Android on PC: On the Security of End-user Android Emulators

Fenghao Xu[1], Siyu Shen[1], Wenrui Diao[2,3], Zhou Li[4], Yi Chen[1], Rui Li[2,3], and Kehuan Zhang[1*]
[1]The Chinese University of Hong Kong
xf016@link.cuhk.edu.hk, {ss019,yichen,khzhang}@ie.cuhk.edu.hk
[2]School of Cyber Science and Technology, Shandong University, diaowenrui@sdu.edu.cn, leiry@mail.sdu.edu.cn
[3]Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University
[4]University of California, Irvine, zhou.li@uci.edu

ABSTRACT

Android emulators today are not only acting as a debugging tool for developers but also serving the massive end-users. These end-user Android emulators have attracted millions of users due to their advantages of running mobile apps on desktops and are especially appealing for mobile game players who demand larger screens and better performance. Besides, they commonly provide some customized assistant functionalities to improve the user experience, such as keyboard mapping and app installation from the host. To implement these services, emulators inevitably introduce communication channels between host OS and Android OS (in the Virtual Machine), thus forming a unique architecture which mobile phone does not have. However, it is unknown whether this architecture brings any new security risks to emulators.

This paper performed a systematic study on end-user Android emulators and discovered a series of security flaws on communication channel authentication, permission control, and open interfaces. Attackers could exploit these flaws to bypass Android security mechanisms and escalate their privileges inside emulators, ultimately invading users' privacy, such as stealing valuable game accounts and credentials. To understand the impact of our findings, we studied six popular emulators and measured their flaws. The results showed that the issues are pervasive and could cause severe security consequences. We believe our work just shows the tip of the iceberg, and further research can be done to improve the security of this ecosystem.

CCS CONCEPTS

• **Security and privacy → Systems security**.

KEYWORDS

Android Emulator; Security Assessment

ACM Reference Format:
Fenghao Xu, Siyu Shen, Wenrui Diao, Zhou Li, Yi Chen, Rui Li, and Kehuan Zhang. 2021. Android on PC: On the Security of End-user Android Emulators. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3460120.3484774

*Kehuan Zhang is the corresponding author. Part of Fenghao Xu's work was done when visiting University of California, Irvine.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484774

1 INTRODUCTION

Android emulator creates virtual Android devices on the host computer, enabling users to run Android apps on their desktops. Android developers have been using emulators shipped with Android Studio to test their apps [11], but in recent years the Android emulators designed especially for end-users are gaining traction. These emulators are designed to make better use of the powerful CPU/GPU performance of the host (usually a Windows PC) and provide many assistive functionalities like keyboard mapping to facilitate users' tasks. Compared with mobile phones, users can enjoy a larger screen and higher performance when running Android apps. Gamers especially enjoy these emulators, as they can use mouse and keyboard to perform gaming actions more quickly and precisely, and at the same time avoid the issues of limited battery and over-heating on mobile phones. As a result, end-user emulators have already gained a large user base. For instance, BlueStacks [15] alone has 400 million users worldwide.

On the other hand, the Android emulators could process users' sensitive data, which makes them lucrative targets to attackers. For example, when using the emulators, the users might need to login to their Google accounts to download apps from the Google Play Store (pre-installed in emulators). The gamers might have in-game purchases associated with their bank/payment account. At least, the Android emulators should not weaken the security guarantees from Android OS. However, so far, there has been no systematic study on the security of end-user Android emulators, and we found the term "emulation" could be giving people a false sense of security [50]. Therefore, we are motivated to study the security problems of emulators and demonstrate the importance of emulator security.

New Security Risks. Most emulators run on top of the Virtual Machine (VM), which virtualizes the computing environment and the underlying hardware resources. The actual Android OS is placed in the VM to support and manage the installed mobile apps. To provide a better experience, the emulator usually separates the above VM process from its main UI process which users actually interact with. In this way, it is inevitable to introduce a communication channel between the above processes, i.e., between the host-side main UI process and guest-side Android OS. For example, to provide a seamless user experience, users can use their input method (IME) tool on the host OS when typing, and the emulator will encapsulate the completed text and send it to Android.

Moreover, since the underlying VM only supports raw mouse and keyboard inputs natively, this IME text is usually transmitted through the customized channel introduced by emulator vendors. Besides, the implementation of gaming assistant tools and other functionalities also rely on that channel, which we will illustrate in Section 2.2.

In practice, the emulator developers design the communication architecture through customizing Android firmware, such as modifying system components and adding new Android services. Such customization might not be securely designed and scrutinized. Although researchers have studied security issues regarding Android customization on mobile phones [26, 27, 66, 82], we note that their methods can not be simply applied to emulators because emulators have different, more complicated structures and support new functionalities. Therefore, it is unclear what security requirements should be satisfied, what are the security and privacy implications behind such new emulator services, and whether their unique architecture brings new risks to the emulator system.

Our Work. To bridge the research gap, we conducted a systematic study on popular Android emulators. As a platform to host various third-party apps, emulators should always comply with the existing Android security model (e.g., the separation between apps and permission protection on the sensitive resources) when customizing the system. Yet, we reached an opposing conclusion: we found that, when designing their custom services, the emulator vendors usually make incorrect assumptions about the adversaries' capabilities (Section 3.2), which lead to inadequate protection on emulator services. Hence, the attacker can abuse the customized components to launch a variety of high-risk attacks.

We investigated 6 popular emulators and tested their components related to key functionalities (e.g., user input handling). We found that they all suffered from different security issues. The root cause is that emulators do not adequately protect their security-sensitive communication channel, which is utilized to transmit commands and data from the host to Android or from Android to the host and open the door to *confused-deputy* attacks. For example, by abusing the IME text channel, a malicious Android app located in the same emulator can inject any text to Android (details in Section 4.1). As another example, attackers can launch a *race condition* attack to hijack the app installation process (Section 4.2) and finally install an arbitrary app. In another case, we can manipulate the Windows tabs to launch a phishing attack (Section 4.3). Even worse, a malicious app can gain system privilege (e.g., `root`) without user consent, which will change the app into "God Mode" and enable the adversary to do nearly anything within the emulator (Section 4.4). These vulnerabilities are rooted in different communication channels, including TCP/UDP, virtual device, and Android IPC (inter-process communication).

Measurement Result. We inspected six popular emulators to understand how they implemented the above functionalities and whether they are vulnerable to our attacks. The results (in Table 4) show that all of them are flawed and vulnerable to some of our concrete attacks. We have uploaded our attack demos at https://sites.google.com/view/emulatorsec.

Contributions. In summary, we make the following contributions in this paper.

- *New Understanding.* We systematically studied and uncovered the typical architecture of Android emulators, which helps in understanding the security model and identifying new vulnerabilities in this ecosystem.
- *New Flaws and Attacks.* We reported a series of flaws associated with the emulator features. By exploiting them, we constructed five practical attacks and revealed realistic security and privacy risks.
- *Measurement and Discussions.* We measured the scope and magnitude of the flaws by inspecting each emulator and testing our attacks, which showed their broad impacts and severe consequences. We also suggested some mitigation approaches for building a more secure emulator system.

Roadmap. The rest of this paper is organized as follows. Section 2 introduces relevant background on Android emulators. Section 3 shows the fundamental causes and the threat model of the attacks. Section 4 details the exact design flaws regarding each functionality and shows our attacks. In Section 5, we measure the implementations of four emulator services and corresponding attacks across six emulators. Then we give some suggestions on mitigation and discuss future works in Section 6. Section 7 summarizes related works and Section 8 concludes this paper.

2 BACKGROUND

In this section, we introduce the relevant background of the Android emulator. We first present an overview of different types of emulators with their target user groups. Next, we describe the typical emulator architecture which we learned through reverse engineering.

2.1 Android Emulator Types

The Android emulators can be divided into two categories: for developers and for end-users. The former ones are mainly leveraged to assist app development or perform automated app testing. For example, Genymotion [18] contains both desktop and cloud development environment that enables parallel testing and remote monitoring on different Android versions. Android Studio [11] integrates the official emulator from Google for testing and debugging without leaving the IDE. Though the developer-oriented emulators could be vulnerable, we did not investigate them in this work since they usually run a single app under the debugging environment and do not involve private information. Yet, their security needs to be examined when they are used by end-users, and we acknowledge this limitation in our study scope in Section 6.3.

Different from the developer-oriented emulators, the end-user emulators aim to enhance the user experience with Android apps, by taking advantage of the larger screen, better GPU performance, and physical keyboards on PCs. It is particularly favored to play mobile games and live-stream the gameplay (e.g., by YouTubers) [5]. We focus on the security of end-user emulators, because they are much more likely to carry sensitive information from the users[1]: 1) When game apps are played on the emulator, the user will log in with her game credential, which can be linked to in-game purchases and become valuable if traded. In fact, there have been a

[1]The terms "Android emulator" and "emulator" in the following text refer to end-user Android emulators for brevity.

Table 1: Popular Emulators.

Name	VM	Host OS	Downloads	Languages
BlueStacks	VB	Win, macOS	400M	16
LDPlayer	VB	Win	100M+	15+
MEmu	VB	Win	100M	21
MuMu	VB	Win, macOS	5M	5
NoxPlayer	VB	Win, macOS	150M	20
GameLoop	VB/self*	Win	500M	11

VB: VirtualBox
*: It has self-developed VM engine co-located with VB, which runs apps from GameLoop's own app store.

number of attack incidents targeting game credentials [29, 45, 46]. 2) Beyond the game credentials, the user might need to input other sensitive information to use the app, like the Google Play credential to download the app and the social network credentials to share the gaming activities. 3) Beyond gaming, other utility apps are also provided in the app stores and supported by these emulators, like the Facebook app, which are also under threat when the emulators are vulnerable.

In this paper, we select six end-user emulators based on their popularity [8], usability, and whether they are actively maintained (regularly updated). Table 1 lists these emulators and their download count as claimed on their websites [15, 16, 20, 22, 24, 51]. It can be seen that they are targeting a global user base and have received millions of downloads, but through this study, we found they all introduce severe security and privacy risks. Note that although the underlying Virtual Machine (VM) of all chosen emulators happened to be based on VirtualBox, the discovered flaws in this paper originated from the vendor customization at Android-layer and are orthogonal to the specific VM-layer implementation. One example is Gameloop. Although it has a version with a self-designed engine rather than VirtualBox, we actually found both versions have the same flaws (e.g., overly-opened ADB as described in Section 4.4).

2.2 Emulator Architecture

Understanding the architecture of the emulators is an essential step prior to the security analysis. However, none of the emulator vendors provide detailed documents for their architectural designs. Hence, we obtain such insider information from reverse-engineering and reading the hypervisor documents (VirtualBox documentation [23] in particular). The high-level emulator architecture is illustrated in Figure 1.

First, the emulator needs to host Android OS within a virtualized environment. We found most emulator vendors leverage open-sourced VirtualBox [23] as their underlying hypervisor. The other options, such as the self-developed AOW engine on GameLoop [24] and lightweight container scheme like AnBox [6], are rarely seen.

The hypervisor will create a VM to host the Android OS image, which is customized from Android Open Source Project (AOSP) [10] by the emulator vendors. The VM working process (called *VMCore* process in this paper) usually runs in the *headless* mode [70] (in the background) and talks to another process of the host (called

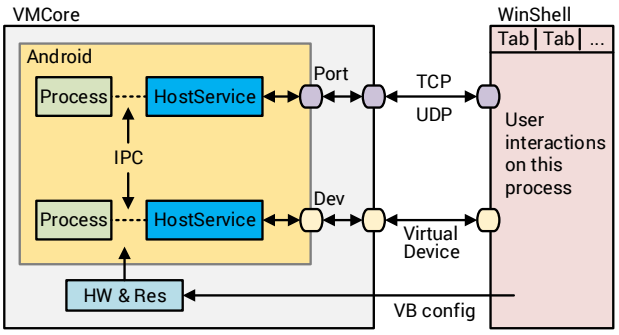

Figure 1: Emulator Architecture. ("HW & Res" are short for "Hardware & Resources".)

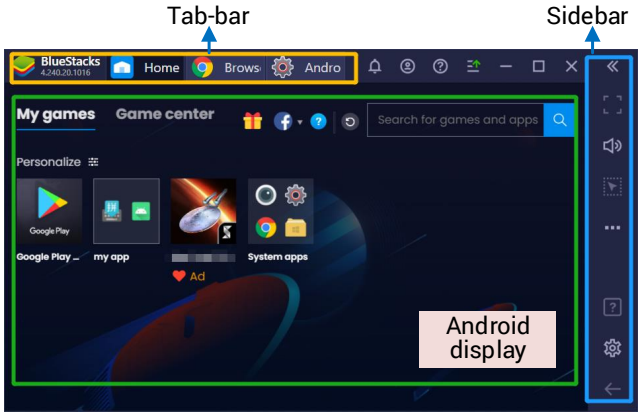

Figure 2: Emulator Example - BlueStacks.

WinShell process when the host is Windows[2]) managed by the emulator. *WinShell* shows the main UI of the emulator, handles user interactions, and commands *VMCore*. An example of the emulator UI is shown in Figure 2, which integrates the app tab-bar and button sidebar, and shows the Android display in the foreground.

The tab-bar shows the labels (app name and icon) of current and recent Android apps (see Figure 2). By interacting with the tabs, users can conveniently switch or close apps to achieve the same effect as using the "back", "home", and "recent" navigation buttons. The sidebar usually contains shortcuts to perform utility functions like taking screenshots and installing apps. Besides, there is also a settings menu on the windows UI, through which users can adjust the emulator configurations (e.g., the CPU/memory allocation to Android and notifications on/off).

Host-Guest Communication. To perform the aforementioned functionalities, *WinShell* is designed to forward the relevant commands and (or) data to *VMCore*, let *VMCore* transport the commands to Android OS, and relay the feedback from Android. Take the "install" shortcut on the sidebar as an example. The user can select an APK file from Windows and install it on Android. To do so, *WinShell* will send the selected file and an "install" command

[2]We focus on Windows in this paper due to its more extensive user base. Unless otherwise indicated, "host" always refers to Windows in this paper.

to Android. A daemon within Android (called *HostService* in this paper) will receive the command and initiate the installation process (e.g., running `PackageInstaller` in Android). The daemon could be a Linux native program written by C/C++ or an Android app written by Java. It usually has the high privilege and (or) system-level permissions. Our analysis of the 6 emulators discovered different host-guest channels, including TCP/UDP, Virtual Device, VirtualBox (VB) Configurations, and Shared Folders.

• *TCP/UDP.* Given that host-guest communication has to bypass the VM sandbox restriction of the Virtual Machine, it is a convenient approach to treat the host and guest as two network entities and exchange messages through TCP/UCP. By configuring a set of *port-forwarding rules* on VB, the ports on *VMCore* process could communicate with the ports of *WinShell*, and forward the host messages to Android, as shown by the upper path in Figure 1. Simultaneously, the VM's NAT network is also configured such that Android views Windows as another peer machine inside a *local network*. As a result, Android can also send packets to Windows ports, just by specifying the virtual "IP address" of the Windows machine as configured by the network adapter.

• *Virtual Device.* The virtual device [69] supported by VB provides another channel for host-guest communication (shown as the middle path in Figure 1). Android inside *VMCore* can add customized virtual devices and make them accessible to the *WinShell* with VB APIs. These devices are shown as ordinary Linux devices on Android, so they can also be read/written by Android processes.

• *VB Configuration and Shared Folders.* We note that hypervisors like VB can adjust the guest OS with configurations, such as the number of occupied CPU cores, RAM size, Network Adapter preferences (shown as "HW & Res" in Figure 1), although this channel is unlikely to carry the app's data. VB's shared folder feature is also used by the host and Android to exchange files, such as photos, videos, and application packages.

3 PROBLEM OVERVIEW AND ANALYSIS

In this section, we first describe the threat model under the emulator setting and the scope of this study. Then, we formalize the security guarantees that should be met by the emulator and show how they can be violated under the new communication channels introduced by the emulator.

3.1 Threat Model

In this study, we focus on the end-user Android emulators and assume the host OS is Windows. We assume the emulator itself and the host OS (and host apps) are both benign. During the attack, we assume a malicious app disguised as a benign app has been installed onto the emulator's Android, but it does not ask for any sensitive, dangerous, or system-level permission, e.g., `GET_ACCOUNTS`. The attacker can deliver the malicious app to users' emulators by uploading it to app stores, like the official store (i.e., Google Play) and the 3rd-party stores (e.g., Nox App Center of NoxPlayer). These app stores are usually bundled with the emulators. To make the malicious app innocuous and trick the victim users to install it, the attacker can integrate the malicious code into repackaged apps or malicious SDKs [59]. Noticeably, previous works on studying Android security often follow the same threat model [28, 47]. Once the malicious app is launched, it could run in the background by launching an Android Service [33], and exploit the design flaws of the emulators to break their security guarantees (elaborated in Section 3.2). The attacker's goals are similar to the previous attacks targeting Android apps, like stealing user's credentials and selling them on the underground market [62].

In addition to the malicious-app (within the emulator) threat model, host-side attacks are also possible. For example, the IPC mechanism of BlueStacks used to be vulnerable to webpages that are loaded on the host and launching DNS rebinding [2]. Since some of our attacks also exploit the same channels like HTTP, we expect they can be performed on the host side as well. We discuss this setting in more detail in Section 6.3.

Problem Scope. This paper focuses on the design and implementation loopholes related to the emulator's architecture and customized services. Since we assume the attacker only owns an app, the attacks compromising the components outside the Android OS (e.g., the supply-chain attack compromising the emulator's code base [61]) are out of scope. The vulnerabilities related to the standard components of Android OS (e.g., framework library), other pre-installed apps (e.g., Chrome Browser) are also out of scope. In Section 6.3, we further discuss the scope.

3.2 Security Analysis

Security Guarantees. We first summarize the security guarantees that the Android emulator should offer. Essentially, the Android emulator should inherit the security guarantees from both Android OS and VM sandbox.

- **SG#0.** The apps running in the emulator should not be able to escape the VM sandbox and attack the hypervisor/host.
- **SG#1.** The apps inside the emulator should be properly separated from each other and the system.
- **SG#2.** The emulator should keep high-risk preferences turned off by default and notify users promptly when the high-profile resources are accessed (e.g., developer mode, accessibility service [30, 31]).
- **SG#3.** The emulator should faithfully forward the data and commands between the Windows UI (i.e., *WinShell* process) and the Android. In other words, the UI integrity should be preserved.

Attack Surface. Since the end-user emulator is mainly developed to enhance user's experience with Android apps (as summarized in Section 2.1), we expect the changes to the underlying architecture and policies of VirtualBox and Android are minimized. Hence, the security guarantees provided by Android and VM sandbox should be preserved, e.g., VM isolation for **SG#0** and **SG#3**, Linux access control [12] and Android permission mechanism [32] for **SG#1** and **SG#2**.

Yet, as described in Section 2.2, the new host-guest communication channels introduce new ways for an app to interact with other components, and their designs might not be secure. One mistake that developers usually make in designing the communication channels is forgetting to verify the *authenticity* and *integrity* of the sender/receiver, which leads to the *confused-deputy* attacks. Taking the commands from *WinShell* as an example, though they are expected to be sent from the host, an app inside Android can

also access the network ports/devices/folders and thus send the same commands. The missing checks on *HostService* could let an attacker app impersonate a user and issue arbitrary commands, including the ones requiring high privilege. On the other hand, building a secure host-guest communication channel is a known challenging task, since customized features are to be supported and they do not always share the same communication protocol.

Therefore, we are motivated to analyze the communication mechanisms of the emulator and test if they can be abused to break the security guarantees. Following the emulator architecture shown in Figure 1, we analyzed the TCP/UDP channel, virtual device channel, and the Android local IPC channel that bridges apps internally.

• *TCP/UDP.* Since an app inside the emulator can be assigned with TCP/UDP ports and the Windows host has an accessible IP address, an app can connect to *WinShell* by using the host's IP with *WinShell*'s ports on Windows, or connect to *HostService* through its ports on Android. We found the ports are often *fixed* after the services/emulators are restarted, which avoids the attack overhead in guessing the ports. Although Android doesn't allow apps to sniff TCP/UDP traffic (except with root privilege), it is lenient when an app *generates* TCP/UDP traffic (only `android.Internet` permission under the `normal` protection level is needed and it is automatically granted after app's installation). Though servers on the Internet can authenticate the clients through application-layer protocols like TLS and mechanisms like certificates, we found none of the emulators enforce such protection or encrypt the data/messages in this channel. A previous study investigated the risks of open ports by Android apps [72], and we extend this direction to emulators.

• *Virtual Device.* Virtual devices are usually exposed to Android OS and apps as Linux device nodes under `/dev`. Though the virtual device can be protected by Linux file permissions, previous studies have shown that the permissions are not often set at the right granularity, which opens the door to the attacker [82]. Sometimes, the device driver on the Linux kernel might have extra verification, e.g., involving complex protocols and synchronization procedures to mediate the access. However, this is not a fundamental hurdle for the attackers as long as they reverse-engineer these mechanisms.

• *Local IPC.* If neither of the above host-guest interfaces is open to the malicious app, the attacker still has a chance to exploit the local IPC channels (if they exist) to compromise the communication paths. As shown in Figure 1, upon receiving messages from the host, *HostService* can execute the corresponding commands by itself, or act as a hub to delegate the commands to another Android process through local IPC. Depending on the emulator's implementation, this local IPC can be on the Android level, e.g., Service Intent or Broadcast [19], or on a lower level, such as another TCP/UDP channel. It is difficult to achieve comprehensive protection on the local IPC channel, as shown in prior works studying Android IPC security [36, 64]. The situation becomes more complicated when the emulator introduces non-standard ways for IPC. For instance, we found some emulators use customized system properties (i.e., `SystemProperty` [17] to store key-value pairs) to share system status or configurations (e.g., server port number, foreground app name) between processes, but any app can use Java reflection to call `android.os.SystemProperties.get()` to read any item (most of them can also be written).

Table 2: Emulator features, security/privacy risks and security implications.

Feature	Count*	Risk	Security Implication
User Input (IME, Macros, Keyboard-map)	6	high	Sensitive information within user's input could be sniffed or spoofed.
Install App	6	high	Malicious 3rd-party apps from APKs could be installed on the emulator.
Tab Manage	4	high	Misleading tab information facilitates app spoofing/phishing.
ADB	6	high	ADB shell can run privileged commands.
Screenshot	6	high	Private information on the screen can be captured without users' consent.
Shared Folder	6	high	Some files shared between host and guest can be security-sensitive.
Set Location	5	low	App uses fake GPS location.
Shake	6	low	App senses false phone shaking events.
Back/Home Button	5	low	Navigation without users' consent.
Rotate Screen	6	low	Adversely affects user experience.
Clear Recent Apps	2	low	Adversely affects user experience.
Reboot	2	low	Interrupts user's operation.
Volume Control	5	low	Adversely affects user experience.

*: Count of the emulators (within the 6 studied ones) having the feature.

4 FLAWS AND ATTACKS

In this section, we analyze the concrete vulnerabilities associated with the emulator functionalities and demonstrate practical attacks against them.

Exploited Functionalities. We first investigate the website descriptions of the six studied emulators (mentioned in Table 1) and enumerate *WinShell* UI elements to discover their provided features, which are listed in Table 2. To notice is that we skipped the features that are irrelevant to Android, such as full-screen mode and cursor locking. Among all the features, we identified six of them as security-critical which have severe security implications if being exploited. The others are marked as low-risk, because exploitation on them has minor impacts on users.

Our Approach of Vulnerability Discovery. We thoroughly studied the high-risk features, identified vulnerabilities and constructed effective attacks against them, as described from Sections 4.1 to 4.4. We discuss "Shared Folder" together with "Install APK" in Section 4.2. The remaining high-risk feature "Screenshot" and the low-risk features are briefly discussed in Section 4.5.

Below we summarize the general steps we have taken in analyzing each emulator and feature. These steps help us locate the components (e.g., the *HostService* daemons and their binary/APK files) that are responsible for the vulnerabilities and further verify them. Section 6.3 provides more details about the vulnerability discovery process.

(1) We compared the customized firmware in the emulator to the AOSP firmware and extracted the customized programs, Android packages, and particular device nodes.
(2) By inspecting VirtualBox standard configuration files and logs on the host, we learned all the forwarded ports (from Android to Windows) and added virtual devices, and then validated them on the emulator.
(3) We associated the channels (TCP/UDP and virtual devices) with the emulator components through dynamically monitoring the network and resource usage (e.g., through "netstat", "lsof", "dumpsys" commands on Android).
(4) We triggered each feature and captured all generated loopback traffic (using Wireshark [71] on Windows and tcpdump [65] on Android).
(5) We examined whether they are vulnerable and constructed exploitation through either dynamically replaying/injecting the testing requests or statically analyzing their programs.

Demo. We posted our attack demos online at https://sites.google.com/view/emulatorsec.

4.1 User Input

Users can use a keyboard and mouse on emulators to conduct the same operations as touching the screen of a smartphone. The mouse click will be sent to Android to generate a raw touch event through the VirtualBox standard "hardware" or customized channels, which could also deliver the raw keyboard events. However, special treatment is needed when users want these raw events to be processed by Input Method (IME) to generate words and texts. The typical Android IME may require multiple times of touches to complete a word, which is convenient on mobile phones, but repeatedly clicking letters can be annoying on PC. Thus, to provide a seamless experience, emulators allow users to keep using Windows IME when typing, and transfer the text into Android directly once completing the current typing cycle. Since this delivery of texts is not natively supported by VirtualBox, emulators usually introduce their customized channels.

Input Processing on Emulator. Figure 3 shows the typical input processing flow on the emulator, which is built upon the Android standard input subsystem [9]. The inputs on Windows fall into two categories: raw input (usually as events) and Windows IME input (as text). Generally, we found three ways for an emulator to send inputs to Android:

A. The raw input is transported to *HostService* through their customized channels like TCP/UDP or customized virtual device, which is shown as channel A in the figure. Then, *HostService* daemon utilizes Linux *uinput* [21] mechanism to create a local input device, which will then be captured by the standard Android input system and finally dispatched to the target app.

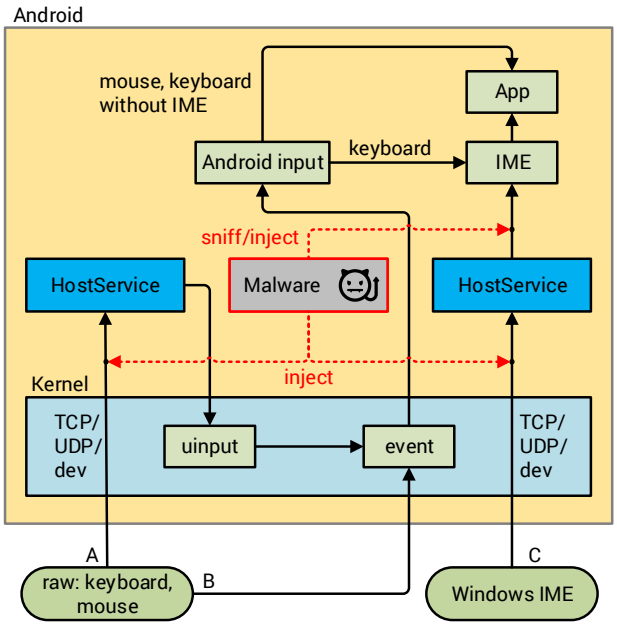

Figure 3: Emulator Input Subsystem and Attacks.

B. A virtual device relays raw input from host to Android kernel driver, shown as channel B in the figure. The kernel receives the input as if it were from standard hardware, and the inputs will go through the standard Android input system. VirtualBox's built-in keyboard and mouse functions follow this link.
C. When Windows IME is active, shown as channel C in the figure, the completed text will be sent to *HostService* through a customized channel, and *HostService* will forward it to the Android IME, which can submit text to the focused input area. The IME is specially designed by the vendors to receive texts from *HostService* instead of interacting with the user through the soft keyboard.

User's input is considered sensitive, and injecting input events is protected by the `INJECT_EVENT` permission. Also, unprivileged apps cannot sniff user's input in other apps. There have been some attacks on Android input subsystem, which are based on accessibility service and UI overlays [40], inferred from sensors [34], or through third-party keyboard apps [37]. Different from previous works, the vulnerabilities we found are introduced by the architecture of emulators. Specifically, we found the path A and path C both involve emulator customized communications, which are exposed to attackers, resulting in input spoofing or sniffing.

Attack #1: Input Spoofing (violates SG#1, 3). In this attack, the malicious app tries to inject spoofing IME data or raw input data to Android. For instance, in NoxPlayer, both types of input are transmitted through UDP to fixed ports, where the touch event contains a series of packets: `MULTI:1:0:<xCoord>:<yCoord>`, `MULTI:0:6`, `MULTI:0:6`, `MULTI:0:1`, and `MSBRL:0:0`, with each ending with a line feed, and the text input uses a packet `{c:2,t:"<text>"}`. Knowing the format, we can inject arbitrary touch/key event or

IME text into Android. Note that all the above events are global, which means our app in the background can perform basically any UI operation with this attack. For example, the attacker could arbitrarily modify system critical settings, grant any dangerous permission, or open the email app, draft a fake email and send it out.

Attack #2: Keylogger via Input Sniffing or MITM (violates SG#1). Here we give two examples about sniffing on users' input, one by passive sniffing and another by active man-in-the-middle (MITM). In LDPlayer, *HostService* relays completed text from host to Android IME using Android `Broadcast`, with a `protected-broadcast` tag. However, this tag can only prevent unprivileged apps from sending the broadcast packets but still allows receiving. Thus, the attacker can register a `BroadcastReceiver` with the intent filter specifying action `android.intent.action.EMU_IME_ACTION` to receive the broadcast intent, and simply learn the input text through `intent.getStringExtra("text")`.

In another case, *HostService* of BlueStacks reads IME text from a virtual device and forwards it to Android IME through a local TCP connection. We observed the port number of IME (TCP server) is not fixed, and BlueStacks stores the dynamic port number in customized `SystemProperty` with the key `bst.config.imelistenerport`, so that the sender can find the server. However, this `SystemProperty` turns out to be both readable and writable by any app. The attacker could launch a MITM attack by modifying this port number to its own TCP server port, and then relaying the packets to the real server. This flaw gives the attacker sniffing and spoofing capability at the same time, and hide from users.

Discussion. Here we reason about the design rationale behind the customized channels. Path C is designed to enable users to use Windows IME instead of Android IME. For the path A, we found it can facilitate their game assistant features, such as keyboard mapping (converting certain key events to touch events) and macro recording (pre-recording a series of input events which can be replayed). In this case, it is more convenient to use customized channel and data formats (e.g., a recorded input sequence) to avoid low-level data encoding of standard inputs.

4.2 App Installation

Emulator users can use the 1st-party or 3rd-party app stores such as Google Play Store on Android to install apps. Besides, because users may have downloaded some APK files on their host OS, emulators also provide a *"side-load"* installation feature, enabling users to install APK files stored on the host file system onto Android conveniently. This side-load installation inevitably introduces a host-guest communication channel, which may expose an exploitable interface to attackers.

Package Installation Flaws. In Android, an unprivileged app needs to explicitly ask for the user's consent through the system dialog when it wants to install another app. In contrast, silent app installation, i.e., installing without alarming users, is considered highly-sensitive and protected by the `INSTALL_PACKAGES` permission, which is only granted to system-level apps (e.g., Package Installer, official app store). Previous research has discovered vulnerabilities in the installation process of some app stores that enable attackers to hijack the process and silently install other

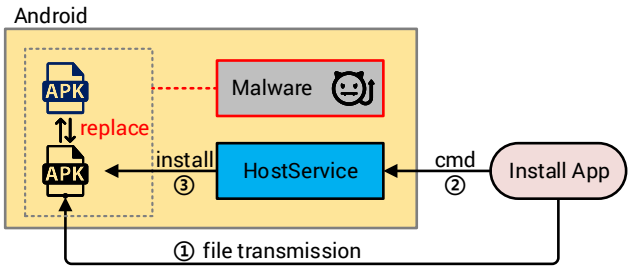

Figure 4: App Installation Flow and Hijacking Attack.

unwanted apps [47]. On emulators, we found similar attacks can be applied to the side-load installation process, which has not been reported before.

As shown in Figure 4, the side-load installation procedure includes three steps. First, the user clicks the "install" button of the emulator and selects the target APK, which could have been stored in any host folder. Then, the APK file is sent from the host to a certain location in Android. This step is commonly done through VirtualBox shared folder service, which allows Android to mount folders from the host OS into its file system. The emulator could either statically configure the folders to be shared, so that they will be mounted upon booting, or dynamically add them while Android is running. Next, the host will issue the installation command, usually specifying the path of the target APK file. Finally, the *HostService*, which is privileged and has the `INSTALL_PACKAGES` permission, performs the app installation on Android.

One possible way to attack this procedure is to inject a spoofing installation command to *HostService* specifying the path of the attacker's malicious APK file. This works if the command channel is unprotected. Besides, we found the emulators are vulnerable to a more general type of hijacking attack. Specifically, we found that most emulators expose the shared folders to all apps with the `READ_EXTERNAL_STORAGE` permission, which is a commonly used permission, and even to any app (e.g., the file access mode is 777). Although emulators may quickly delete the APK file after installation, it does create a side channel for us to track installation requests by monitoring the existence of APK files. More than that, we could replace the target APK file with ours, expecting that Android will wrongfully install our app instead of the target.

Attack #3: Installation Hijacking (violates SG#1, 2). We try to launch a "Man-in-the-Disk" attack to hijack the app installation. First, we assume the attack app has silently downloaded the malicious APK file from the attacker's server to the emulator's storage. To notice, the APK downloading does not trigger any installation action. Then, the attacker exploits the aforementioned flaw to install the malicious APK in place of the original one. This involves a *race condition* between our process and the installer process. We need to find the right timing so that we will replace the file just after it appears in Android, and before the start of installation.

The basic procedure for this attack is shown in Algorithm 1. We monitor the folder for any new APK files that appears, and then copy (or link) our prepared trojan APK to replace the target

Algorithm 1: Installation Hijacking Attack

Input: *apkDir*: the directory of APKs to be installed, usually shared folder
Input: *attackerAPK*: the trojan APK file prepared by the attacker

```
runFlag ← true;
while runFlag do
    files ← apkDir.listFiles;
    for file ∈ files do
        if file.name.endsWith(".apk") then
            copyFile(attackerAPK, file);
            runFlag ← false;
            break;
        end
    end
end
```

file. For example, we can let the check run at a high frequency so that we have a decent chance to succeed in the race (i.e., our app gets installed). We can further extend the attack by checking the content of the APK and replacing it with the corresponding phishing app, so that we can launch phishing attacks to steal user account credentials.

Discussion. In the installation process, the emulators utilize VirtualBox shared folder to transfer the target APK file. Originally, this shared folder is widely used in emulators to share some user files between the host and guest. For example, users may push some pictures or audios into the folder so that apps could access them within Android. Therefore, the shared folder conceptually belongs to the "external storage" on Android, which is initially designed for storing public and less sensitive files. However, it is inconsistent with the requirement of APK installation, and opens the door for our attack.

Another interesting finding we observed in MuMu is that the whole parent folder of the target APK on the host will be shared into Android, with an access mode of "777". Consequently, a malicious app can even gain access to other files/folders within the same directory of the target APK on the host, which poses a threat to the host OS as well.

4.3 Tab Management

The emulators also brought new modes of interaction with apps to adapt to the desktop environment. We found that many emulators (4 out of 6) have introduced a tab-bar above the Android window, similar to the browser tabs, as shown in Figure 2. This feature informs users about the running apps and enables users to switch between or close apps easily. Naturally, users will rely on the information provided by tabs to identify apps and their states. For example, if the active tab shows a label "Google Play Store" and the Play Store's icon, users will consider the current foreground app as Play Store, as long as the app's UI looks like the expected one.

Since the tabs are implemented on *WinShell* process, the emulator relies on the host-guest communication channel to properly synchronize the states of apps and tabs. As shown in Figure 5,

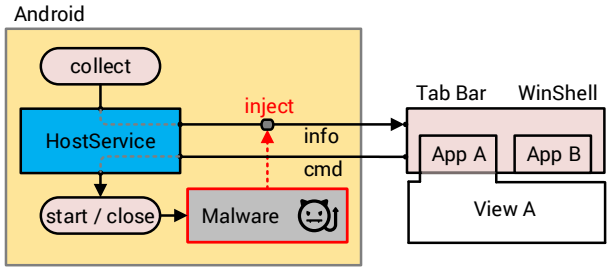

Figure 5: Tab Session Management and Hijacking Attack

the user's interaction with the tabs will generate a corresponding command, e.g., moving an app to the foreground or closing an app, which will be transmitted to Android and executed by a process (*HostService*) with high privilege. On the other hand, when an app is started, moved to the foreground, or closed in Android, *HostService* process will collect the app state information and notify *WinShell*, so that tabs can be updated accordingly.

However, without proper protection on the communication channel, a malicious app can send spoofing messages in either direction, and break the consistency between tabs and app states. Here we present the Tab Session Hijacking attack, in which the user will be deceived to recognize our bogus activity as a normal activity from a benign app (called "target app"). As a result, the user may perform sensitive actions on the malicious app, such as entering the login credentials.

Attack #4: Tab Session Hijacking (violates SG#3). We can hijack the tab session by spoofing the status (active/inactive) of tabs. Suppose our app is running in the background, and we can launch the attack as follows:

(1) *Monitor app states.* Our app keeps inquiring the current foreground app/activity through some channels exposed by the emulator. For example, MuMu's *HostService* provides such information through HTTP, so any app can send request `GET localhost:6667/v1/apps?running_apps` and the response contains the foreground app's package name. Such information is considered highly-sensitive in modern Android system and should never be leaked to 3rd-party apps (see discussion below).
(2) *Launch our bogus activity.* Once the target app is in the foreground, our app immediately starts the prepared bogus activity, which looks the same as that of the target app. This step will also bring our malicious app to the foreground, so its tab will be shown as active.
(3) *Change active tab.* Our app quickly sends a message to *WinShell*, so that it will set the target app's tab as active. In MuMu, this message is an HTTP request `POST <hostAddr>:22471/player/tab` with the data containing the target app's task ID (can be inferred), name, and so on. Note that this message will only change tabs but not impact the actual app states, so the user still sees our bogus activity (as the previous step makes it in the foreground), but with the target app's tab as active. Since the steps could be executed with negligible delays, the user can hardly notice the change.

The above attack is not the only way to hijack tab sessions. For example, in some emulators, we can even manipulate tabs with finer granularity, like specifying arbitrary icons, labels, and package names. As a result, we can create a tab that shows a legitimate app's icon and label name, but binds to a malicious app, i.e., clicking the tab will start the malicious app. With a phishing UI mimicking the real one, the user will not be able to tell she is actually interacting with a malicious app.

Discussion. Along with the tab hijacking issue, we found it is common that emulators disclose app states information. In addition to the aforementioned MuMu, BlueStacks also leak app states by writing the current foreground app information to `SystemProperty` which is publicly readable. LDPlayer will send an unprotected Broadcast `android.intent.action.TOP_ACTIVITY_CHANGED` containing the foreground app's package name whenever the top activity changes, which any app can learn using a `BroadcastReceiver`. However, in AOSP, this app state information is treated as sensitive and protected by a signature-level permission `REAL_GET_TASKS`. Revealing this information will not only invade user's privacy, but also serve as the basis of various phishing attacks [35, 38, 67]. Recently, Possemato et al. even developed an automated system to detect such vulnerabilities in Android [55], resulting in 6 CVEs. Our result shows similar problems exist in the emulators.

4.4 Shell and ADB

Overly-Opened ADB. Android Debug Bridge (ADB) [7] is an official tool to communicate with Android devices over USB or network, and facilitate various actions like installing or debugging apps, transferring files, and accessing the Linux shell on Android. Once started, ADB starts daemon `adbd` on the Android phone which acts as the server, and a client can connect to it via TCP. With its powerful capabilities, ADB can also open the door for adversaries to circumvent many of the security checks and measures of Android, which has been analyzed by previous research [43].

On Android phones, ADB is designed for developers, so a series of protection and warnings are designed to prevent normal end-users from opening it. Specifically, the user must explicitly enable developer mode and allow debugging on their devices before using ADB functions, as shown in Figure 6. Also, the connection intention must be confirmed by users. Since the emulators we study are designed mainly for end-users, ADB should also be protected just like on a real phone. Here we derived the following security policies from AOSP that emulators should conform to:

- User should be able to disable/enable the ADB in emulator settings (i.e., the global settings menu on host OS) or in Android settings. It should be disabled by default.
- When a new ADB client is connected, a dialog should be prompted for user consent.
- The ADB shell's privilege should be limited (i.e., uid is `shell`[3] rather than `system` or `root`).

We then examined how ADB is configured by emulators (shown in Table 3). We found all emulators under our study come with ADB functionalities, but surprisingly none of them adheres to the above policies. They are all overly-opened, and for 4 of them, *ADB cannot be turned off*, and is always connectable by any client without notification. In this case, a malicious app can connect to the server `adbd` via local TCP connection and escalate its privilege.

[3]Official Android assigns uid `shell` to adbd. This uid has some privilege but is still restricted from critical system resources.

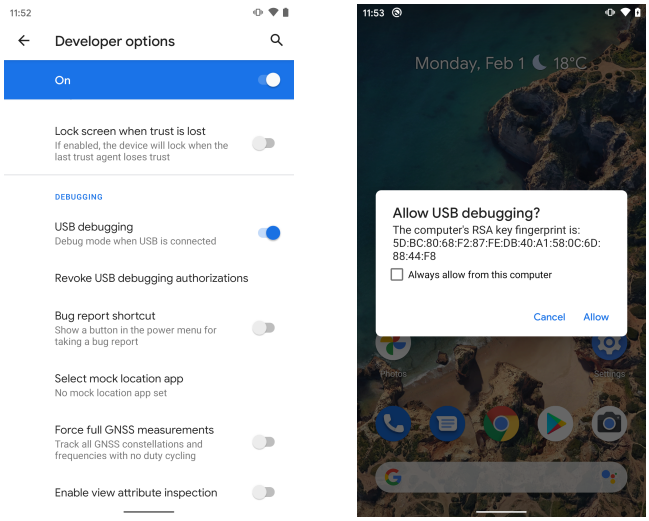

Figure 6: Enable and confirm ADB connection.

Table 3: ADB features on emulators.

Emulator	User Switch	User Confirm	Privilege
BlueStacks	Yes*‡	No	shell
LDPlayer	Yes*	No	root
MEmu	No	No	root
MuMu	No	No	root
NoxPlayer	No	No	root
GameLoop	No	No	root

*: Disabled by default.
‡: Warn users about security risks when being turned on.

Exposed Shell. Besides ADB, we found some emulators also exposed other privileged shells to the attacker. As mentioned in Section 2.2, there is usually a *HostService* daemon that receives messages from the host and performs corresponding actions. In most cases, the vendors designed their own simple protocol to indicate the command actions. But for some emulators, their *HostService* also accepts inputs as an entire shell command, which runs in a Linux shell with even the `root` privilege. For example, on MuMu one app can send an HTTP request `POST /tools/cmd` to 6667 port with the body `{"action": "run","params": {"command": "<command>"}}`, and from our reverse-engineering, we found this command is executed by a root shell without any check.

One reason for vendors to introduce this shell is probably for flexible updates. With the hard-coded command formats, the vendors need to update both Windows and Android programs in order to adjust/add features. But with this shell, this can be done by a simple hot patch on the host. The shell is designed to run as root so that all future features will be possible. This clearly violates

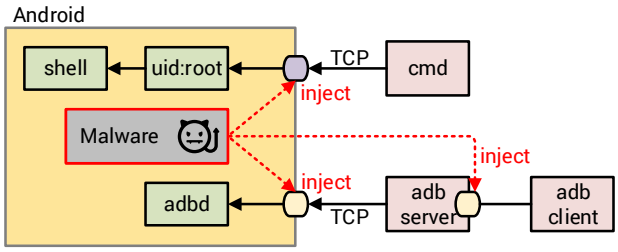

Figure 7: Emulator Shell & ADB Abusing Attack.

the "least privilege" principle, and it exposes all powerful shell functionalities to malicious third-party apps.

Attack #5: God-Mode App (violates SG#1, 2, 3). By leveraging the exposed shell or ADB service (as shown in Figure 7), a malicious app could gain `root` privilege on Android. It extends the scope of previous attacks which was originally impossible in some emulators. For example, with system-level privilege, the attacker can directly inject or sniff any input event into Android. Sniffing any network traffic becomes possible (e.g., by using tcpdump [65] tool) as well. The attacker can also remount read-only system partition into writable one and modify critical system components. Or she can also plant a permanent virus or backdoor to monitor and collect privacy.

Discussion. The surprising openness of ADB made us wonder about the design rationale. We found that although these emulators try to attract end-users by providing an easy-to-use experience, they are also trying to impress some "advanced" users and even some developers by showing their highly-configurable features and interfaces. These users may want to manage several different game accounts at the same time, perform automated tasks, and perform a bit of "hacking" to explore the system. This can be seen from the multi-instance feature (running multiple Android systems and perform batch operations) provided by these emulators, and the ADB tutorials on their official websites (e.g., MuMu's ADB guide [3]). However, the problem is that they mixed the two types of users, and sacrificed end-users' security to exchange for the convenience of advanced users.

While exploiting the ADB, we also tackled some obstacles by utilizing the ports on the host OS. We found some emulators like MEmu will start their own ADB server (not `adbd`, but a host-side process used to manage communication between ADB client and the `adbd` daemon[4]) and maintain the connection with `adbd`, which might block new connection requests. To tackle the issue, the malicious app could initiate a connection to the ADB server (usually on port 5037) remotely instead (shown as the middle red line in Figure 7). The ADB server allows multiple client connections and further forwards ADB data to the `adbd` daemon on Android.

4.5 Other Functionalities

Screenshot. The screenshot could contain sensitive information for the current app (e.g., user profile, chatting messages). Therefore, capturing global screenshots from 3rd-party apps is considered dangerous and will be protected. However, with exposing the features to the host, the emulators might leak the screen content to adversaries as well. We evaluated all six emulators and found three of them suffering from the issues, as shown in the "Screenshot" column of Table 4. Besides, MuMu and Gameloop implement this feature purely on Windows side, which do not introduce host-guest communication (but the attack still works on MuMu through the exposed shell as mentioned in Section 4.4). Note that the emulators usually store the screenshot in the public folder so that attackers can retrieve the images easily after injecting commands.

Low-risk Features. There are also other customized features that involve the host-guest communication channels and might suffer from similar design flaws. These functionalities may not be as security-critical as the above ones, but once undermined, they can still degrade user experience or assist other powerful attacks. For instance, the emulators usually display "back", "home", "recent" navigation buttons or "reboot" power button on the sidebar, which will send the corresponding command to Android when being clicked. We can inject these commands to emulators, and this function alone can achieve a DoS attack. In another example, the "shake" button on the emulator sidebar will make the virtual accelerometer in Android produce a series of data simulating a shake motion. We found some emulators use an unprotected channel so that we can also inject motion events to Android. The sidebar allows users to set the system's location anywhere, and the instruction is also sent through the channel. We can spoof the command to change the system location on some emulators.

5 MEASUREMENTS

In this section, we summarize our findings across emulators under each type of attack described in Section 4. The overall result is shown in Table 4. We can see that the emulators show diversity in their service implementations. Accordingly, our attacks are implemented in different forms. Such diversity makes a unified mitigation strategy challenging.

Attack on User Input. We found almost *all* emulators can be exploited by a malicious app. For four out of six emulators, their open channels enable attackers to inject arbitrary characters/words to Android. Among them, NoxPlayer also can be injected with touch events, so we can perform arbitrary operations on behalf of the user, such as modifying app/system settings and performing some in-app purchases. For two emulators, we can sniff or intercept user's input, meaning that we can steal user private information, like account credentials. In particular, BlueStacks is vulnerable to a complete MITM attack because we can inject inputs to its IME server and manipulate the server port information (stored in `SystemProperty`) to intercept the input messages (let them flow to our malicious server) in the meantime.

Attack on App Installation. We found three emulators' installation commands are transferred through unprotected HTTP/TCP channels, indicating that attackers can install arbitrary APKs bypassing the `INSTALL_PACKAGES` permission. Besides, we found that uninstalling apps is also possible in these emulators. Consequently, we can launch a phishing attack by uninstalling the target app and installing our bogus one. Furthermore, five emulators expose their ready-to-install APK files and directories to 3rd-party apps. And we

[4]It is the official Android communication model [7].

Table 4: Implementations of main functionalities by the emulators and corresponding attacks. "Host-guest", "Local", "Install cmd" and "Top app info" columns indicate what channels are used in the corresponding implementation. "Attack" columns show what attacks (or means of attack) are enabled.

Emulator	Raw input event		IME input		Attack on Input	Screenshot	
	Host-guest	Local	Host-guest	Local		Host-guest	Attack
BlueStacks	Dev	N/A	Dev	TCP†‡	IME input MITM	HTTP†	by HTTP inject
LDPlayer	Dev	N/A	Dev‡	Broadcast*	sniff/intercept IME input	Dev	-
MEmu	Dev	uinput	Dev	TCP†	inject IME input	Dev	-
MuMu	TCP	uinput	TCP†	N/A	inject IME input	-	by shell
NoxPlayer	UDP†	uinput	TCP/ UDP†	N/A	inject raw input, IME input	TCP†	by TCP inject
GameLoop	Dev	uinput	Dev	N/A	-	-	-

Emulator	App installation			Tab management			ADB & shell
	Install cmd	File/folder access	Attack	Top app info	Guest-host cmd	Attack	
BlueStacks	HTTP†	no access	uninstall + install	`SysProp`*	HTTP†	hijack tab session	ADB**
LDPlayer	Dev‡	rw/ro	race condition	Broadcast*	Dev†	hijack tab session	ADB**
MEmu	Dev	rw/rw	race condition	-	Dev	-	ADB
MuMu	HTTP†	rw/rw	uninstall + install	HTTP†	HTTP†	hijack tab session	ADB, shell
NoxPlayer	TCP†	rw/rw	uninstall + install	N/A	N/A	N/A	ADB, shell
GameLoop	Dev	rw/rw	race condition	N/A	N/A	N/A	ADB

Flaw types: * = can sniff, †= can inject (spoof), ‡= can intercept
**: with user switch

have successfully launched a more general installation hijacking on three emulators by race condition attack, such that installation functions of all studied emulators are vulnerable.

Attack on Tab Management. Except for the two emulators which do not have tabs on Windows, we successfully implemented the attack in Section 4.3 on three out of four emulators. They disclosed the information of the foreground app in different ways, such as sending Android Broadcasts, writing it in `SystemProperty`, or putting it in the response of an HTTP request. Such information tells the right timing for our attack. We could spoof the tab status through the guest-to-host channel, leading to the inconsistency between the tab and the corresponding app. For LDPlayer, we can also spoof tabs to have the appearance (label name and icon) of other apps but bind to our app, resulting in more flexible attacks.

Attack through ADB and Privileged Shell. In addition to opened ADB (in Table 3), MuMu and NoxPlayer both expose root shells and directly execute commands from the host messages, which makes them extremely vulnerable to adversaries. Through abusing ADB or exposed shell, all previous attacks could be enabled.

Special Cases. Although most emulators can be attacked by simple steps such as sniffing, spoofing, reading/writing, some emulators have made attempts to protect their communication links. We noticed that BlueStacks actually added a 128-bit token verification to the HTTP server, such that only requests with the correct token will be processed. However, the token is not only a fixed value (will only change after re-installation of BlueStacks), but also stored in public `SystemProperty`. Therefore, in our attacks towards BlueStacks, we will retrieve the token and append to each HTTP request header. We noticed that the token is originally used to prevent fake commands from the host side [2], but it becomes invalid in our attack model.

6 DISCUSSION

6.1 Responsible Disclosure

We have reported our findings to corresponding emulator vendors. Among them, MuMu has confirmed our reported vulnerabilities and patches will be released in the future. We also got positive responses from BlueStacks and LDPlayer. They will improve and update their products in the following versions. For the rest, we are actively communicating with them to discuss and propose mitigation solutions.

6.2 Lessons and Mitigation

Lessons Learned. The root cause of many vulnerabilities presented in this work is that emulator vendors failed to protect the customized communication channels and privileged services they introduced to Android. Previous research has demonstrated that Android customization can introduce many security problems (see Section 7). We believe the emulator developers should take extra care to understand the Android security model and ensure the newly introduced functionalities comply.

Meanwhile, the vendors need to understand their target users and apply the least-privilege principle. Some privileged functions

such as ADB target advanced users (e.g., run shell script for batch operations) and should not be opened by default.

Mitigation Suggestions. Based on the existing emulator architecture and considering the necessity of customized communication channels, here we give some mitigation suggestions. TCP/UDP channels (HTTP is also over TCP) do not provide authentication and encryption in the transport layer by default. Therefore, additional application-layer protection should be deployed. For example, SSL/TLS [1] is an immediate standard solution that is widely used to enhance communication security. In addition to this, the data encryption can be based on standard symmetric-key algorithms like AES. The implementations of authentication are diverse, and pre-shared secret (e.g., token or key) can be a lightweight approach. The secret can be written into guest's (Android) firmware before initiating the guest when launching the emulator for the first time.

Also, virtual Linux devices should be protected by Linux user-based access model or fine-grained access control mechanisms on the driver. The Android IPC using Intent (Broadcast is also an Intent) should specify specific permissions, as suggested by Chin et al. [36]. Alternatively, an additional system service could be introduced, and it acts as an intermediary between apps and daemons to prevent their direct communication [64]. The mutual connection requests must be validated by this system service. Beyond that, the app installation process should be handled with great care. We extend the suggestions given by Lee et al. [47] and recommend the vendors to store APK in private locations [48] and verify the hash values of the target APK file before installation (comparing with the hash value of the user-selected file on the host).

With the above measures taken, a malicious app can still escalate its privilege using ADB. We suggest the vendors conform to the ADB security policies mentioned in Section 4.4 to minimize the chances of attackers. Furthermore, previous work [43] has also suggested ADB defenses such as automatically disabling it when idling, restricting ADB functionalities, and adding access checking on the ADB server, which can also be applied to emulators. From another aspect, extending the current ADB protocol is also a choice, such as introducing the HMAC mechanism [52].

6.3 Limitations and Future Work

Extension of Threat Model. Our attacks are initiated from an Android app within the emulator. This adversary model could be extended to initiate the attacks from the host side. Because some of our attacks are achieved by spoofing HTTP/TCP requests to particular open ports on Android, which are also mapped to host loopback network, host-side attackers could attack the same set of ports with spoofing. One consequence is abusing the vulnerable ADB service to monitor user activity, steal private data, or install malicious Android app on the emulator, as described in Section 4.4.

Here we describe two approaches under this host-side setting. 1) As described in [2], the attacker might trick the user to visit a malicious webpage (e.g., `http://evil.com/ipc/command`), and change the website IP to `127.0.0.1` to launch a DNS rebinding attack. When the emulator listens to localhost, the attacker's command could be executed. 2) The attacker can compromise a Windows application and use it to relay commands to the emulator. However, this approach might be less favorable, as the attacker can directly break the integrity of the emulator (e.g., by accessing emulator configure files, the whole Android image file on the host), without the need of carefully crafting the attack packets.

Study Scope. We focused on the Windows versions of the studied emulators because of the popularity of the Windows OS. Whether the macOS versions are also vulnerable could be investigated in the future. Besides, we focus on the security of end-user emulators, while leaving out the developer-oriented emulators like Genymotion [18] and "emulators" (i.e., Android image) requiring manual installation on a VM like Bliss OS [14]. Extending the study to them would require efforts in reverse-engineering their host-guest communication interfaces or pre-installed apps, which are non-trivial. We leave the investigation as a future task.

Vulnerability Discovery. In this study, most vulnerabilities are discovered with manual reverse-engineering and verification. We learned that different emulators use similar types of communication channels and can be summarized by an abstract model, but their architectures and implementation details are varied. For example, the *HostService* daemon may either be implemented as a native app (written by C/C++) or an Android app (written by Java), and the same functionalities can be implemented in totally different channels and protocols for different emulators. Therefore, it is challenging to automate the analysis procedure. However, we believe the discovered issues are relevant to other emulators not studied in this paper, as long as they involve customized host-guest communication. In the following, we discuss possible full or semi-automation approaches for assisting vulnerability analysis and identification.

Generally speaking, the emulator runs in the controlled environment within a host, where we can mimic and automate user operations (like recognizing UI components, clicking sidebar buttons through available desktop automation tools [13, 25]), intercept network traffic, read/analyze log files, etc. In addition, we have a privileged Android app within the emulator for inspecting the Android system (e.g., using "dumpsys" tools [4] or through API hooking) and loading dynamic tests. Below we describe how different types of actions can be simulated to trigger abnormal emulator behaviors.

- *User Input.* We could first send the normal user input (either raw or IME input) to the emulator. Meanwhile, all network traffic associated with the emulator and Android Intent events will be collected. The content of user input and the network traffic will be matched to pair the ones that are causally dependent. Previous works [83, 84] have similar steps to identify vulnerabilities in online services, which we could adapt to our analysis. Next, we try to *replay* the network request or Android Intent from an attack app with only normal permissions within the emulator. If the same content is shown in the Android system (e.g., on the foreground input box), we consider the emulator is potentially vulnerable.
- *App Installation.* We observed that when installing an app from the host, the emulator usually utilizes the shared folder (between Android and host OS) to transmit the target APK file. We could extract the folder path from standard configuration or log files of the VM/emulator. Then we will analyze the access permission settings of the shared folder, its sub-folders, and the target APK file. If the access permissions are too loose to allow replacing, overwriting,

or soft-linking on the target APK, we will simulate the installation of a legitimate APK and launch the race condition attack in the meantime. The installed package list will be monitored to see if the installation procedure is hijacked.

• *Tab Management.* Similar to the above discussion on User Input, we capture the network traffic relevant to tab generation and tab status update. Next, we could open a random app and mutate the tab-related request fields (e.g., package name, icon, status of activeness). Then we send the mutated requests from the attack app and check if the tab content/status (by some Windows UI tools on the host) and the actual activity shown in the foreground (by dumping the top activity information on Android) are inconsistent.

• *ADB.* Since the default ADB port is 5555 on Android, we could use standard ADB protocol to probe the port (if the ADB port is not constant, we can probe all open ports). We then verify whether it satisfies the security policies in Section 4.4. It could be done by monitoring and inspecting the Android pop-up dialog (a standard Android `View`), and examining the `uid/gid` of the ADB shell.

Other Issues about Emulators. This paper analyzed the unique host-guest communication mechanism of emulators. We acknowledge that not all security issues are covered: e.g., the ones about the standard components within Android OS and emulator's own cloud services like its membership service and app recommendation service. Besides, we assume emulators are benign, but this assumption may not hold. For example, the emulator may stealthily track users, like what happens on the TV streaming devices [49].

7 RELATED WORK

In this section, we review the related work, including Android customization, virtualization security, and emulator detection.

Android Customization. The stock firmware of Android phones is often customized to deliver vendor-specific features. However, such customization may introduce new vulnerabilities. For example, Zhou et al. [82] discovered several flaws related to driver customization, allowing an unauthorized app to take pictures and record the user keystrokes on the touchscreen. Aafer et al. [26] found the hanging attribute references flaw – when an attribute is used on a device but the party defining it has been removed. Tian et al. [66] focused on the security of AT (ATtention) commands, and new AT commands that are constantly added into the stock firmware. Aafer et al. [27] proposed DroidDiff to detect security configuration changes introduced by the customization. Possemato et al. [54] performed a longitudinal study on Android OEM customization, focusing on SELinux configurations, system binaries hardening, init scripts, and the Android Linux kernel.

In a related direction, some works studied the security of pre-installed apps introduced by vendors. Wu et al. [73] studied the permission over-privilege and privacy leakage of pre-installed apps. Zheng et al. [81] designed DroidRay to detect the pre-installed malware in firmware images. More recently, Elsabagh et al. [39] studied the privilege-escalation vulnerabilities, and Gamba et al. [41] explored the ecosystem of pre-installed apps.

Virtualization Security. Virtualization is the foundation of modern computing technologies, and their security issues have been well studied on platforms like Windows, Linux, and cloud. The major threat is the VM escape attack, which lets the attacker interact with the host machine or other VMs outside of the hosting sandbox [42]. Ristenpart et al. [57] discovered VM reset vulnerabilities that affect the random number generators. On the cloud environment, Rocha et al. [58] showed that a malicious insider can steal confidential data of another cloud user, like passwords, crypto keys. Other security risks, like VM rollback [75], VM relocation [63], row hammer [77], multiple kinds of covert channel [74, 76, 78] and side-channel [56, 79, 80] were also studied. Our work looks into the security of Android emulator on a PC, which has not been systematically studied before.

Emulator Detection. To detect Android malware, one common approach is to run apps in a controlled emulator and expose the malicious behaviors. To evade such defense, malware can detect the presence of the emulator and hide its malicious behaviors. Vidas et al. [68] presented a number of ways to detect Android emulators, including the differences in behavior, performance, hardware, and software components. A similar work was conducted by Petsas et al. [53], which found trivial techniques were enough to evade some dynamic analysis. Jing et al. [44] presented Morpheus, a system that automatically generates emulator detection heuristics. It can retrieve observable artifacts from both emulators and real devices. Sahin et al. [60] uncovered the detection methods based on discrepancies in instruction-level behavior between software-based emulators and real ARM CPUs. Though also targeting emulator security, this direction is orthogonal to our work.

8 CONCLUSION

Android emulator enables people to run mobile apps on the desktop environment, thus providing users with a better experience under some scenarios. Millions of users, especially gamers, choose to use emulators, and attackers are also seeing the emulator as a lucrative target. In this paper, we uncovered the unique architecture of Android emulators that incorporates communication channels between host OS and Android OS. These channels are used for supporting various emulator functionalities (e.g., keyboard mapping). We found that these services are usually not adequately protected in Android, so attackers could easily exploit them to escalate their privileges within the emulator. To demonstrate the security consequences of such flaws, we further came up with five attacks targeting four main functionalities of Android emulators. Through these attacks, we could stealthily collect sensitive user logs, steal account credentials and even get root privilege of the system. We measured the flaws on six popular emulators and found the issues are prevalent. Through our research, we hope to raise the awareness of emulator vendors and users and advance the relevant studies on the security of the emulator ecosystem.

ACKNOWLEDGMENTS

We want to thank our shepherd Güliz Seray Tuncay and all the anonymous reviewers for their valuable comments. This work was supported in part by National Key Research & Development Project of China (Grant No. 2019YFB1804400), and Hong Kong S.A.R. Research Grants Council (RGC) General Research Fund No. 14209720. Wenrui Diao was partially supported by National Natural Science Foundation of China (Grant No. 61902148). Zhou Li was partially supported by gift from Cisco and Microsoft.

REFERENCES

[1] 2018. *The Transport Layer Security (TLS) Protocol Version 1.3.* Retrieved May 7, 2021 from https://tools.ietf.org/html/rfc8446

[2] 2019. *BlueStacks Flaw Lets Attackers Remotely Control Android Emulator.* Retrieved August 15, 2021 from https://www.bleepingcomputer.com/news/security/bluestacks-flaw-lets-attackers-remotely-control-android-emulator/

[3] 2019. *DeveloperGuide of MuMu (in Chinese).* Retrieved May 7, 2021 from http://mumu.163.com/help/func/20190129/30131_797867.html

[4] 2020. *dumpsys | Android Developers.* Retrieved August 15, 2021 from https://developer.android.com/studio/command-line/dumpsys

[5] 2020. *How to stream PUBG Mobile on YouTube with laptop.* Retrieved May 7, 2021 from https://www.sportskeeda.com/esports/how-stream-pubg-mobile-youtube-laptop

[6] 2021. *Anbox - Android in a Box.* Retrieved May 7, 2021 from https://anbox.io/

[7] 2021. *Android Debug Bridge (adb) | Android Developers.* Retrieved May 7, 2021 from https://developer.android.com/studio/command-line/adb

[8] 2021. *Android Emulators Wiki.* Retrieved May 7, 2021 from https://emulation.gametechwiki.com/index.php/Android_emulators

[9] 2021. *Android Input.* Retrieved May 7, 2021 from https://source.android.com/devices/input

[10] 2021. *Android Open Source Project.* Retrieved May 7, 2021 from https://source.android.com/

[11] 2021. *Android Studio.* Retrieved May 7, 2021 from https://developer.android.com/studio

[12] 2021. *Application Fundamentals | Android Developers .* Retrieved May 7, 2021 from https://developer.android.com/guide/components/fundamentals

[13] 2021. *AutoHotKey.* Retrieved August 15, 2021 from https://www.autohotkey.com/

[14] 2021. *Bliss OS.* Retrieved May 7, 2021 from https://blissos.org/

[15] 2021. *BlueStacks – Fastest Android Emulator for PC & Mac |100% Safe and FREE.* Retrieved May 7, 2021 from https://www.bluestacks.com/

[16] 2021. *Cooperation with NetEase MuMu (in Chinese).* Retrieved May 7, 2021 from http://mumu.163.com/2016/12/15/25241_661774.html

[17] 2021. *core/java/android/os/SystemProperties.java - platform/frameworks/base - Git at Google.* Retrieved May 7, 2021 from https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/os/SystemProperties.java

[18] 2021. *Genymotion Android Emulator | Cloud-based Android virtual devices | Develop - Automate your tests - Validate with confidence.* Retrieved May 7, 2021 from https://www.genymotion.com/

[19] 2021. *Intent and Intent Filters | Android Developers.* Retrieved May 7, 2021 from https://developer.android.com/guide/components/intents-filters

[20] 2021. *LDPlayer - Fastest Android Emulator for PC, Free Download.* Retrieved May 7, 2021 from https://www.ldplayer.net/

[21] 2021. *Linux Uinput.* Retrieved May 7, 2021 from https://www.kernel.org/doc/html/v4.12/input/uinput.html

[22] 2021. *MEmu - The Best Android Emulator for PC - Free Download.* Retrieved May 7, 2021 from https://www.memuplay.com/

[23] 2021. *Oracle VM VirtualBox.* Retrieved May 7, 2021 from https://www.virtualbox.org/

[24] 2021. *PC Android Emulator for PUBG, CODM - GameLoop.* Retrieved May 7, 2021 from https://www.gameloop.com/en?adtag=default

[25] 2021. *Top Free Automation Tools for Testing Desktop Applications (2021).* Retrieved August 15, 2021 from https://testguild.com/automation-tools-desktop/

[26] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiao-yong Zhou, Wenliang Du, and Michael Grace. 2015. Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS), Denver, CO, USA, October 12-16, 2015.*

[27] Yousra Aafer, Xiao Zhang, and Wenliang Du. 2016. Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis. In *Proceedings of the 25th USENIX Security Symposium (USENIX-Sec), Austin, TX, USA, August 10-12, 2016.*

[28] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick D. McDaniel, and Matthew Smith. 2016. SoK: Lessons Learned from Android Security Research for Appified Software Platforms. In *Proceedings of the 37th IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016.*

[29] Akamai. 2020. *Akamai Report Reveals Broad, Persistent Cyber Attacks Targeting Video Game Players and Companies.* Retrieved May 7, 2021 from https://www.prnewswire.com/news-releases/akamai-report-reveals-broad-persistent-cyber-attacks-targeting-video-game-players-and-companies-301136183.html

[30] Android. 2021. *AccessibilityService | Android Developers.* Retrieved May 7, 2021 from https://developer.android.com/reference/android/accessibilityservice/AccessibilityService

[31] Android. 2021. *Configure on-device developer options.* Retrieved May 7, 2021 from https://developer.android.com/studio/debug/dev-options

[32] Android. 2021. *Permissions on Android | Android Developers.* Retrieved May 7, 2021 from https://developer.android.com/guide/topics/permissions/overview

[33] Android. 2021. *Services overview.* Retrieved May 7, 2021 from https://developer.android.com/guide/components/services

[34] Liang Cai and Hao Chen. 2011. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Security (HotSec), San Francisco, CA, USA, August 9, 2011.*

[35] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*

[36] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David A. Wagner. 2011. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys), Bethesda, MD, USA, June 28 - July 01, 2011.*

[37] Junsung Cho, Geumhwan Cho, and Hyoungshick Kim. 2015. Keyboard or keylogger?: A security analysis of third-party keyboards on Android. In *Proceedings of the 13th Annual Conference on Privacy, Security and Trust (PST), Izmir, Turkey, July 21-23, 2015.*

[38] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. 2016. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016.*

[39] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware. In *Proceedings of the 29th USENIX Security Symposium (USENIX-Sec), August 12-14, 2020.*

[40] Yanick Fratantonio, Chenxiong Qian, Simon P. Chung, and Wenke Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, USA, May 22-26, 2017.*

[41] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. 2020. An Analysis of Pre-installed Android Software. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (Oakland), San Francisco, CA, USA, May 18-21, 2020.*

[42] Bernd Grobauer, Tobias Walloschek, and Elmar Stöcker. 2011. Understanding Cloud Computing Vulnerabilities. *IEEE Security & Privacy Magazine* 9, 2 (2011), 50–57.

[43] Sungjae Hwang, Sungho Lee, Yongdae Kim, and Sukyoung Ryu. 2015. Bittersweet ADB: Attacks and Defenses. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (AsiaCCS), Singapore, April 14-17, 2015.*

[44] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. 2014. Morpheus: Automatically Generating Heuristics to Detect Android Emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC), New Orleans, LA, USA, December 8-12, 2014.*

[45] Swati Khandelwal. 2015. *Minecraft hacked! More than 1800 Minecraft account Credentials Leaked.* Retrieved May 7, 2021 from https://thehackernews.com/2015/01/minecraft-game-hacked.html

[46] Swati Khandelwal. 2019. *Exclusive — Hacker Steals Over 218 Million Zynga 'Words with Friends' Gamers Data.* Retrieved May 7, 2021 from https://thehackernews.com/2019/09/zynga-game-hacking.html

[47] Yeonjoon Lee, Tongxin Li, Nan Zhang, Soteris Demetriou, Mingming Zha, XiaoFeng Wang, Kai Chen, Xiao-yong Zhou, Xinhui Han, and Michael Grace. 2017. Ghost Installer in the Shadow: Security Analysis of App Installation on Android. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Denver, CO, USA, June 26-29, 2017.*

[48] Xiangyu Liu, Zhe Zhou, Wenrui Diao, Zhou Li, and Kehuan Zhang. 2015. An Empirical Study on Android for Saving Non-shared Data on Public Storage. In *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings.*

[49] Hooman Mohajeri Moghaddam, Gunes Acar, Ben Burgess, Arunesh Mathur, Danny Yuxing Huang, Nick Feamster, Edward W. Felten, Prateek Mittal, and Arvind Narayanan. 2019. Watching You Watch: The Tracking Ecosystem of Over-the-Top TV Streaming Devices. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS), London, UK, November 11-15, 2019.*

[50] Shane Monroe. 2017. *Quora: Is it safe and secure to log into an Android Emulator .* Retrieved May 7, 2021 from https://www.quora.com/Is-it-safe-and-secure-to-log-into-an-Android-Emulator-Bluestacks-or-NOX-App-Player-using-my-Google-account-on-my-PC

[51] Nox. 2021. *About Nox.* Retrieved March 2, 2021 from https://www.bignox.com/about

[52] Krzysztof Opasiak and Wojciech Mazurczyk. 2019. (In)Secure Android Debugging: Security analysis and lessons learned. *Computer & Security* 82 (2019), 80–98.

[53] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *Proceedings of the Seventh European Workshop on System Security (EuroSec), Amsterdam, The Netherlands, April 13, 2014.*

[54] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. 2021. Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland), Online, May 23-27, 2021.*

[55] Andrea Possemato, Dario Nisi, and Yanick Fratantonio. 2021. Preventing and Detecting State Inference Attacks on Android. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS), Virtual, 21st - 25th February, 2021.*

[56] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security (CCS), Chicago, Illinois, USA, November 9-13, 2009.*

[57] Thomas Ristenpart and Scott Yilek. 2010. When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 28th February - 3rd March, 2010.*

[58] Francisco Rocha and Miguel Correia. 2011. Lucy in the Sky without Diamonds: Stealing Confidential Data in the Cloud. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), Hong Kong, China, June 27-30, 2011.*

[59] Elena Root and Andrey Polkovnichenko. 2019. *SimBad: A Rogue Adware Campaign On Google Play - Check Point Research.* Retrieved May 7, 2021 from https://research.checkpoint.com/2019/simbad-a-rogue-adware-campaign-on-google-play/

[60] Onur Sahin, Ayse K. Coskun, and Manuel Egele. 2018. Proteus: Detecting Android Emulators from Instruction-Level Profiles. In *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses (RAID), Heraklion, Crete, Greece, September 10-12, 2018.*

[61] Ignacio Sanmillan. 2021. *Operation NightScout: Supply-chain attack targets online gaming in Asia | WeLiveSecurity.* Retrieved May 7, 2021 from https://www.welivesecurity.com/2021/02/01/operation-nightscout-supply-chain-attack-online-gaming-asia/

[62] Help Net Security. 2020. *What's trending on the underground market?* Retrieved May 7, 2021 from https://www.helpnetsecurity.com/2020/05/27/underground-market-trends/

[63] Saeed Shafieian, Mohammad Zulkernine, and Anwar Haque. 2014. Attacks in Public Clouds: Can They Hinder the Rise of the Cloud? In *Cloud Computing*. Springer, 3–22.

[64] Yuru Shao, Jason Ott, Yunhan Jack Jia, Zhiyun Qian, and Zhuoqing Morley Mao. 2016. The Misuse of Android Unix Domain Sockets and Security Implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria, October 24-28, 2016.*

[65] tcpdump. 2021. *TCPDUMP/LIBPCAP public repository.* Retrieved May 7, 2021 from https://www.tcpdump.org/

[66] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Ruales, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin R. B. Butler. 2018. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In *Proceedings of the 27th USENIX Security Symposium (USENIX-Sec), Baltimore, MD, USA, August 15-17, 2018.*

[67] Güliz Seray Tuncay, Jingyu Qian, and Carl A. Gunter. 2020. See No Evil: Phishing for Permissions with False Transparency. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020.*

[68] Timothy Vidas and Nicolas Christin. 2014. Evading Android Runtime Analysis via Sandbox Detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (AsiaCCS), Kyoto, Japan, June 03 - 06, 2014.*

[69] VirtualBox. 2020. *PDM.cpp in vbox/trunk/src/VBox/VMM/VMMR3 – Oracle VM VirtualBox.* Retrieved May 7, 2021 from https://www.virtualbox.org/browser/vbox/trunk/src/VBox/VMM/VMMR3/PDM.cpp

[70] VirtualBox. 2021. *VBoxHeadless, the Remote Desktop Server.* Retrieved May 7, 2021 from https://www.virtualbox.org/manual/ch07.html#vboxheadless

[71] Wireshark. 2021. *Wireshark Go Deep.* Retrieved May 7, 2021 from https://www.wireshark.org/

[72] Daoyuan Wu, Debin Gao, Rocky K. C. Chang, En He, Eric K. T. Cheng, and Robert H. Deng. 2019. Understanding Open Ports in Android Applications: Discovery, Diagnosis, and Security Assessment. In *26th Annual Network and Distributed System Security Symposium, NDSS.*

[73] Lei Wu, Michael C. Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. 2013. The Impact of Vendor Customizations on Android Security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS), Berlin, Germany, November 4-8, 2013.*

[74] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Proceedings of the 21th USENIX Security Symposium (USENIX-Sec), Bellevue, WA, USA, August 8-10, 2012.*

[75] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. 2012. Defending against VM Rollback Attack. In *Proceedings of the 2012 IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), Boston, MA, USA, June 25-28, 2012.*

[76] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. 2012. A Covert Channel Construction in a Virtualized Environment. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS), Raleigh, NC, USA, October 16-18, 2012.*

[77] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. 2016. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *Proceedings of the 25th USENIX Security Symposium (USENIX-Sec), Austin, TX, USA, August 10-12, 2016.*

[78] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh R. Joshi, Matti A. Hiltunen, and Richard D. Schlichting. 2011. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop (CCSW), Chicago, IL, USA, October 21, 2011.*

[79] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. 2011. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland), Berkeley, California, USA, 22-25 May, 2011.*

[80] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS), Raleigh, NC, USA, October 16-18, 2012.*

[81] Min Zheng, Mingshen Sun, and John C. S. Lui. 2014. DroidRay: A Security Evaluation System for Customized Android Firmwares. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (AsiaCCS), Kyoto, Japan, June 03 - 06, 2014.*

[82] Xiao-yong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (Oakland), Berkeley, CA, USA, May 18-21, 2014.*

[83] Chaoshun Zuo, Wubing Wang, Zhiqiang Lin, and Rui Wang. 2016. Automatic Forgery of Cryptographically Consistent Messages to Identify Security Vulnerabilities in Mobile Services. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016.*

[84] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2017. AUTHSCOPE: Towards Automatic Discovery of Vulnerable Authorizations in Online Services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017.*