domain of the app (i.e., *untrusted_app*). su then connects to a local socket of daemonsu. Upon receiving the request, daemonsu forks a new process and then executes the input commands under *root* uid and *init* security context.

In this section, we presented our findings on various rooting methods available for users, and how apps perform privileged operation through su. These technical details are essential for apps to detect whether a device has been rooted. In the following section, we discuss what methods are used by current apps to perform rooting detection, and how those detection techniques can be evaded.

4. DETECTION AND EVASION

Rooting traits are unique properties exhibited on a rooted Android device. Depending on the rooting method employed for rooting (e.g., OTA update packages, rooting apps), and what root apps or tools have been installed after rooting, different traits will be exhibited. Thus, some rooting traits are present on all rooted devices (e.g., su binary, *SuperSU* app), while others only on some. In addition, rooting traits could be present in different parts of the system (e.g., file system, process, package manager), and could be detected via different API calls and commands. Rooting detection on Android devices is achieved today by examining different parts of the system and checking whether one (or more) of these rooting traits can be found. When one of these rooting traits is detected, it is an indication that the device has been rooted.

4.1 Methodology

To evaluate the effectiveness of existing rooting detection methods, we conducted three empirical studies. Specifically, these studies helped us to understand what detection methods used by existing mobile apps (RQ3), and whether and how those detection methods can be evaded (RQ4).

4.1.1 Exploratory Study

As a starting point, we conducted an exploratory study by reverse-engineering 30 apps that contain rooting detection logic. The list of apps was obtained from a popular root cloaker website [22], which claims that those apps had been "successfully tested". For each app, we downloaded the app's package (APK), converted the APK into JAR using dex2jar [4], and then decompiled the JAR using both JAD [8] and JD-GUI [9]. We manually analyzed the source code to identify the root detection methods employed by these apps.

We observed that rooting detection functions are commonly called from the *launch* activity of the app. For each rooting detection method identified, the API used by the app and its input parameters were documented. We found four apps that perform rooting detection in native code, instead of Java, and three other apps had their code obfuscated. We kept these apps for dynamic analysis in the formative study, but did not attempt to reverse-engineer them any further, as no new rooting detection methods were found after first 20 apps.

Manual analysis of the source code was valuable, allowing us to understand what APIs and parameters are used by apps to detect rooting traits. Manual analysis, however, is neither effective nor scalable due to the following constraints:

- Manual source code inspection is a time-consuming and error-prone process (e.g., miss or mis-identifies a rooting detection method).
- Java code can be obfuscated before distribution, making it difficult to analyze.
- Native code (C/C++) used for rooting detection is compiled into machine code when the app is distributed.
- The effectiveness of the identified rooting method could not be evaluated.

4.1.2 Formative Study

To overcome the constraints of manual analysis, we developed a tool, named "RDAnalyzer", for semi-automatic dynamic analysis of rooting detection methods employed by apps. RDAnalyzer consists of two parts. The first part provides user interface for configuration settings, such as the list of targeted apps (specified in package name) and the blacklisted rooting traits (e.g., file paths, package names, commands, keywords). The second part is an API hooking module that intercepts a set of API calls, both in Java and native code. The list of hooked APIs and input parameters were initially gathered from the exploratory study and were revised throughout the confirmatory study (described in the next section). The hooking module was implemented as an extension of Cydia Substrate [3], a popular userland API hooking framework for both Android and iOS. Substrate injects our hooking module into each app when the app process is forked by the *zygote* process.

We first deployed RDAnalyzer to dynamically analyze apps that were examined in the exploratory study. For each API hooked at runtime, RDAnalyzer logs the input parameters and manipulates the output of the API to evade detection. For instance, RDAnalyzer returns false when the app checks the existence of a file (e.g., /system/xbin/su), removes rooting traits from the output of an API call, or throws an exception when a targeted shell command (e.g., "which su") is executed. Whenever RDAnalyzer failed to evade a detection method, we reverse-engineered the corresponding app to understand which APIs and input parameters should be considered, and then revised RDAnalyzer to evade the newly discovered detection technique(s). To confirm whether RDAnalyzer has successfully evaded an app's root detection, we inspect that app manually (e.g., message shows that "the device is not rooted", functions prohibited to run on a rooted device is enabled when RDAnalyzer is activated).

4.1.3 Confirmatory Study

By applying RDAnalyzer to a sample of apps, we collected data on the use of various detection techniques in the wild. During this study, whenever we uncovered previously unknown detection techniques, we revised RDAnalyzer accordingly, and re-analyzed previously analyzed apps from the sample. For each app, we recorded the detection techniques it employed. The primary objective of this study was to obtain descriptive statistics about the use of rooting detection techniques in the wild.

To facilitate the collection of the sample apps in systematic and reproducible way, we developed a tool, named "APK downloader", that automatically downloads apps from Google Play based on query keywords. We selected apps that (a) most likely have rooting detection logic in place, such as those that can be found on Google Play by searching for the query strings on free apps "root check" and "root verify" with APK Downloader and (b) popular apps that require the device to be rooted (e.g., backup, file manager) that could be found in Google Play and third party app stores.

Overall, we selected 242 apps, out of which, we excluded apps (1) whose description mentioned specifically that no root is required, (2) with UI in a language other than English or Spanish, (3) that did not work properly (e.g., the app could not be opened due to an error), (4) in which "root", "check" or "verify" in the description has a different meaning than Android rooting (e.g., "math square root"), (5) that are designed for specific device models (e.g., Samsung Galaxy only), (6) with low popularity (i.e., less than 1,000 downloads), and (7) that we considered not to be valid for analysis as we used them for rooting purposes (e.g., Busybox and SuperSU). Overall, we excluded 45 apps. At the end, 152 apps were selected for analysis in the confirmatory study.

We first downloaded all the apps' apk and meta data (e.g., app description, popularity in number of downloads, category) for a certain query using APK Downloader. We then proceeded to install all the apps on a rooted Nexus 7 device with Android version 4.2.1. We employed the following procedure for the analysis of each app:

- Without activating RDAnalyzer, we launched one-by-one all the installed apps from the tablet. For each app we inspected whether it detects that the device has been rooted. We did this by (1) looking for visual indications that could appear on the screen (e.g., `su` permission prompt, message shows that "the device is rooted" or "the app is prohibited to run on a rooted device"), and (2) testing app features that need root and verifying if they work or not.
- We then activated RDAnalyzer. We again launched one-by-one all the apps and looked for the indications mentioned previously to confirm whether they detect that the device has been rooted. We also obtained the log generated by RDAnalyzer to classify the rooting detection techniques each app used.
- If the analyzed app surpassed RDAnalyzer, we first installed the app in an unrooted Nexus S with Android version 4.1.2 to check whether the detection was a false positive (i.e., the app reports rooting even when the device is not rooted). In the case it is not a false positive (which was the case for all the examined apps), we reverse-engineered the app to find new detection methods. We then revised RDAnalyzer and tested the app again, until all rooting detection methods employed by the app were evaded successfully.

4.2 Results: Detection Methods

The results of the uncovered rooting detection methods are presented in this section. Using RDAnalyzer, all of the rooting detection methods we discovered can be evaded. Figure 4 illustrates the distribution of detection methods used by the apps in the confirmatory study, ordered by download popularity from most (left) to least (right). Note that D4 (Check System Properties) was identified in the exploratory study, but none of apps in the confirmatory study used this detection method.

In the rest of this section, we discuss each of the seven detection techniques and the ways we were able to evade them.

4.2.1 D1: Check Installed Packages

Certain apps packages are commonly installed during or after rooting, for instance:

1. SuperSU app: An app that installed along with the `su` binary for users to regulate root access [19, 30, 32].
2. Rooting apps: Apps that exploit privilege-escalation vulnerabilities to root the device (e.g., One Click Root, iRoot, Root Genius).
3. Root apps: Apps that require root privileges for their functions, such as BusyBox, SetCPU, Titanium Backup, Wireless Tether, Adfree, ShootMe.
4. Root cloakers: Apps that hide the fact the device has been rooted (e.g., Root Cloaker [22], Root Cloaker Plus [21]).
5. API hooking frameworks: Libraries that provide API hooking functions (e.g., Cydia Substrate [3], Xopsed Framework [16]).

`PackageManager` is an Android Java class for retrieving various kinds of information related to installed application packages. This class is commonly used by the studied apps to check whether a specific app package that requires root privileges is installed on the device. For instance, `getInstalledPackages` or `getInstalledApplications` return a list of all packages/applications that are installed on the device. The detecting app then iterates through the results to see whether a specific package is presented. Other common used APIs are `getPackageInfo` and `getApplicationInfo` which retrieves overall information about an install package/application.

We found that some PackageManager's methods do not directly return an application/package information, but they are leveraged by apps to infer whether a given package is installed or not. For example, `getApplicationLogo` retrieves the logo icon associated with an application, `getLaunchIntentForPackage` returns an intent to launch a front-door activity in a package, and `getPackageGids` returns an array of all of the secondary group-ids that have been assigned to a package.

Evasion: For APIs that return a list of packages or applications, RDAnalyzer calls the hooked function and then removes the target package names from the result before returning them to the app. For APIs that retrieve specific information for an given package, RDAnalyzer returns *null* directly to falsely indicate to the app that the package does not exist.

4.2.2 D2: Check Installed Files

The `File.exists` Java method, or `access`, `open` C APIs can be used to check the existence of a file in the file system. Commonly checked files include the follows:

- /system/xbin/su, system/bin/su. We found that some apps manipulate the path attempting to confuse evasion (e.g., /system/xbin/../xbin/su).
- /system/xbin/busybox and all symbolic links of commands created by busybox.
- /data/app/<APK name> or /system/app/<APK name> of popular apps packages that are installed during or after rooting.

We found that some apps check APK files under /data/data/ directory, however, access to this directory are restricted by default for user installed apps.

Evasion: When the target file path is detected in the input parameter, RDAnalyzer calls the original hooked function with a fake file path that certainly does not exist (e.g., /system/xbin/doesnotexist) as the input parameter. While `openat()` and other related functions can be used to remove full path names from the executable and refer to the file by a file handle, RDAnalyzer or root cloakers could find the full path of the file, given the file handle.

4.2.3 D3: Check the BUILD tag

By default, stock Android images from Google are built with “`release-keys`” tag. If “`test-keys`” are presented, this can mean that the Android image on the device is either a developer build or an unofficial build. The follow code is used by apps to test whether the Android image on the device is an official build or not.

```
if (!android.os.Build.TAGS.equals(‘‘release-keys’’)) {//rooted}
```

Evasion: We examined the source code of Build.TAGS and found that this build tags information is a class-level `static final` string retried from “`ro.build.tags`” system property through `System.getProperty` method. To mask this value, RDAnalyzer uses Java reflection APIs to change it to “release-keys”.

```
Class _class = Class.forName("android.os.Build");
Field field= _class.getDeclaredField("TAGS");
field.set(null, "release-keys");
```

4.2.4 D4: Check System Properties

If the value of system property `ro.secure` equals “0”, the adb shell will be running as *root* instead of *shell* user. `System.getProperty` Java API can be used by apps to examine this property value. We found that some apps also check if “`ro.debuggable=1`” and “`service.adb.root=1`”. Examining ADB source code reveals that the adbd daemon will be also running as *root* user if both properties are set to one.

Evasion: RDAnalyzer hooks `System.getProperty` and returns “‘1” for `ro.secure` property, and “0” for `ro.debuggable` and `service.adb.root` properties.

4.2.5 D5: Check Directory Permissions

Some rooting tools make certain root folders readable (e.g., /data) or writable (e.g., /etc, /system/xbin, /system, /proc, /vendor/bin) to any process on the device. The `File.canRead` and `canWrite` Java APIs, or `access` C API can be used to check whether such condition exists.

Evasion: If the target path is presented in the input parameter, RDAnalyzer simply returns false (e.g., not readable or not writable) to the app.

4.2.6 D6: Check Processes/Services/Tasks

The `ActivityManager.getRunningAppProcesses` method returns a list of currently running application processes. This API is used by studied apps to check whether a specific app that requires root privileges is running on the device. Similarly, `getRunningServices` or `getRecentTasks` APIs are also used by apps to retrieve a list of current running services or tasks respectively to check whether SuperSU and popular apps for rooted devices are running on the device.

Evasion: RDAnalyzer calls the hooked function and then removes the target names from the result before returning them to the app.

4.2.7 D7: Check Rooting Traits Using Shell Commands

Shell commands are commonly used by studied apps to detect aforementioned rooting traits. Apps can use `Runtime.exec` Java API, `ProcessBuilder` Java class, or `execve` C API to execute a specified command in a separate process, and then examine the output of the shell command to detect rooting traits. Commonly employed shell commands are as follows:

- `su`: If the `su` binary exists, this shell command will exit without error; otherwise, an IOException will be thrown to the calling app.
- `which su`: The which command outputs the full path of a specified command; in this case, the `su` command (e.g., /system/xbin/su).
- `ps | grep <target>`: The ps command outputs a list of the current running processes. The output of ps can be piped through a `grep` command to search specific app processes that requires root access (e.g., daemonsu, SuperSu).
- `ls - <target>`: ls command can be used to check the existence of a file in the file system.
- `pm list packages`: List all installed packages using the `pm` command to search for specific targets.
- `pm path <package>`: Output the full path of the targeted package.
- `cat /system/build.prop | grep ro.build.tags`: Check whether ro.build.tags=release-keys

Evasion: RDAnalyzer (1) throws an IO exception if the input command involves `su`, or (2) removes target names from the command output by piping the input command through an invert-match `grep` command (e.g., `| grep -v "su"`), so that only non-matching lines are returned to the calling app.

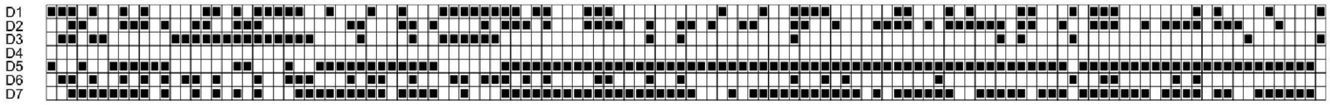

Figure 4: The distribution of rooting detection methods used by apps in the confirmatory study, ordered by download popularity from most (left) to least.

4.3 Results: Detection Apps

In this section, we report on the results of our confirmatory study. They are summarized in Figures 4 and 5. Our analysis of rooting detection apps revealed several interesting aspects.

First of all, while we identified D4 in our exploratory study, none of the apps we analyzed in the confirmatory study employed this detection method. The popularity of root detection apps did not correlate with the number of the detection techniques they used, as can be seen from Figure 4. Almost 2/3 of the analyzed apps employed only 2-3 detection methods. Figure 6 illustrates the popularity of the detection methods.

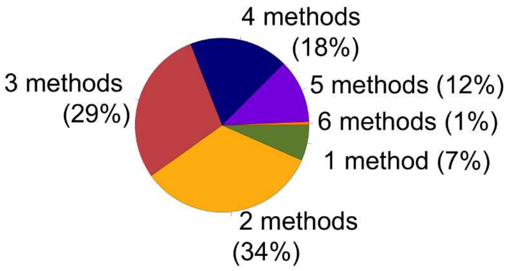

Figure 5: The employment of detection methods by apps. It shows how many methods the analyzed apps employed.

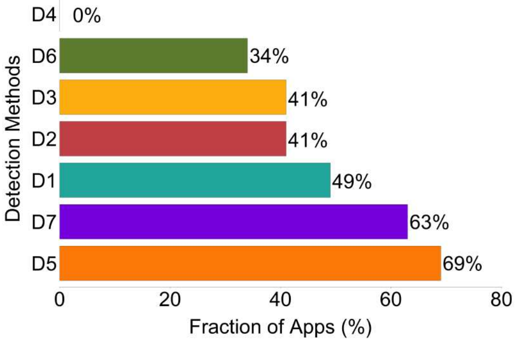

Figure 6: The popularity of detection methods. Y-axes is the percentage of the analyzed apps, which employed the method in question.

5. DISCUSSION

Pre-boot rooting v.s. freedom of customization. Despite the effort of security enhancements for preventing rooting, pre-boot rooting methods (unlockable bootloaders in particular) would probably remain available for device users to customize their devices. This is because one of Android's biggest selling points has always been its "freedom of customization", and installing a third-party Android build or useful root apps are attractive options for user. Besides customizing the system, an unlockable bootloader may even be the only way to run an updated version of Android on devices that have stopped receiving updates from device manufacturers. This is probably the reason why most recent devices provide a way for users to unlock the bootloader and install custom builds or recovery OS. Nevertheless, from a mobile app's perspective, an unlocked bootloader is a strong risk indicator. Once a bootloader is unlocked, all bets are off, the entire computing base of the device cannot be trusted anymore. Thus, vendors and manufacturers should coordinate efforts to provide a unified way for mobile apps to query the security state of a bootloader. In addition, records of modifications made to the boot and system partitions should be provided as well. It could be an interesting subject of future research to investigate whether and how remote attestation with a strong hardware root of trust and other techniques could be employed for supporting detection of unlocked bootloader or even rooted device.

Impacts of SEAndroid on the targets of rooting apps. Rooting apps need to exploit privilege-escalation vulnerabilities found in system daemons, kernel, or device drivers in order to persist root access. While SEAndroid could be circumvented by pre-boot methods, it can make rooting apps difficult to exploit privilege-escalation vulnerabilities found in system daemons. In addition, Linux kernel is compact and gradually stablized, and hence discovering new kernel vulnerabilities would be a challenging task, if not impossible. Therefore, vendor-specific kernels and device drivers would become the main targets of rooting apps in the future, as they are implemented by a single party and mostly closed source, without undergoing rigid security review.

Asymmetric arms-race for rooting detectors. Our study shows that the arms race between rooting detectors and evaders is an asymmetric competition that favors evaders. While detectors are sandboxed by Android security model, evaders are armed with root privileges that have full control over the device and, in theory, can bypass any security protection mechanism placed by vendors or manufacturers. Even if detectors find new rooting traits or employ advanced detection techniques, evaders could always advance their techniques to evade the detection methods. Therefore, ultimately, mobile apps need a reliable rooting detection API provided by Android OS, with its implementation from a trusted source, likely relying on hardware root of trust, e.g., integrity-protected kernels or external trusted execution environments. How such an API can certify correctness of the result is a subject of future research.

5.1 Additional Observations

During our studies, we also observed two new detection techniques that could complicate current root cloakers. These new techniques were tested against two popular root cloakers [22, 21].

Exception Call Stack Inspection. We found that Java exception call stack can be leveraged by apps to infer whether the device has been rooted. When performing rooting detection using a shell command via `Runtime.exec("su")` or `ProcessBuilder("su")`, a Java IO exception will be thrown to the app if the `su` binary does not exist. We found that when the API has been hooked, the call stack will be different. That is, when the API has been hooked, the hooking function will be presented on the top of the call stack, instead of the hooked API (e.g., Runtime.exec). Thus, apps could determine whether the API call has been hooked by checking the function name on the top of an IO exception call stack. If the API has been hooked, the device must be rooted because API hooking requires root privilege. To evade this detection method, the root cloakers need to tamper with Java runtime in order to hide its trace left on the IO exception call stack.

Stateful rooting detection. We found that existing rooting detection methods are stateless (i.e., detection is solely based on the return value of a single API call), and therefore they can be evaded easily. Even when a new rooting trait has been discovered and used in detection, due to the simplistic nature of stateless detection mechanism, it is trivial for root cloakers to evade it (e.g., by adding the new traits to its configuration). To complicate evasion, a stateful rooting detection mechanism could be used. For instance, an app can first create an interactive shell that will not trigger evasion (e.g., `Runtime.exec('sh')`, `execv ('sh')`). Once the shell object has been successfully obtained, the app then pipes detection commands (e.g., `ls /system/xbin/su`, `which su`) through the input stream of the shell object and examines the output to determine whether the device has been rooted. In order to evade this detection method, root cloakers need to intercept both input and output streams of the shell object.

5.2 Limitations

App samples. In the course of our study, we observed that there are two general reasons why developers add root detection logic to an app. The first reason is to improve usability—to tell users whether their devices are rooted, to warn users that their devices have not been rooted when the app need root access to perform an action, or to display different options to users based on whether the device has been rooted. The second reason is for security—to protect sensitive or high value user data against the security impact of having a persist root access on the device.

Intuitively, apps detecting root for usability reasons might need a lower quality of root detection than apps detecting root for user data protection. In the confirmatory study, we excluded paid apps and those apps that required having accounts that we were unable to register (e.g., banking, mobile device management apps). Most of those excluded apps might be performing rooting detection for security reason, which might employ such detection techniques that are more difficult to evade. In addition, the majority of the apps in our sample were downloaded from Google Play, except for a small number of popular root apps downloaded from third-party app stores (because they are available only there). We plan to address these limitations as part of a larger-scale version of the confirmatory study.

Rooting method samples. We used XDA Developers Forums as our primary source for studying existing rooting methods. There may be other existing rooting methods from other sources that we are unaware of.

5.3 Future Work

In addition to addressing the limitations of our current study, we plan to conduct the following future work:

Understanding user's risk perception of rooting. The security of a rooted device relies on the device user to regulate root access properly. Thus, understanding user's mental model and risk perception of rooting could provide informed design improvements to raise user awareness and improve root-management functionalities.

Profiling root apps for reliable detection. Understanding and characterizing the behaviours of root apps (e.g., system calls, objects accessed and actions) might shed lights on runtime detection of rooted devices, and provide informed design improvements for root-management apps.

Hypothetically, a root detection approach can also look for telltale signs that the device was put in a configuration state that allowed future (or past) installation of a root framework. While we did not find any approaches that do so, it could be part of future studies.

6. CONCLUSION

This work uncovers the details of arm race between rooting methods and prevention mechanisms, as well as rooting detection and evasion techniques. By creating a taxonomy of the rooting methods, we found that despite the layers of several countermeasures that, besides other protections, are supposed to resist rooting, pre-boot rooting methods would remain as options for rooting so long as "freedom of customization" is embraced by vendors. For rooting detection and evasion, we found a wide variety of techniques used by apps to detect rooted devices, but unfortunately, all rooting detection methods studied can be evaded. Our study results suggest that, ultimately, a reliable rooting detection method should be provided by Android OS, with rooting detection logic implemented in the trusted parts of the system, such as integrity-protected kernels or external trusted execution environments.

7. ACKNOWLEDGEMENTS

We would like to thank anonymous reviewers for their helpful feedback, which allowed us to improve the paper and our research methods.

8. REFERENCES

[1] ClockworkMod Recovery. https://twrp.me/. [Online; accessed 29-May-2015].

[2] CyanogenMod. http://www.cyanogenmod.org/. [Online; accessed 29-May-2015].

[3] Cydia Substrate: The powerful code modification platform behind Cydia.

http://www.cydiasubstrate.com/. [Online; accessed 29-May-2015].

[4] Dex2Jar. https://github.com/pxb1988/dex2jar. [Online; accessed 29-May-2015].

[5] Fastboot. http://www.xda-developers.com/tag/fastboot/. [Online; accessed 29-May-2015].

[6] Heimdall. http://glassechidna.com.au/heimdall/. [Online; accessed 29-May-2015].

[7] iRoot. http://www.mgyun.com/m/en/. [Online; accessed 29-May-2015].

[8] JAD: Java Decompiler. http://varaneckas.com/jad/. [Online; accessed 29-May-2015].

[9] Java Decompiler GUI. http://jd.benow.ca/. [Online; accessed 29-May-2015].

[10] ODIN. http://www.xda-developers.com/tag/odin/. [Online; accessed 29-May-2015].

[11] One Click Root for Android. http://www.oneclickroot.com/. [Online; accessed 29-May-2015].

[12] Root Explorer. https://play.google.com/store/apps/details?id=com.speedsoftware.explorer. [Online; accessed 29-May-2015].

[13] Root Genius. http://http://www.shuame.com/en/root/. [Online; accessed 29-May-2015].

[14] Team Win Recovery Project. http://forum.xda-developers.com/wiki/ClockworkMod_Recovery. [Online; accessed 29-May-2015].

[15] Titanium Backup. http://www.titaniumtrack.com/titanium-backup.html. [Online; accessed 29-May-2015].

[16] Xposed Framework. http://www.cydiasubstrate.com/. [Online; accessed 29-May-2015].

[17] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *Proceedings of the 22nd Usenix Security Symposium (Security 13)*, 2013.

[18] cernekee. Multiple Android Superuser vulnerabilities. http://forum.xda-developers.com/showthread.php?t=2525552, 2013. [Online; accessed 29-May-2015].

[19] ChainsDD. su binary for android Superuser. https://github.com/ChainsDD/su-binary, 2012. [Online; accessed 29-May-2015].

[20] China Internet Watch. 80% China's Mobile Users Rooted Smartphones in 2014. http://www.chinainternetwatch.com/12926/80-china-smartphone-users-rooted/, 2015. [Online; accessed 29-May-2015].

[21] devadvance. RootCloak. https://github.com/devadvance/rootcloak, 2014. [Online; accessed 29-May-2015].

[22] devadvance. RootCloak Plus - Completely Hide Root from Apps. http://forum.xda-developers.com/showthread.php?t=2607273, 2014. [Online; accessed 29-May-2015].

[23] S. Egelman, L. F. Cranor, and J. Hong. You've been warned: an empirical study of the effectiveness of web browser phishing warnings. In *CHI '08: Proceedings of the SIGCHI Conference on Human factors in Computing Systems*, pages 1065–1074, New York, NY, USA, 2008. ACM.

[24] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[25] Garner Inc. Market Share: Devices, All Countries, 4Q14 Update. http://www.gartner.com/document/2985017, 2014. [Online; accessed 29-May-2015].

[26] Google Inc. Inside OTA Packages. https://source.android.com/devices/tech/ota/inside_packages.html. [Online; accessed 29-May-2015].

[27] Google Inc. Android Security 2014 Year in Review. https://static.googleusercontent.com/media/source.android.com/en//devices/tech/security/reports/Google_Android_Security_2014_Report_Final.pdf, 2014. [Online; accessed 29-May-2015].

[28] Google Inc. Security Enhancements in Android 4.3. https://source.android.com/devices/tech/security/enhancements/enhancements43.html, 2014. [Online; accessed 29-May-2015].

[29] Google Inc. Verified Boot. https://source.android.com/devices/tech/security/verifiedboot/verified-boot.html, 2014. [Online; accessed 29-May-2015].

[30] Jorrit Chainfire Jongma. How-To SU: Guidelines for problem-free su usage. http://su.chainfire.eu/, 2014. [Online; accessed 29-May-2015].

[31] P. Kelley, S. Consolvo, L. Cranor, J. Jung, N. Sadeh, and D. Wetherall. A Conundrum of Permissions: Installing Applications on an Android Smartphone. In J. Blyth, S. Dietrich, and L. Camp, editors, *Financial Cryptography and Data Security*, volume 7398 of *Lecture Notes in Computer Science*, pages 68–79. Springer Berlin Heidelberg, 2012.

[32] Koushik Dutta. ClockworkMod Superuser. https://github.com/koush/Superuser, 2014. [Online; accessed 29-May-2015].

[33] Y. Shao, X. Luo, and C. Qian. Rootguard: Protecting rooted android phones. *Computer*, 47(6):32–40, June 2014.

[34] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of the 2013 Network and Distributed System Security Symposium (NDSS'13)*, 2013.

[35] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying Wolf: An empirical study of SSL warning effectiveness. In *Proceedings of 18th USENIX Security Symposium*, pages 399–432, 2009.

[36] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: A survey of current android attacks. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[37] XDA. XDA Developers. http://forum.xda-developers.com/, 2015. [Online; accessed 29-May-2015].

[38] Z. Zhang, Y. Wang, J. Jing, Q. Wang, and L. Lei. Once Root Always a Threat: Analyzing the Security Threats of Android Permission System. In *Information Security and Privacy*, volume 8544 of *Lecture Notes in Computer Science*, pages 354–369. Springer International Publishing, 2014.

[39] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, May 2012.