IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 19, 2020, accepted October 22, 2020, date of publication November 2, 2020,
date of current version November 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3035089

Android Data-Clone Attack via Operating System Customization

WENNA SONG[1], MING JIANG[2], HAN YAN[1], YI XIANG[1], YUAN CHEN[1], YUAN LUO[1], KUN HE[1], AND GUOJUN PENG[1]

[1]Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China
[2]Computer Science and Engineering Department, The University of Texas at Arlington, Arlington, TX 76019, USA

Corresponding author: Guojun Peng (guojpeng@whu.edu.cn)

ABSTRACT To avoid the inconvenience of retyping a user's ID and password, most mobile apps now provide the automatic login feature for a better user experience. To this end, auto-login credential is stored locally on the smartphone. However, such sensitive credential can be stolen by attackers and placed into their smartphones via the well-known credential-clone attack. Then, attackers can imperceptibly log into the victim's account, which causes more devastating and covert losses than merely intercepting the user's password. In this article, we propose a generalized Android credential-clone attack, called data-clone attack. By exploiting the new-found vulnerabilities of original equipment manufacturer (OEM)-made phone clone apps, we design an identity theft method that overcomes the problem of incomplete credential extraction and eliminates the requirement of root authority. To evade the consistency check of device-specific attributes in apps, we design two environment customization methods for app-level and operating system (OS)-level, respectively. Especially, we develop a transparent Android OS customization solution, named CloneDroid, which simulates 101 special attributes of Android OS. We implement a prototype of CloneDroid and the experimental results show that 172 out of 175 most-downloaded apps' accounts can be jeopardized, such as Facebook and WeChat. Moreover, our study has identified 18 confirmed zero-day vulnerabilities. Our findings paint a cautionary tale for the security community that billions of accounts are potentially exposed to Android OS customization-assisted data-clone attacks.

INDEX TERMS Automatic login, data-clone attack, identity theft, OS customization.

I. INTRODUCTION

Nowadays, most of the existing mobile apps support automatic login mechanism [1]–[5], which reduces the hassle of typing user ID and password in a small keyboard and thus optimizes the users experiences. With such mechanism, even if apps have been launched for an extensive time (e.g., half a year) or phones have been restarted, users can still enjoy the services smoothly (e.g., payment, video streaming services) without re-authentication. For example, users can launch the Tencent video app to watch the very important person (VIP)-only movies without passwords or pay directly via the Alipay app if the ''Password-free Payment/Automatic Deduction'' function has been activated.

The automatic login mechanism depends on the login credentials, which is returned by the server and stored locally when the users log into the accounts for the first time. Researchers have confirmed that these private credentials can be extracted from the victim's phone and placed into the attacker's phone. Then, the attacker can access the victim's account without knowing the victim's ID and password. This type of attack is called credential-clone attack [6]–[8]. Compared with login the victim's account with stolen ID and password, the credential-clone attack is more stealthy since it can evade many account protecting mechanisms, such as login notification on new devices and device limitations.

Unfortunately, the previous work only tested a single-digit number of apps to prove the existence of this type of attack, and we were unable to reproduce their attack on most apps in an extensive dataset. Their limitations are threefold. First, those attacks require the attacker to obtain the root authority and access private local storage on the victim's device, which is a relatively strong assumption on the attack capability. Second, in addition to the files that save user credentials, many auto-login functions also rely on different private files under the directory of ''/data/data/[app_name]/,'' such as configuration files. Therefore, only copying a single credential file is not enough. Third, an increasing number of apps implement consistency check of device-specific attributes. More specially, if they detect any device-specific discrepancies,

The associate editor coordinating the review of this manuscript and approving it for publication was Amjad Ali.

they will disable the auto-login feature and switch to the manual login pages.

In this article, we investigate the security risk caused by mobile apps' auto-login feature and propose a more generalized attack strategy, called *data-clone attack*. For the apps that do not implement a consistency check, we design a basic attack scheme. To overcome the limitations of previous work (i.e., the first and second limitations), we discover a new data-clone attack vector—original equipment manufacturer (OEM)-made phone clone apps [9], which support transferring private data in ''/data'' partition between two phones of the same manufacturer. With this attack vector, the attacker can steal the victim's private data without obtaining root privileges. During the basic attack, we clone all of the data under the directory of ''/data/data/[app_name]/'' into the attacker's device to bypass user authentication. In this way, the attacker also avoids the trouble of locating the various particular storage files that are associated with the auto-login function.

For the apps that implement a consistency check, we propose an advanced attack scheme based on the basic attack. To overcome the third limitation of previous work, we first design an automated tool to reverse test apps and extract device attributes. Then, we determine 101 device attributes based on the output of the automated tool and develop a general app-level custom device attribute editing platform using Xposed [10] technology. However, considering that some critical apps (e.g., Wechat) can even fingerprint the Android runtime hooking framework Xposed, we continue to explore an Android operating system (OS)-level customization solution, named *CloneDroid*. In the above-customized environment, attackers can freely configure the phone to match different victim phone profiles.

To assess Android apps' susceptibility to our data clone attacks, we conducted empirical research on the 175 most-downloaded apps from American and Chinese Android app markets on Google Nexus 6P phone devices. The basic attack can successfully bypass user authentication and uses 135 apps (e.g., Facebook, Snapchat, QQ, Weibo, Netflix, and Prime Video) account authorization functions. For the remaining 40 apps, we implemented the advanced attack, and 37 apps were successfully cloned into the customized environment. More specifically, the Xposed-based customized environment made 17 out of 37 apps successfully attacked; CloneDroid made all 37 apps attacked successfully.

Experimental results verify that many mainstream apps are vulnerable to data-clone attacks. Although some apps have already checked the consistency of device footprints to secure the auto-login function, however, these protection mechanisms are weak and cannot resist our data-clone attacks. The results of comparative experiments using the Xposed technology solution and the Android OS modification solution verify that the OS-level customization environment is more transparent to clone apps. So it is difficult for clone apps to detect device changes. We hope that our work spurs discussion and inspires the security community to redesign auto-login functions. Otherwise it leaves billions of Android accounts vulnerable to Android operating system customization-assisted data-clone attacks. In a nutshell, we make the following three significant contributions:

- Our work presents a generalized procedure of Android data-clone attack. This attack exploits the vulnerability of the OEM-made phone clone apps, which can overcome the problem of incomplete credential extraction and eliminate root authority requirement.
- We develop two device-attribute editing platforms for app-level and OS-level, respectively, to assist Android data-clone attacks. Especially, the Android OS-level customization solution, named CloneDroid, reveals a strong resilience to various device-consistency checks. Our in-depth study shows that realistic customization of the victim's smartphone is the key to launch a data-clone attack successfully.
- Our work reveals the security risk of Android apps' auto-login feature. A set of experiments on 175 most-downloaded apps indicate that 98% of them are vulnerable to our proposed attack. Besides, 18 vendors confirmed our zero-day vulnerability.

A. ETHICAL CONSIDERATIONS

We have made responsible disclosure to the app vendors that are severely vulnerable to the data-clone attack. 18 vendors have confirmed our report as vulnerability. Some of the mainstream security vendors, for example, Netflix, Tencent, iQiyi, Alibaba, Qihoo 360, Xiaomi, NetEase, etc. Our experiments do not involve personally identifiable information or other kinds of sensitive data.

II. BACKGROUND AND RELATED WORK

In this section, we first introduce the popularity of the app automatic login mechanism and related knowledge. We also discuss the automatic login security risk from android apps and highlight that the existing works are limited. We then introduce OEM-made phone clone apps, which we take as an attack vector to clone private data. At last, we introduce related work on Android sandbox detection.

A. AUTOMATIC LOGIN MECHANISM OF APPS

Usually, due to Android's small-scale touchscreen limiting one app to running in a smartphone's foreground, users frequently switch to other apps in the background. Considering these limitations of smartphone resources, if multiple apps exist in the background for a long time, they may be killed by the system or users to release resources. In this case, if the user has to enter the ID and password every time they access the app, it is very troublesome. In fact, most apps support automatic login mechanism by default. As a result, users only need to input their ID and password at their first login time. After that, users can access the app smoothly without retyping their ID and password. By killing the app process to clear the app cache's login status and then restarting the app process, the app's automatic login function is effective once the app appears in the login status. We selected 175 apps to check the automatic login function and found that 174 apps are available.

Most auto-login functions store user credential data locally and use the credentials to build the subsequent authenticating interactions with the server. User credentials are typically stored either in the form of key-value pairs as SharedPreferences or structured data in an SQLite database. These data are usually located in the ''/data/data/[app_name]/'' private directory. In this case, the automatic login mechanism plays an important role. That is, once the login credentials are valid, the app has the opportunity to use the automatic login mechanism to restore the login status.

B. AUTO-LOGIN MECHANISM SECURITY RISKS AND LIMITATIONS OF EXISTING WORK

User credential data are always an attractive target for cybercriminals [11], [12]. Suppose cybercriminals steal the target app's login credentials on the victim's device and push them under their phone's same directory. In that case, the target app on the cybercriminals' device can automatically login to the victims' accounts without knowing their ID and password. That means cybercriminals can leverage the auto-login mechanism to bypass the authentication from the server-side. As a result, the user's sensitive data will be in jeopardy without raising suspicion. Recent work has discussed such security risk [6]–[8] and proposed a ''user credential cloning attack'' to exploit the pervasive auto-login feature in Android apps. These studies are based on some common assumptions. For example, the victim's device is rooted, and the attacker can physically access the victim's phone or use malware installed on the victim's machine to steal credential data. A large number of potentially harmful applications will use privilege escalation vulnerabilities to root devices for different attack purposes [13], [14]. These papers [6]–[8] showed the feasibility of data-clone attack with only six apps. However, after testing with 175 most-downloaded apps in October 2019, we only reproduce their proposed attack on less than 11% apps. Upon further investigation, we locate two major limitations that lead to the remaining failure cases.

Limitation 1: only cloning user credential data. The previous approaches [6]–[8] first identify the location of the storage file that saves user credentials. Then they clone that storage file to the same directory of a target device. However, in addition to user credentials, more than 70% of our tested apps' auto-login functions also depend on various files under the folder of ''/data/data/[app_name]/.'' We take WeChat, a social media app with over 1 billion daily active users [15], as a case study. WeChat stores AES-encrypted user credentials in a SQLite database file ''EnMicroMsg.db.'' This file is under the directory of ''/data/data/MicroMsg/[xxxx...xxxx]/,'' in which ''xxxx...xxxx'' is the 32-bit md5 value of a file name. We reverse-engineer WeChat's auto-login function and find that it relies on multiple files under the same directory and a system configuration file, ''/data/data/MicroMsg/systemInfo.cfg.'' ''systemInfo.cfg'' is an XML plaintext containing the connection information with the app server. Apparently, only cloning ''EnMicroMsg.db'' is not enough at all. Note that the exact files that are needed by the auto-login function vary on a case-by-case basis. Therefore, the best strategy is to clone all of the data under ''/data/data/[app_name]/.''

Limitation 2: unaware of device-consistency checks. We still take WeChat as an example to explain the second limitation of existing work [6]–[8]. We cloned all of the data under ''/data/data/MicroMsg/'' from one smartphone to another smartphone, but we found that WeChat pops up the login interface on another device and asks us to retype ID and password. The root cause of this failed result is that WeChat can detect changes in the smartphone environment, and then terminate the automatic login process. We reversed WeChat and found that WeChat has detected many device fingerprints, such as phone number, international mobile equipment identity (IMEI), and Bluetooth address. In our dataset, a total of 37 apps such as Chrome, Apple Music, KakaoTalk, and PayPal also conduct a similar detection when invoking their auto-login functions. However, no previous work [6]–[8] further researched device-consistency checks and even realized that automatic login would be protected by device consistency.

Compared to the measures that can address Limitation 1, overcoming Limitation 2 will be much more challenging. Bianchi *et al.* [16] describe a scheme to bypass app authentication by simulating device-public information. However, their login credentials are stored in publicly accessible locations that any apps running on a device can access (i.e., Google authentication). Thus, they are studying different types of app authentication from ours. In our tested 175 most-downloaded apps, no app adopts such an authentication scheme, including WhatsApp and Viber. Besides, they use the Xposed framework [10] to create a custom environment to simulate device-public information[10]. However, the hook traces of their scheme are easily detected by the test apps, and the number of device attributes they simulate is only 7. We have extended their Xposed framework scheme and proposed an OS-level device-attribute editing platform that is transparent and can edit nearly ten times more device attributes. CloneDroid has broad applications that rely on a customized phone environment, such as analyzing trigger-based malware [17], [18] developed for a specific phone model.

C. OEM-MADE PHONE CLONE APPS

Our proposed new attack utilizes OEM-made phone clone apps to address Limitation 1. OEM-made phone clone apps [9] automatically transfer data (e.g., photos, music, apps, and contacts) and even locally-stored app-private data from one device to another one. Based on these advantages, OEM-made phone clone apps are becoming increasingly popular among users and phone manufacturers. Their user downloads have even exceeded 100 million times, and many well-known Android phone manufacturers (e.g., Samsung, ONEPLUS, OPPO, Huawei, Xiaomi) [19] have also made their own phone clone tools (e.g., OnePlus Switch, Huawei Phone Clone, and Xiaomi Mi Mover).

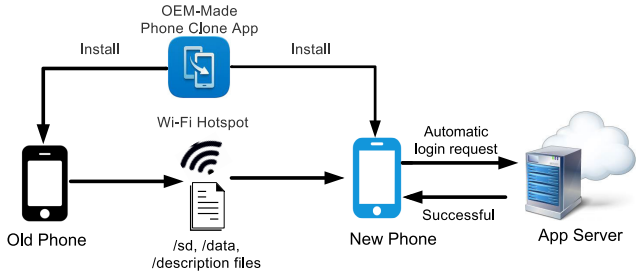


FIGURE 1. OEM-made phone clone app.

These OEM-made phone clone apps have a unique advantage: they have the privilege to call Android backup application programming interface (API) [20] on the same OEM phones. Therefore, they can access app-private data in ''/data' partition and transfer them between the same OEM phones without rooting devices. This advantage brings users great convenience when they upgrade their devices. Fig. 1 illustrates an example of OEM-made phone clone app. It sets up a Wi-Fi hotspot to transfer data including ''/data'' partition from the old phone to the new phone. Therefore, the new smartphone just becomes the replica of the old device, and the user can still access the apps without retyping their ID and password.

D. DETECTING ANDROID SANDBOX ENVIRONMENT

Another line of research related to our work is Android sandbox environment detection, which is related to our work on Limitation 2. The sandboxes based on full-system emulation provide an isolated dynamic malware analysis environment. However, a challenge of full-system emulation is to realistically simulate various hardware and device effects [21]. Therefore, related work [22]–[26] has summarized a set of detection heuristics to find the hardware-related discrepancies caused by full-system emulation. For example, Bordoni *et al.* [25] find that the return values of sensor-related APIs are different between mobile emulators and real devices. In contrast, our custom environment design can directly access hardware devices at any time. Another kind of sandboxes is built on app-virtualization techniques such as Xposed [10] and VirtualApp [27]. The host app creates a virtual machine like environment, which is able to run multiple copies of the same apps (i.e., guest apps) [28]–[30]. However, the host app has to heavily rely on hooking mechanism to deceive both Android system services and guest apps, which leaves many host app's signatures in the guest app's call stack and memory [29]. Our CloneDroid's device customization and configuration do not adopt hooking mechanism and thus have better transparency than app-virtualization techniques.

III. BASIC ATTACK

To address Limitation 1 of the previous work, we define a ''data-clone attack'' as the attack that clones ''/data/data/[app_name]/'' into the device of an attacker to bypass user authentication. In this section, we propose the detailed design of the basic attack scheme, and analyze two types of case studies: 1) victim identity theft, and 2) break through the paying-subscriber limit. We reveal that data-clone attacks are a real threat to both the app revenues and user privacy.

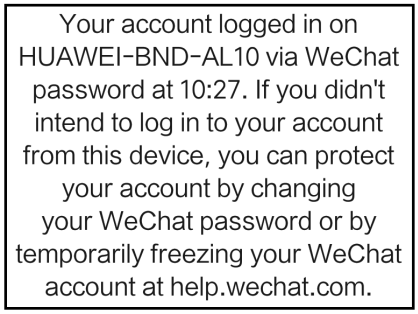


FIGURE 2. WeChat's warning notification.

A. DATA-CLONE ATTACK'S ADVANTAGE

Our work exploits pervasive auto-login functions to bypass user authentication. Compared with login the victim's account with stolen ID and password, our data-clone attack is stealthy since it can evade many accounts protecting mechanisms. For instance, logging in by entering the user ID and password triggers the login device identity restriction. Many apps limit the number of logins on the phone simultaneously with the same account. Once the limit is exceeded, the app will notify the user whether to allow login on the current device. For example, WeChat only allows a single user to log in from one mobile device at a time. When an attacker logs into WeChat from a different phone by typing the victim's ID and password, the victim's phone will receive a warning notification as shown in Fig. 2. However, our key observation is that multiple auto-login attempts of the app from the same device will not affect server counting the number of login devices, which leaves us a backdoor to eliminate the login-device number limit. Thus, our data-clone attack can make the attacker login to the victim's account and send no notifications to the victim, so it will not arouse the victim's suspicion. Suppose a messaging app (e.g., WhatsApp) is compromised in this way. In that case, the adversary can not only view or modify the victim's account information (e.g., address book, email) but also impersonate the victim to send messages, which may harm the privacy of victims and reputation of the company's products.

B. DATA-CLONE ATTACK MODEL

In this article, we build a data-clone attack model based on formal methods. To the best of our knowledge, we are the first to perform formal analysis on data-clone attack on a mobile platform. This model includes the attack vector (A_v), attack hypothesis (A_h), attack object (A_o), attack conditions (A_c), and attack result (R). Formally describe our attack model as follows.

$$A_v + A_h + A_o \xrightarrow{A_c} R \tag{1}$$

In (1), this model expresses that under A_h and A_c, the adversary attacks A_o through A_v and then produces R. In this model, R represents the attack model's result. As a result,

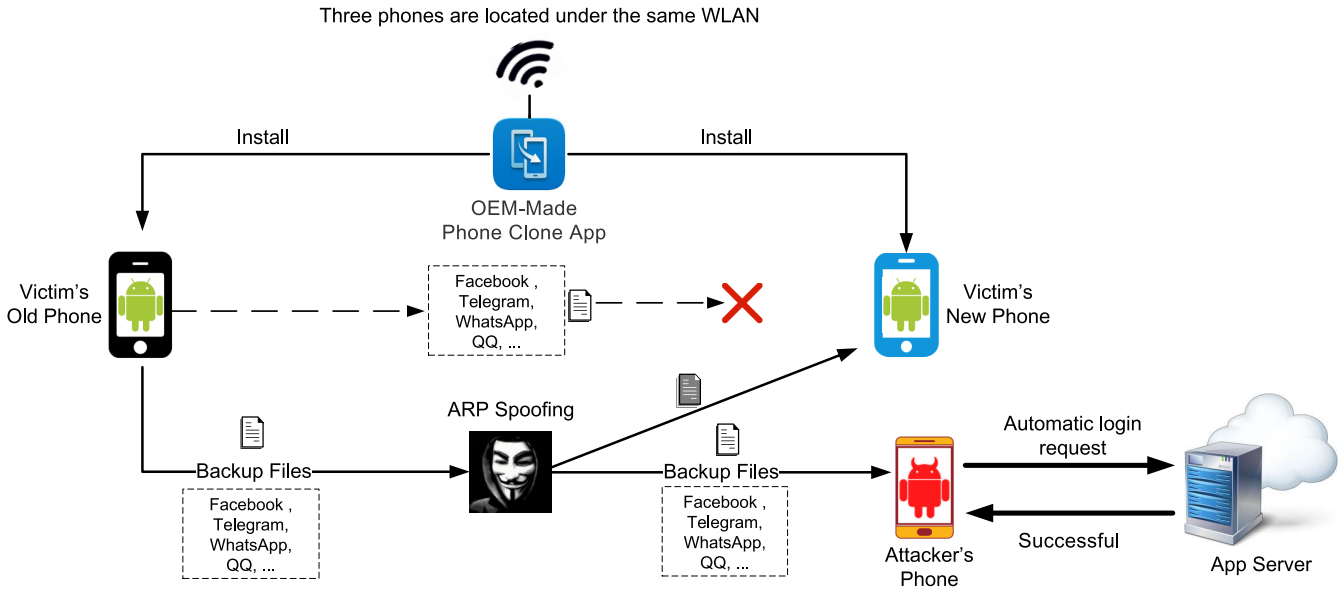


FIGURE 3. Data-clone attack model: data-clone attack steals user credentials to access victim's accounts automatically.

the attackers can hijack the victim's data and push the data to his device. A_o is a representation of an attack object app. A_c represents the attack scenario's occurrence condition where the victim uses an OEM-made phone clone app to migrate the A_o between the same manufacturer.

A_h defines the restrictions under which the attacker performs the above attack as follows.

$$A_h \stackrel{\text{def}}{=} \exists V_p \bigwedge A_a \tag{2}$$

In (2), this model explains what A_h is. A_h consists of the constraints on the victim's phone system (Vp) as well as limitations on the ability of the adversary's ability (A_a). When A_a satisfies the attack requirement, and there is a V_p scenario, this model can produce A_h's result. We assume that V_p is a trusted environment without root and does not contain any vulnerabilities. We limit A_a cannot physically access or inject malware into the V_p to clone A_o's automatic login data. Besides, A_a do not have access to servers, so the adversary cannot modify server-side logic.

A_v represents the OEM-made phone clone app vulnerability used by attackers to launch attacks as follows.

$$A_v \longrightarrow \exists precondition \bigwedge A_m \tag{3}$$

In (3), A_v is effective if *precondition* is occurring and there is an OEM made phone clone app vulnerability. *precondition* indicates there is a scenario where the victim is using the OEM-made phone clone app to transfer A_o data between two same manufacturer phones. A_m represents A_a can successfully find vulnerabilities (weak passwords, plaintext transmission, etc.) in the OEM-made phone clone app to intercept private user data. Then, we use a case study to demonstrate that attackers can use this attack model to intercept user private data and push the data to their device.

This vulnerability from a top OEM-made phone clone app involves two design flaws: weak password and plaintext transmission. First, the app sets up a Wi-Fi hotspot to transfer data, but the Wi-Fi password is just the first eight alphanumeric characters of the Wi-Fi hotspot service set identifier (SSID)'s MD5 value. Therefore, as long as an attacker's device can detect the Wi-Fi hotspot signal, he can easily calculate the password and access the wireless local area network (WLAN). Second, this phone clone app transfers data in plaintext, which causes the attacker to intercept user private data by applying address resolution protocol (ARP) spoofing [31] or packet sniffing [32]. As a result, when a victim is cloning phone data in some places, it is very likely that an adversary performs a man-in-the-middle (MITM) attack but without raising suspicion. Fig. 3 illustrates how we exploit this vulnerability for the purpose of data-clone attack. After we get into the WLAN via the weak password, we send spoofed ARP messages to associate our phone's media access control (MAC) address with the internet protocol (IP) address of the victim's new phone, so that the traffic meant for the victim's new phone will be redirected to our phone. We intercept "/data/data/[app_name]/" and put them in our phone to bypass user authentication. Note that this vulnerability exists in all versions of this OEM-made phone clone app, which are across nine years. The app vendor, one of the top-three Android phone manufacturer has confirmed our finding as a high-severity vulnerability and assigned a common vulnerabilities and exposures (CVE)-2019-15843.

C. CASE STUDY

1) VICTIM IDENTITY THEFT

App user identity is used by the server to distinguish different users. The app relies on a specific user identity to request the server's response to access an account authorization page. Once the user enters the page, he can browse a large amount

of user privacy information. Currently, a large number of apps rely on users identities to perform various critical tasks. Typical examples include editing personal information and sending chat messages, such as WhatsApp, Facebook, and QQ. Using the attack model in Fig. 3 can break the access control from the victim's identity and pretend to be the victim using the app to interact with the server continuously. For example, the attacker used this attack model to steal the login credential file of WhatsApp, then push the data to his mobile phone to restore the login status, and found that it does not affect the victim's use of WhatsApp functions. WhatsApp only allows a single phone to log in at a time. However, our data-clone attack model can break this limitation.

2) BREAK THROUGH THE PAYING-SUBSCRIBER LIMIT

Breakthrough the paying-subscriber limit is another app function damage after victim identity theft caused by data-clone attacks. The subscription-based app economy thrives in mobile markets, and customers have acclimated to the idea of regular payments for a better service [33]. Typical examples are the apps that provide video and music streaming services, such as Netflix, Amazon Prime Video, and iQiyi. For a subscription-based app, only a paying subscriber can enjoy its premium service, and it also enforces the maximum number of the same user's login from different devices at a time. For example, Netflix's premium plan allows at most four screens a user can watch on at the same time. However, our data-clone attack can break the paying subscriber limit. Fig. 3 illustrates such an example, and eventually, the attacker can access Android premium apps in his devices without payments. Although Fig. 3 just shows a single-user attacker case, once attack model is turning into full-fledged, coordinated attacks in Android black market, malicious actors can infringe the revenue model of subscription-based apps, resulting in a great financial loss to software vendors. Most of the zero-day vulnerabilities that we found belong to this category, and the leading app vendors such as Netflix, Xiaomi, and Alibaba, have confirmed our findings.

IV. ADVANCED ATTACK WITH CUSTOM ENVIRONMENT

In this section, we introduce the advanced attack, which continues the attack model of hijacking private data in §III. The difference is that these private data are not directly put into the victim's real machine, but into the customized environment that matches the attributes of the victim's device. Next, we will introduce the custom environment in detail, and this can address the Limitation 2. Our custom environment design attempts to achieve two design goals: 1) our design ensures that Android emulator detection heuristics are in vain for our platform; 2) the custom environment in the attacker's device should be imperceptible to cloned apps. In order to achieve the first goal, we design customized solutions that run on real machines to counter the emulator fingerprint detection of the apps. Then, we elaborate on how we achieve our second design goal—evading device-consistency checks so that cloned apps are unaware of the change of device.

A. OVERVIEW

In order to counter the device consistency check in the apps, we conducted a shallow and profound analysis of the app's environmental detection capabilities, respectively. The shallow means that the app only checks the changes in device-specific attributes. The profound analysis not only checks the consistency of the device but also detects hook fingerprints.

Most apps detect different device attributes. To cover as many attributes as possible, we attempt to collect data from various sources: 1) android open-source project (AOSP) code, 2) the existing commercial Android sandbox detection tool, and 3) our automated reverse engineering of the apps that perform device-consistency checks. We use python language and embed the apktool library to build an automatic attribute extraction tool. This tool automates the reverse test apps and extracts the device attribute data. Tab. 1 shows the distribution of editing device attributes that contain eight categories, with up to 101 attributes. Thus, these custom device attributes are comprehensive. Besides, the device attributes are all present in the tested mainstream apps, so the collected device attributes are practical.

Using the above attribute information, we have implemented two custom environments to support data-clone attacks bypass device-consistency checks. One is built by using mainstream Xposed technology, and the other is created directly by modifying the AOSP code. Considering that the AOSP code is open and easy to download [34], we have modified them to avoid app detection of hooks. Although there are many ways to hide the traditional hook technology, it is difficult to conceal all hook points with the constant game of attack and protection.

B. ADVANCED ATTACK WITH XPOSED-BASED CUSTOM ENVIRONMENT

Xposed is an advanced Android runtime hooking framework [10], and can provide an app-virtualization environment, in which a user can customize many device attributes such as central processing unit (CPU) model, SSID, MAC address, phone number, and IMEI. The underlying mechanism of Xposed is performing API hooking to return fake device attributes and thus deceive guest apps (apps running in the sandbox environment virtualized by hook technology). Android device-attribute editing tools such as XxsqManager, iGrimace, and NZT can achieve this goal, and all of them are Xposed-based sandboxes. Fig. 4 shows the configuration page of an Android device-attribute editing tool, and users can input new parameters that are different from the current physical device's profiles to simulate a new phone's runtime environment.

The number of device attributes contained in the existing commercial Xposed environment is not comprehensive enough. Besides, the Android system provides developers with multiple interfaces to obtain an object. In order to prevent the app from bypassing our hook points, we try to hook more specific attributes from multiple objects.

TABLE 1. Customizable device-attribute options and numbers.

Type	Customizable Device-Attribute Options	Number
System Property	Hardware Serial Number, Device Fingerprint, Device Version Number, Device Model, Device TAGs, Manufacturer, Device Version Type,Brand, Httpagent, Device Bootloader, Product Board, Product Locale Language, Compile machine name, Compiler, SDK, SDK_INT, Version increment, Product Device, User Key, RADIO, Compile time, Compile type, Product Name, Specific version number, Product Local region,Version ID.	26
Kernel Version	UTS_VERSION, LINUX_COMPILE_BY, LINUX_COMPILE_HOST, LINUX_COMPILER, UTS_VERSION.	5
Memory	AvailROMSize, TotalROMSize, AvailRAMSize, TotalRAMSize.	4
CPU	CPU Model, CPU Cores, CPU Hardware, CPU Architecture, CPU Version, CPU Variant, CPU Part, Feature, CPU Serial Number, CPU Vendor.	10
Network	MAC address, SSID, BSSID, RSSI, IP Address, DNS1, DNS2, Gateway, Netmask, WiFiState, NetworkInerfaces, TypeName, NetworkId, NetworkType.	14
Power	Battery Level, Battery Type, Battery Temperature, Battery health, Battery Voltage.	5
Bluetooth	Bluetooth Name, Bluetooth Address, Bluetooth Scanmode, Bluetooth State.	4
Location	Longitude, Latitude, Bearing, Altitude, Location Area Code, Cell Identity, NetworkId, SystemId, BaseStationId.	9
Telephony	IMEI1, IMEI2, MEID, IMSI, IMEISV, ESN, ICCID, Phone Number, SIMState, carrier_name, Mobile Country Code, SIMOperatorName, Phone Type, Mobile Network Code, SIMCountryIso, SIMOperator.	16
Display & GPU	GPU Version, Vendor, Density, Renderer, Resolution, ScaledDensity, xdpi/ydpi, GPU Extension.	8

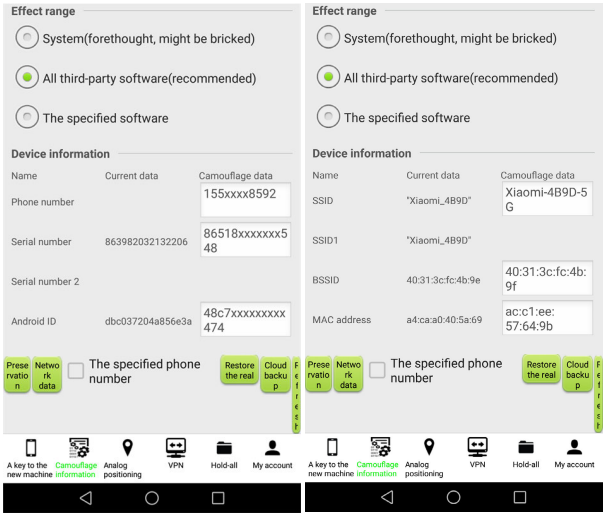


FIGURE 4. Configuring device-specific attributes in a Xposed-based sandbox.

Our hook objects include system APIs, fields, and files. The modification of the "System Property" in the Tab. 1 is mainly implemented by hooking the "value" corresponding to the "filename" of the "Build.class" by "XposedHelpers.setStaticObjectField." The "CPU" attributes are customized through file redirection, and other attributes are customized by the hook system APIs. Finally, we can use Xposed to provide customization for 101 device attributes.

However, app-virtualization technique is not completely transparent to guest apps [29]. For example, The hooking mechanism adopted by Xposed leaves identifiable fingerprints in package name, call stack, suspicious native methods, and shared objects loaded into memory [35]. Besides, installing Xposed framework requires rooting device. We find some cloned apps (e.g., Chrome, Alipay, and Apple Music) can detect the existence of Xposed or root, and thus prevent data-clone attacks. In what follows, we explore a realistic customization environment by leveraging Android OS code modification technology.

C. ADVANCED ATTACK WITH OS-LEVEL CUSTOM ENVIRONMENT

We have developed an OS-level custom environment, named *CloneDroid*. With CloneDroid, attackers are free to configure different device settings according to the victim phone's attributes. Our design ensures that Android emulator and app-virtual detection heuristics are in vain for CloneDroid and cloned apps are almost unaware of device changes.

In order to achieve these goals, our customization is mainly based on the native and kernel system functions. From bottom to top, the Android system's startup sequence is the kernel, native framework, java framework, and application layer, and the last three parts belong to the user space of the Android system. The user space generates a soft interrupt by issuing a system call to the kernel space, thereby causing the program to fall into the kernel state and perform corresponding operations. The user apps have no permission to operate the system call for entering the kernel space. Although, the native framework provides a "jni interface" that can be called directly by the app. Once we modify these interfaces, the value returned to the app is still our modified value. Therefore, it is reasonable for CloneDroid to focus on native and kernel layers to modify. Tab. 2 reveals the information about the customized location.

Fig. 5 shows the workflow of CloneDroid. CloneDroid provide a configuration file "build.clonedroid.prop" in advance. CloneDevice app in Fig. 5 is used to edit device properties by users, and these property values will eventually be filled into "build.clonedroid.prop." The device property changes will take effect after the system reboots, and CloneDroid

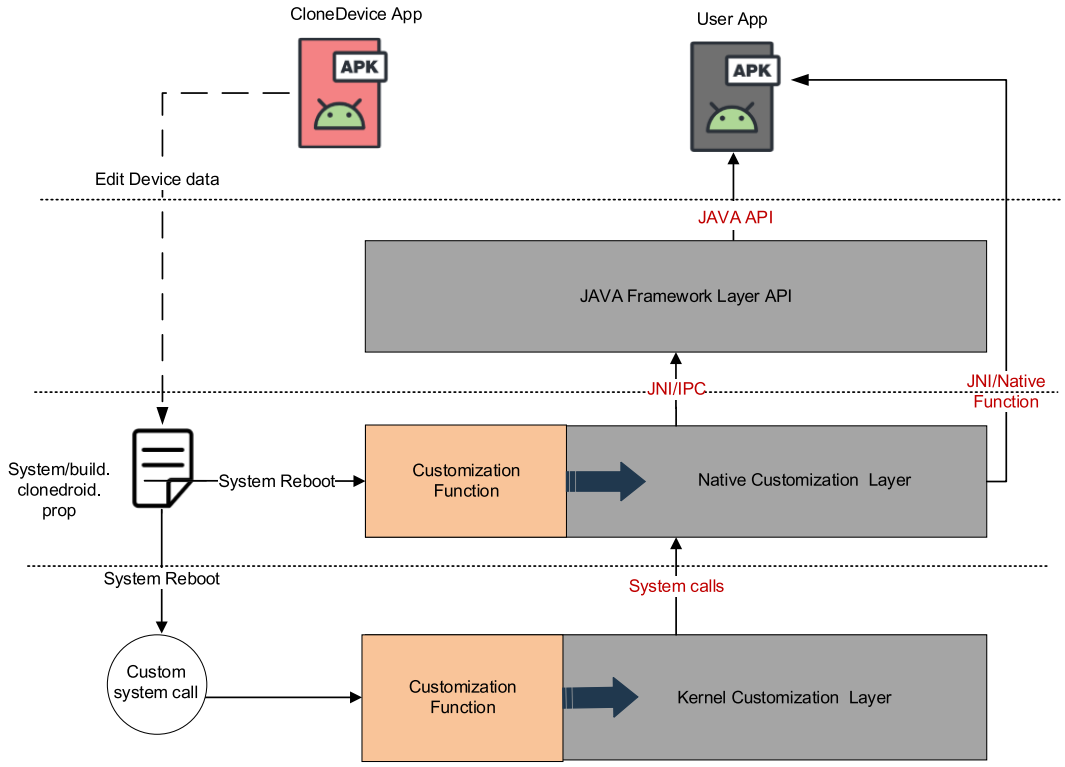


FIGURE 5. CloneDroid users provide device-specific attributes in "build.clonedroid.prop" configuration file. The system's customization happens either in the kernel drivers or the Native code.

TABLE 2. Information of customizable location for data-clone attacks with phone device.

Type	Customizable location	Android system Layer
System Property	Init process	Native
Kernel Version	Kernel proc driver	Kernel
Memory	Kernel proc driver	Kernel
CPU	Kernel proc driver	Kernel
Network	Kernel veth driver	Kernel
Power	Kernel power driver	Kernel
Bluetooth	Bluetooth process	Native
GPS	system_servel process	Native
Telephony	RILD process	Native
Display	Init process	Native
GPU	OpenGL service process	Native

will load the properties into the custom function to wait for the app to access. In practice, the "build.clonedroid.prop" stores device-specific attributes in the form of key-value pairs. We classify key-value pairs in it into two categories: native-layer device properties and kernel-layer device properties. They have different customization methods.

1) ANDROID NATIVE LAYER ATTRIBUTES CUSTOMIZATION

Tab. 2 reveals the customization of the "System Property," "Bluetooth," "GPS," "Telephony," "Display," and "GPU" attributes occur in the native layer of the Android system. Our system function modification includes two items: (1) adding user identifier (UID) condition judgment to filter the current user app; (2) when matching the current UID to the user app, execute the "Customization Function", the return value of this function is the custom device data, otherwise execute the normal flow of the original system function.

Here, we take the customize "System Property" as an example to describe the native layer design of CloneDroid. Android system properties are const values that describe the configuration information of the mobile device, including brand, serial number, device tags, etc. They are stored in the shared memory of the init process. Other processes enquire about Android system properties at run time by calling "property_get," an API for native code to read the data in the shared memory space from other processes. We modify the specific implementation of "property_get" in the system source code. When the phone is started, the customization data is loaded by the customization function. Once the user app calls the "property_get," the function will first determine whether the current query request is from the user app by checking the current user UID. If yes, it calls "Customization Function" to get the customized data from "build.clonedroid.prop" and then returns the fake data to the user app. we did not add the hook code snippet, which reduces the largescale hiding work in order not to expose the hook trace.

2) ANDROID KERNEL LAYER ATTRIBUTES CUSTOMIZATION

Tab. 2 reveals the customization of the "Kernel version," "Memory," "CPU," and "Network" attributes occur in the kernel layer of the Android system. Since the kernel directly interacts with the hardware, once the modification is inappropriate, it will cause the device to crash. Therefore, we filter the applicable objects of the kernel operation after customization. Only when the current user UID is app user, the customization modification will take effect. In addition

to the two custom items in §IV-C1, we need to customize the system call function. The customized data in the system native layer has no privilege to enter the kernel space. One way for these customized data to take effect is to recompile the CloneDroid source code and perform the flashing process. However, if these processes are executed every time update the mobile phone environment, it will make the customization time too cumbersome and inefficient. To overcome this obstacle, we create a new system call to copy data from the userspace to the kernel space. Our customization functions in the kernel drives work in a similar style. For example, we customize the CPU-related profiles (e.g., CPU Serial Number, CPU Version) in the kernel proc driver and use the user UID to determine whether the query request is from the user app or system. If the request is from the user app, it will call our created syscall to extract customized data. Otherwise, the normal system flow will be executed.

3) THE ADVANTAGES OF CloneDroid CUSTOMIZATION

Compared with the app-level customization solution, our OS-level customization solution revels distinct advantages. **First**, our custom method is highly concealed. Our tool does not rely on any user-level hooking mechanism, which means that the user app can not identify custom environments through commonly used hook detection technology. **Second**, CloneDroid offers vast and reliable Android device-attribute editing options. We now provide 101 device configuration options, which span a wide spectrum of device attributes. More importantly, CloneDroid performs reliable custom operations on the native layer and the kernel layer of the Android system, which makes it difficult for the user app to bypass our customization point. As a result, the CloneDroid ensures the user app's consistency to obtain the device attribute value from the system context.

V. EVALUATION

We evaluate the effectiveness of the data-clone attack by cloning the intercepted private data to the attacker's real machine environment, Xposed-based custom environment, and CloneDroid environment and checking the status of the app that restored the login state. The first environment is mainly used to analyze the effectiveness of our basic data-clone attack. The latter two situations focus on the device consistency check features of the app, and CloneDroid is also used to evaluate the app's ability to check the hook sandbox.

We test our proposed data-clone attacks with 175 most-downloaded apps. We crawled the top 300 ranked Android apps from audio, social, and financial categories on Huawei App Market, Xiaomi App Market, and Google Play, with more than one million downloads. In order to make the crawled data set categories names consistent with our attack object types, we did not directly follow the app category name of the application market. The crawled data are reclassified into new types of social media, payment, subscription. Their distributions are shown in Tab. 3, which shows the examples of these compromised apps.

TABLE 3. Total number of test apps and distribution of categories.

Type	Apps	number
Social media	Snapchat, Sina Weibo, QQ, Instagram, Pinterest, WhatsApp, Messenger, Telegram...	90
Payment/banking	Postmark, Pinduoduo, Letgo, iHerb, DiDi, Shpock, OfferUp...	60
Subscription	iQiyi, Netflix, KKBox, Tencent Video, BBC News, Amazon Prime Video,Youku Video...	25
Sum	/	175

TABLE 4. The number of successes when performing data-clone attacks. Xposed-based sandbox and CloneDroid have been configured to match the victim phone's profiles.

	#Apps	Real device	Xposed	CloneDroid
Social media	90	71	80	88
Payment/shopping	60	43	49	59
Subscription	25	21	23	25
Sum	175	135	152	172

A. BASIC ATTACK

We follow the style of Fig. 3 to intercept auto-login depended on data, and then push the intercepted data to the attacker's real device. As a result, the third column of Tab. 4 lists the number of successes when performing data-clone attacks with real devices, which shows the effectiveness of our attack model. We can automatically log into 135 out of 175 apps, including 71 social media apps, 43 payment/shopping apps, and 21 subscription apps. The second column of Tab. 3 shows the examples of these compromised apps, including prominent apps that have been downloaded for more than one billion times (e.g, Facebook, QQ, and Sina Weibo). These apps contain a large amount of user's private information. Once stolen, they will cause some problem (e.g., fake accounts, fake post messages, etc.) to disrupt the healthy life of users. Most importantly, some subscription-based apps break through the paying-subscriber limit during the data-clone attack model.

For the remaining 40 failed cases, when we run them in the new device, they exhibit one of the following responses: 1) the app terminates and exits; 2) the app requests the user to type ID and password again. Many apps also pop up a new window showing that the app is running on a different device. We surmise that these apps have already detected the change of device and thus disabled the automatic login. To confirm our conjecture, we conduct a separate experiment to clone these apps to an Xposed-based sandbox.

B. ADVANCED ATTACK WITH XPOSED-BASED CUSTOMIZATION ENVIRONMENT

In our new experiment, we install an Xposed-based sandbox on our phone (Redmi Note 4x). This sandbox provides 101 device configuration options and we change them to the same profiles with our old phone (Xiaomi Redmi Note 4). In our study, we assume the attacker knows the target device's attributes. Such information can be inferred through social engineering [36], but it is out of our paper's scope.

The ''Xposed'' column of Tab. 4 shows the number of successes when performing data-clone attacks with

TABLE 5. The successful cases that are newly added when performing data-clone attacks in Xposed-based sandbox.

Type	Apps
Social media	LINE, Microsoft Outlook, Douban, Toutiao, Baidu Tieba, Tiktok, Douyin, Wickr, BIGO LIVE.
Payment/banking	Best Buy, NetEase Kaola, Ctrip, 5miles, Geek, KFC.
Subscription	TuneIn Radio, Qingting FM.

an Xposed-based sandbox. Compared to the experiment with real devices, we have 17 compromised apps that are newly added in the app-virtualization environment (see Table 5). Our new experiment confirms that some apps have already secured their auto-login functions by checking device-specific attributes, but their detections can be cheated by app-virtualization technique.

The remaining 23 cases failed to recover the app status in the Xposed-based environment, which shows that some apps may not only detect the consistency of device attributes. One of the directions that can be further evaluated is whether the hook fingerprint is considered abnormal by the app, so we clone the app data to the OS-level custom environment for in-depth evaluation.

C. ADVANCED ATTACK WITH OS-LEVEL CUSTOMIZATION ENVIRONMENT

The OS-level customization environment we propose is CloneDroid. Currently, CloneDroid is compatible with Android 8.0. The CloneDroid images are created and configured on a PC and downloaded to the host device via USB. We provide a CloneDevice app for users to edit device attributes. This app is related to configuration file, which includes device attributes and values. When the users want to update the mobile phone environment, they edit the device attributes values in the CloneDevice app, and these attributes values will eventually be filled in the configuration file. Once the user restarts the mobile phone, the system will load the configuration file information to the corresponding customization point, thereby completing the update of the mobile phone device environment.

We evaluate CloneDroid from two dimensions. The First experiment demonstrates that CloneDroid substantially increase the success rate of data-clone attacks. Second, we provide performance measurements to show that CloneDroid reveals native performance.

1) DEVICE-CONSISTENCY CHECK

We repeat our data-clone attacks with most-downloaded apps in CloneDroid. In particular, we take Google Nexus 6P configure the CloneDroid environment as Xiaomi Redmi Note 4, Redmi Note 4x, Huawei Honor 6x, and Honor 8, respectively. These four phone environments represent four victim devices, and we provide four different device-attribute configuration files for CloneDroid to load. The result is that we can achieve the login status of our cloned apps in each mobile customized environment. The last column of Tab. 4 shows the success number of data-clone attacks in CloneDroid: we can compromise as many as 172 most popular apps' accounts. Compared with the attacks on a real device, CloneDroid wins by additional 37 apps; among them, 20 apps can detect Xposed-based sandbox but fail to detect CloneDroid. Tab. 6 shows the apps that are only vulnerable to data-clone attacks in CloneDroid. An identity theft example for the messaging app KakaoTalk are shown in Fig. 6, and it also demonstrates the unique benefit of our approach. KakaoTalk represents the apps that are only vulnerable to data-clone attacks in CloneDroid (see Tab. 6). As long as the legal user is online, the attacker cannot log into KakaoTalk by typing the same user's ID and password. Furthermore, KakaoTalk also disables the automatic login after we copy the data in the directory of ''/data/data/KakaoTalk/'' to a new device. In contrast, the data-clone attack via CloneDroid enables the victim and the attacker to be online simultaneously, and the victim is not aware that her account has been compromised.

TABLE 6. The app examples that are only vulnerable to data-clone attacks in CloneDroid.

Type	Apps
Social media	WeChat, Chrome, KakaoStory, Tagged, Uplive, Tango, MICO Chat, KakaoTalk.
Payment/banking	Alipay, PayPal, UnionPay, Walmart, Wish, Starbucks, Influenster, HEMA, Groupon, China Railway Customer Service Center.
Subscription	Apple Music, Spotify.

The rest of the three failed data-clone attacks in CloneDroid is from banking apps and social media. Upon further investigation of banking apps, the root cause is they do not store auto-login depended data locally in databases or shared preferences. Instead, they put auto-login depended data in memory. The banking apps take a conservative solution: they store auto-login depended data temporarily in their own process memory for better isolation. This type sacrifices usability and are not well suited for social media and subscription-based apps, because users have to retype their ID and password if they restart the app. Another type (adopted by Youtube, Skype, etc) presents a possible countermeasure to data-clone attacks. Note that for some social media apps that rely on Android ''AccountManager'' APIs to manage the auto-login function (e.g., Youtube, Skype), our attacks failed at first. The reason is ''AccountManager'' stores auto-login depended data under the directory of ''/data/system_xx/'' rather than ''/data/data/[app_name]/''. After we copy the ''/data/system_xx/'' folder to the virtual phone in our second try-out, our data-clone attacks succeeded.

2) PERFORMANCE MEASUREMENTS

The modification of the Android system may cause a loss of system performance. In order to prove that our customization does not cause too serious load on the device, we test the runtime overhead and memory consumption of CloneDroid. We measure performance using Google Nexus 6P phones (ARM Cortex-A53, Adreno 430 GPU, 3G RAM, and 32G ROM). Our runtime overhead measurement contains two scenarios. The first one is running a set of benchmark apps

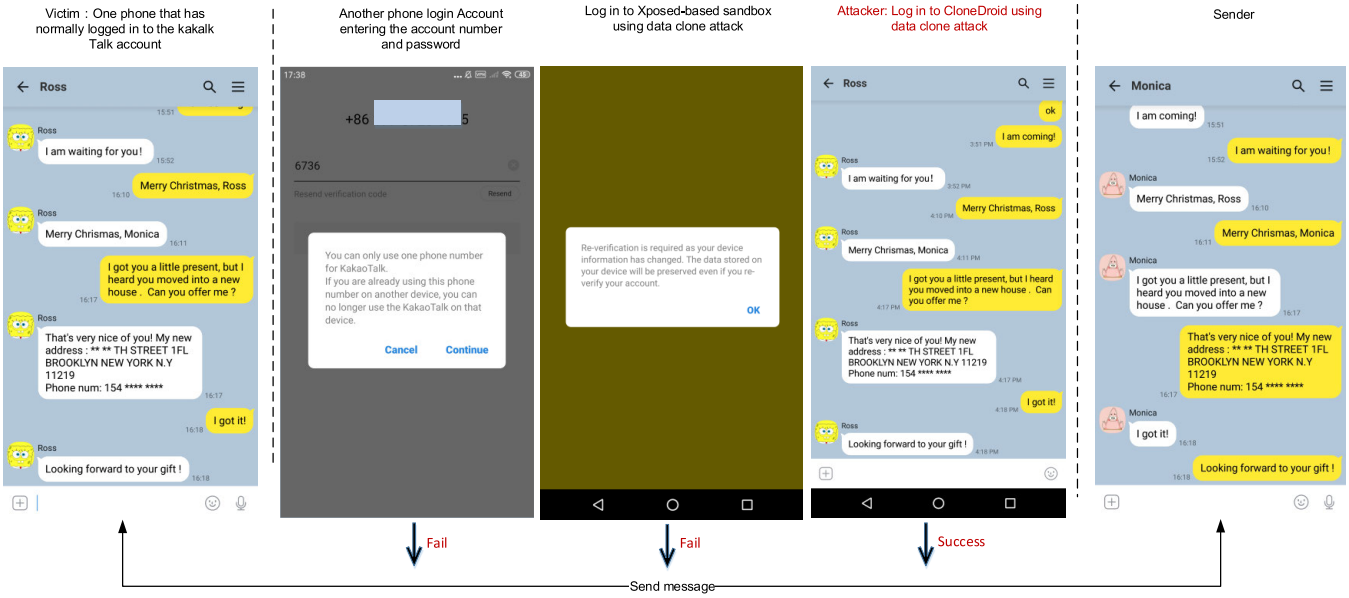


FIGURE 6. Login status of Kakaotalk account under normal conditions and data-clone attacks conditions.

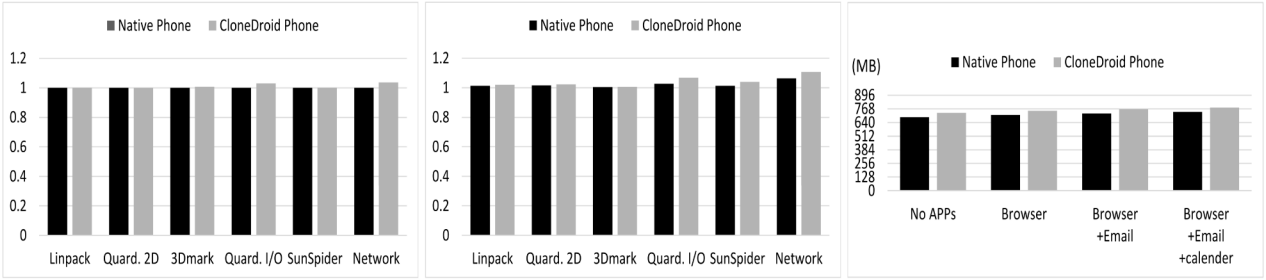


FIGURE 7. CloneDroid's performance measurements. "Quadr." is short for "Quadrant." (a) Normalized Nexus 6p results, (b) Normalized Nexus 6p + music results, (c) Normalized Nexus 6p memory usage in MB.

on CloneDroid's and a native phone, respectively. Native phone refers to a phone running Google Android source code without any modification. The second one is running the same benchmark apps on the CloneDroid and the native phone, but simultaneously with an additional background music player workload. All results are normalized against the performance of running the same benchmark apps on the latest manufacturer stock image available for Google Nexus 6P, but without the background workload. Each benchmark app is designed to stress some aspect of the system performance: Linpack (v1.1) for CPU; Quadrant advanced edition (v2.1.1) for 2D graphics and file I/O; 3DMark (v2.0.4646) for 3D graphics; SunSpider (v1.0.2) for web browsing; and networking using BusyBox wget (v1.21.1) to download a single 500M video file through a PC's Wi-Fi hotspot.

Fig. 7 shows the normalized runtime overhead and memory usage on the Nexus 6P phones. Compared to Native phone, CloneDroid reveals the same level of variability in measurement results. The deviations between "CloneDroid" and "Native Phone" appear in Quadrant I/O, SunSpider, and network in Fig. 7a) and Fig. 7b). Quadrant I/O and SunSpider will consume CPU performance during the testing process. These additional loads are mainly caused by the interaction of the customized information of the CloneDroid device, but the negligible deviations indicate no user-noticeable performance difference between running in CloneDroid and running natively on the phone. The network's difference cannot be attributed entirely to the customized load, because the speed of the network and the blocking of the surrounding obstacles will cause delay to the network.

As shown in Fig. 7c), the increase in the memory difference between CloneDroid and Native phone mainly comes from the first "No APPs" scenario. This scenario describes the memory occupied by the phone's pure system startup. Since CloneDroid starts, it needs to read the configuration file of the device properties and fill it into the memory of the corresponding custom point for the user app to call. It exceeds the memory consumption of 40M than the Native Phone, which is tolerable for Android Mobile phones. The memory usage of most apps now exceeds 100M. In "Browser"

and "Browser+Email +calender" scenes, Native phones and CloneDroid's memory growth rate are almost the same, and these memory growth groups come from the user app's own memory consumption. On the whole, although we have customized most of the system, but it does not have a big impact on the consumption of system memory.

a: VENDOR REACTION

In total, we received responses from 18 vendors that confirmed vulnerabilities. Among them, seven app vulnerabilities confirmed from the china national vulnerability database (CNVD) [37] platform, and the remaining vulnerabilities of the apps were confirmed from the vulnerability submission platform or the reply email of the manufacturer. Vendors such as Alibaba, Tencent, and iQiyi have labeled our findings as high/middle-severity vulnerabilities. We speculate that vendors are more susceptible to paying-subscriber fraud. For example, iQiyi, an online video app with more than 100 million users, has labeled our report as a high-severity vulnerability and added device-consistency checks in the new release version. However, we have evaluated the latest version of iQiyi in CloneDroid and found that CloneDroid can still bypass the newly added device-consistency checks. Netflix confirms our vulnerability finding, and they treat it as a "Single-User Fraud" threat. We confirm that with the latest version of Netflix (Netflix 7.52.0), our data-clone attack without CloneDroid still succeeds.

VI. MITIGATION DISCUSSION

Related work [6]–[8] has provided several methods to mitigate data-clone attacks. For example, user credential data are bound to a specific device, increasing the search time of locating user credential data, and asking users for additional PIN input whenever an automatic login occurs. However, these mitigation methods either can be evaded by CloneDroid, or are too cumbersome in practice. The most fundamental method against data-clone attacks is that a mobile app never stores auto-login depended data in local files. One direction is, like our tested banking apps, to store auto-login depended data temporarily in the app's process memory. However, this strategy, at the cost of sacrificing usability, only works for critical apps that do not require frequent user interactions. Another direction is to leverage ARM TrustZone to encrypt/decrypt auto-login depended data before use. As the decryption key is stored in the Trustzone environment, data-clone attacks cannot copy the decryption key to another device together with the encrypted auto-login depended data, and therefore the server will fail to verify the login credentials. The recent papers, TruApp paper [38] and IM-Visor [39] explore the feasibility of protecting App integrity and sensitive data with TrustZone.

A natural response to breaking through the login-device number limit is to monitor concurrent sessions at the app server side. Unfortunately, the variable nature of mobile devices (e.g., the switch of Wi-Fi hotspot and cellular data) makes it difficult to determine an adequate number of concurrent sessions. The previous work [7] has pointed out that, although many apps do not permit duplicate logins from different devices, they do allow multiple session requests from the same device ID. Our evaluation also confirms that most apps allow maintaining two or more connections per user. As the login from CloneDroid shows a different IP address from the victim's IP, a possible countermeasure is to detect multiple concurrent IPs at the server-side. However, this strategy cannot completely thwart data-clone apps. For quite several apps, such as the Facebook app we tested, they do allow multiple logins from different devices.

We do not assume that evading CloneDroid is strictly impossible, but it can prohibitively increase the cost. CloneDroid now provides 101 device configuration options, but we cannot guarantee that our list is complete. The arms race here is that the auto-login function could check the consistency of some obscure device properties, and finding all of them is an open problem. Our CloneDroid is susceptible to the new update and replacement of hardware devices in future Android versions. In addition, the development of CloneDroid is based on a specific Android version and cannot be extended to multiple Android versions. In the future, we plan to expand CloneDroid so that it can be flexibly transplanted into multiple Android versions in the form of firmware.

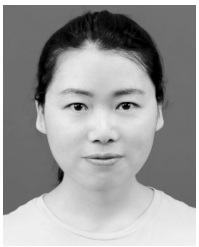
VII. CONCLUSION

In this article, we present a data-clone attack based on the vulnerability of an OEM-phone clone app. This attack addresses some of the limitations of the previous work and can bypass most apps' user authentication. Besides, we also propose an advanced attack that can break the device consistency check of almost all tested apps. Especially, the advanced attack scheme with an OS-level custom environment is more transparent to cloned apps, making most apps insensitive to environmental changes. It reveals a strong resilience to various device-consistency checks within the clone apps. Our systematic study with most-downloaded apps demonstrates that the data-clone attack has been swept under the carpet for a long time, and our proposed attack is an imminent threat. We hope our study can help the community redesign the auto-login feature and improve the runtime environment checking capability.

REFERENCES

[1] T. Tran. (Feb. 2020). *20 of the Best Social Media Apps for Marketers in 2020*. [Online]. Available: https://blog.hootsuite.com/best-social-media-apps-list/
[2] S. J. Vaughan-Nichols, "Will mobile Computing's future be location, location, location?" *Computer*, vol. 42, no. 2, pp. 14–17, Feb. 2009.
[3] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner, "Smart locks: Lessons for securing commodity Internet of Things devices," in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur. (ASIACCS)*, 2016, pp. 461–472.
[4] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.
[5] W. Yang, Y. Zhang, J. Li, H. Liu, Q. Wang, Y. Zhang, and D. Gu, "Show me the money! Finding flawed implementations of third-party in-app payment in Android apps," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[6] J. Cho, D. Kim, and H. Kim, "User credential cloning attacks in Android applications: Exploiting automatic login on Android apps and mitigating strategies," *IEEE Consum. Electron. Mag.*, vol. 7, no. 3, pp. 48–55, May 2018.
[7] J. Choi, H. Cho, and J. Yi, "Personal information leaks with automatic login in mobile social network services," *Entropy*, vol. 17, no. 6, pp. 3947–3962, Jun. 2015.
[8] S. Park, C. Seo, and J. H. Yi, "Cyber threats to mobile messenger apps from identity cloning," *Intell. Autom. Soft Comput.*, vol. 22, no. 3, pp. 379–387, Jul. 2016.
[9] iStarsoft. (Jul. 2020). *Top Three Phone Clone App To Copy Phone Data in 2020*. [Online]. Available: https://www.android-data-recovery.org/phone-clone-app.html
[10] *Xposed Module Repository*. [Online]. Available: https://repo.xposed.info/
[11] CVE. (2020). *Common Vulnerabilities and Exposures*. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-5573
[12] I. L. Kim, Y. Zheng, H. Park, W. Wang, W. You, Y. Aafer, and X. Zhang, "Finding client-side business flow tampering vulnerabilities," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Jun. 2020, pp. 222–233.
[13] Android. (Mar. 2019). *Android Security & Privacy 2018 Year in Review*. [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2018 _Report_Final.pdf
[14] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, "FIRMSCOPE: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in Android firmware," in *Proc. 29th USENIX Secur. Symp. (USENIX)*, 2018, pp. 1–19.
[15] C. Lee. (Jan. 2019). *Daily Active Users for WeChat Exceeds 1 Billion*. [Online]. Available: https://www.zdnet.com/article/daily-active-user-of-messaging-app-wechat-exceeds-1-billion/
[16] A. Bianchi, E. Gustafson, Y. Fratantonio, C. Kruegel, and G. Vigna, "Exploitation and mitigation of authentication schemes based on device-public information," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, Dec. 2017, pp. 16–27.
[17] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection: Countering the Largest Security Threat*. 2008.
[18] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "TriggerScope: Towards detecting logic bombs in Android applications," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 377–396.
[19] AppBrain. (Jan. 2020). *Top Android Phone Manufacturers*. [Online]. Available: https://www.appbrain.com/stats/top-manufacturers
[20] A. Developers. *Data Backup Overview*. [Online]. Available: https://developer.android.google.cn/guide/topics/data/backup.html
[21] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, and M. Grace, "PARTEMU: Enabling dynamic analysis of real-world trustzone software using emulation," in *Proc. 29th USENIX Secur. Symp. (USENIX)*, 2020, pp. 1–18.
[22] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of Android malware," in *Proc. 7th Eur. Workshop Syst. Secur. (EuroSec)*, 2014, pp. 1–6.
[23] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu, "Morpheus: Automatically generating heuristics to detect Android emulators," in *Proc. 30th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2014, pp. 216–255.
[24] J. Gajrani, J. Sarswat, M. Tripathi, V. Laxmi, M. S. Gaur, and M. Conti, "A robust dynamic analysis system preventing SandBox detection by Android malware," in *Proc. 8th Int. Conf. Secur. Inf. Netw. (SIN)*, 2015, pp. 290–295.
[25] L. Bordoni, M. Conti, and R. Spolaor, "Mirage: Toward a stealthier and modular malware analysis sandbox for Android," in *Proc. 22th Eur. Symp. Res. Comput. Secur. (ESORICS)*, 2017, pp. 278–296.
[26] I. Pustogarov, Q. Wu, and D. Lie, "*Ex-vivo* dynamic analysis framework for Android device drivers," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1088–1105.
[27] asLody. (2016). *VirtualApp*. [Online]. Available: https://github.com/asLody/VirtualApp
[28] C. Zheng, T. Luo, Z. Xu, W. Hu, and X. Ouyang, "Android plugin becomes a catastrophe to Android ecosystem," in *Proc. 1st Workshop Radical Experiential Secur. (RESEC)*, 2018, pp. 61–64.
[29] L. Shi, J. Fu, Z. Guo, and J. Ming, "'Jekyll and Hyde' is risky: Shared-everything threat mitigation in dual-instance apps," in *Proc. 17th Annu. Int. Conf. Mobile Syst., Appl., Services (MobiSys)*, 2019, pp. 222–235.
[30] L. Zhang, Z. Yang, Y. He, M. Li, S. Yang, M. Yang, Y. Zhang, and Z. Qian, "App in the middle: Demystify application virtualization in Android and its security threats," in *Proc. ACM Meas. Anal. Comput. Syst. (SIGMETRICS)*, 2019.
[31] S. Whalen. (Apr. 2001). *An Introduction to ARP Spoofing*. [Online]. Available: http://chocobospore.org/arpspoof
[32] M. A. Qadeer, A. Iqbal, M. Zahid, and M. R. Siddiqui, "Network traffic analysis and intrusion detection using packet sniffer," in *Proc. 2nd Int. Conf. Commun. Softw. Netw.*, 2010, pp. 313–317.
[33] R. Harris. (Nov. 2018). *The Subscription Based App Model is Working and Here's Proof*. [Online]. Available: https://appdevelopermagazine.com/the-subscription-based-app-model-is-working-and-here's-proof/
[34] M. V. Vorst. (Jul. 2019). *Download Source Code*. [Online]. Available: https://source.android.com/setup/downloading
[35] Aethaellyn. (Apr. 2019). *Detection of Xposed Framework*. [Online]. Available: https://programmer.group/detection-of-xposed-framework.html
[36] C. Hadnagy, *Social Engineering: The Sci. Human Hacking*, 2nd ed. Hoboken, NJ, USA: Wiley, Jul. 2018.
[37] *China National Vulnerabilty Database*. [Online]. Available: https://www.cnvd.org.cn/
[38] S. Demesie Yalew, P. Mendonca, G. Q. Maguire, S. Haridi, and M. Correia, "TruApp: A TrustZone-based authenticity detection service for mobile apps," in *Proc. IEEE 13th Int. Conf. Wireless Mobile Comput., Netw. Commun. (WiMob)*, Oct. 2017, pp. 1–9.
[39] C. Tian, Y. Wang, P. Liu, Q. Zhou, C. Zhang, and Z. Xu, "IM-visor: A pre-IME guard to prevent IME apps from stealing sensitive keystrokes using TrustZone," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2017, pp. 145–156.



WENNA SONG received the M.S. degree in software engineering from the Wuhan University of Technology. She is currently pursuing the Ph.D. degree in cyberspace security with Wuhan University. Her research interests include mobile system security and privacy preserving.

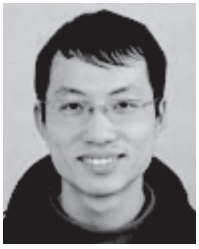
MING JIANG received the Ph.D. degree in information sciences and technology from The Pennsylvania State University, in 2016. He is currently an Assistant Professor with the Computer Science and Engineering Department, UT Arlington. He does research on software/system security, including binary code analysis and verification for security issues, hardware-assisted malware analysis, and mobile system security. His research has been funded by the National Science Foundation (NSF).

HAN YAN received the bachelor's degree in electronic information engineering from Sichuan University, in 2019. He is currently pursuing the master's degree in cyberspace security with Wuhan University. His research interests include the IoT security and automatic exploit generation.

YI XIANG is currently a Senior Student with the School of Cyber Science and Engineering, Wuhan University, Wuhan, China. Her main research interests include privacy preserving and system security.

YUAN CHEN is currently a Senior Student with the School of Cyber Science and Engineering, Wuhan University, Wuhan, China. His main research interests include system security and malicious code analysis.

YUAN LUO received the bachelor's degree from Wuhan University, where he is currently pursuing the Ph.D. degree with the School of Cyber Science and Engineering. His research interests include cyber-physical systems (CPS) security and mobile security.



KUN HE received the Ph.D. degree in computer science from the Computer School, Wuhan University. He is currently an Associate Professor with Wuhan University. His research interests include cryptography, network security, mobile computing, and cloud computing. He has published more than 20 research papers in many international journals and conferences, such as IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE TRANSACTIONS ON MOBILE COMPUTING, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON COMPUTERS, USENIX Security, and INFOCOM.

GUOJUN PENG received the Ph.D. degree from Wuhan University, in 2008. He is currently a Professor with the School of Cyber Science and Engineering, Wuhan University. His research interests include system security and trust computing. He is a member of CCF and CSAC.