Towards Transparent and Stealthy Android OS Sandboxing via Customizable Container-Based Virtualization

Wenna Song[1,2], Jiang Ming[3,†], Lin Jiang[4], Yi Xiang[1,2], Xuanchen Pan[5], Jianming Fu[1,2]
Guojun Peng[1,2,†]
[1]School of Cyber Science and Engineering, Wuhan University, China
[2]Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, China
[3]University of Texas at Arlington, USA
[4]Independent Researcher, [5]Wuhan Antiy Information Technology Co.,Ltd, China

ABSTRACT

A fast-growing demand from smartphone users is mobile virtualization. This technique supports running separate instances of virtual phone environments on the same device. In this way, users can run multiple copies of the same app simultaneously, and they can also run an untrusted app in an isolated virtual phone without causing damages to other apps. Traditional hypervisor-based virtualization is impractical to resource-constrained mobile devices. Recent app-level virtualization efforts suffer from the weak isolation mechanism. In contrast, container-based virtualization offers an isolated virtual environment with superior performance. However, existing Android containers do not meet the anti-evasion requirement for security applications: their designs are inherently incapable of providing transparency or stealthiness.

In this paper, we present *VPBox*, a novel Android OS-level sandbox framework via container-based virtualization. We integrate the principle of anti-virtual-machine detection into VPBox's design from two aspects. **First**, we improve the state-of-the-art Android container work significantly for transparency. We are the first to offer complete device virtualization on mainstream Android versions. To minimize the fingerprints of VPBox's presence, we enable all virtualization components (i.e., kernel-level device and user-level device virtualization) to be executed outside of virtual phones (VPs). **Second**, we offer new functionality that security analysts can customize device artifacts (e.g., phone model, kernel version, and hardware profiles) without user-level hooking. This capability prevents the tested apps from detecting the particular mobile device (e.g., Google Pixel phone) that runs an Android container. Our performance evaluation on five VPs shows that VPBox runs different benchmark apps at native speed. Compared with other Android sandboxes, VPBox is the only one that can bypass a set of virtual environment detection heuristics. At last, we demonstrate VPBox's flexibility in testing environment-sensitive malware that tries to evade sandboxes.

† Guojun Peng (guojpeng@whu.edu.cn) and Jiang Ming (jiang.ming@uta.edu) are the corresponding authors.

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/10.1145/3460120.3484544

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security**.

KEYWORDS

Container-Based Virtualization, Android OS Sandboxing, Anti-Evasion

ACM Reference Format:
Wenna Song, Jiang Ming, Lin Jiang, Yi Xiang, Xuanchen Pan, Jianming Fu, Guojun Peng. 2021. Towards Transparent and Stealthy Android OS Sandboxing via Customizable Container-Based Virtualization. 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, 17 pages. https://doi.org/10.1145/3460120.3484544

1 INTRODUCTION

With the proliferation of mobile systems and networks, smartphones are replacing traditional personal computers to fulfill most users' daily computing needs [22, 27]. The trend of Bring Your Own Device [13] has paved the way for another fast-growing demand: mobile virtualization. It allows users to run multiple *separate instances* of smartphone environments on the same physical device. Although Android's multiple users features [7] can switch among different user accounts without leveraging virtualization, researchers have found a significant number of vulnerabilities from this new feature due to its weak isolation mechanism [1, 52]. Especially, mobile apps are now performing various critical tasks such as online payment [76], GPS navigation [68], and IoT device remote control [32]. Inevitably large amounts of private data, such as user credentials and location data, are stored in the smartphone. The rise in risks of data thefts and fraudulent attacks [33, 38] also drives the trend of secure mobile virtualization, which can provide an isolated environment to run untrusted apps and monitor their behaviors.

Resource-constrained mobile devices limit the adoption of traditional hypervisor-based virtualization [14, 21, 59]. Security experts and researchers have been analyzing Android apps dynamically using emulators [41, 43, 53, 64, 75] on top of a PC. However, traditional Android emulators are often slow in performance and leave plenty of fingerprints regarding the runtime environment, hardware effects, and device artifacts. As they are fundamentally different from real devices, a broad spectrum of anti-emulation heuristics has been proposed to detect emulators [16, 30, 34, 45, 50, 55, 69]. The recent progress on app-level virtualization can run multiple copies of the same app within a host app [11, 12, 51]. The most representative one, Parallel Space [42], has been downloaded hundreds of million times. However, their weak isolation mechanism violates the

least-privilege principle, leading to possible permission escalation attacks to guest apps [20, 56, 57, 77, 78].

In contrast, container-based virtualization can potentially overcome the limitations that exist on both hypervisor-based and app-level virtualization. It is lightweight OS-level virtualization that allows several isolated guest virtual machines to run on top of the Operating System (OS) kernel [28, 61]. Because a container is managed by the OS kernel and executes directly on the hardware, it is able to provide a close-to-native virtual environment with high performance. Moreover, as a container does not have software-emulated hardware, the fingerprint of its presence is also minimized [18, 36, 37].

Cells [10] is the first Android container architecture to run multiple virtual phones (VPs) on a single Android instance. However, the design of Cells does not meet the anti-evasion requirement. Its user-level virtualization method introduces non-system components into the VP. The VP's apps, running at the same privilege level as these virtualization components, can find suspicious files and processes via interface scanning. Besides, Cells's virtualization of many devices has been obsolete (e.g., Binder, Network, Display, and Power), and it also lacks support for some essential devices, such as Bluetooth and GPS. Furthermore, Cells is inherently incapable of customizing the VP's device attributes stealthily. All of these limitations (i.e., the deficiency in stealthiness, incomplete device virtualization, and a lack of device attribute customization) can be exploited by adversaries to fingerprint the presence of the VP. Other follow-up container frameworks [17, 71, 73] share similar limitations of lacking transparency and stealthiness, restricting their applications to security-related tasks.

This project seeks to integrate the principle of anti-evasion into the development of a new Android OS-level sandbox, called *VPBox*.[1] We achieve this goal through two contributions: 1) improving the state-of-the-art Android container work significantly so that transparent virtualization works on mainstream Android versions; 2) customizing the virtual phone's device attributes stealthily to bypass the ad-hoc fingerprinting for a specific phone model.

In particular, we improve Cells [10] significantly to achieve the goal of "*out-of-the-box*" virtualization: having no in-guest virtualization component. VPBox consists of kernel-level and user-level device virtualization methods. The kernel-level mechanism enables transparency and performance, and it also paves the way for our novel user-level device virtualization. For the proprietary devices that are entirely closed source (e.g., Bluetooth) and the devices whose configurations happen at user space (e.g., WiFi), we propose a stealthy user-level device virtualization mechanism without compromising transparency. In addition, we take a set of optimization techniques to minimize memory consumption. To enforce a fine-grained access control policy and record system calls invoked, we also virtualize SELinux to enable SELinux settings for each VP.

Although apps are difficult to recognize the difference between VPBox and the underlying physical device, they can still fingerprint the particular smartphone (e.g., Google Pixel phone) that runs VPBox. We address this limitation by allowing users to configure the VP with various device attributes (e.g., phone model and hardware profiles). Unlike existing work that relies on user-level hooking [54], our customization methods are more stealthy because they run outside of the VP. VPBox leverages the new `device` namespace mechanism to isolate the VP's requests from the host's, and it returns the custom parameters to the VP's inquiries. Our isolation design ensures that an app in the VP is unaware of the custom device artifacts. This new feature also enables security applications that require diversified virtual phones, such as analyzing logic bombs [29] that are triggered by particular device artifacts.

VPBox has been tested to support Android versions from 6.0 to 10.0. Our performance experiments, running a set of benchmark apps in up to five VPs on Google Nexus 6P and Pixel 3a XL phones, demonstrate that VPBox introduces negligible runtime overhead and only modest memory consumption. Unlike emulators, VPBox's native performance indicates measuring execution time's variability will fail to detect it. Next, we test Android emulators, app-virtualization sandboxes, and Android containers using mainstream virtual environment detection heuristics [16, 34, 50, 55, 56, 69, 78], such as detecting Android system properties, sensor events, video frame rate, and instruction-level profiles. VPBox is the only one to exhibit the same hardware effects and device artifacts as the underlying physical device. Besides, VPBox is immune to two advanced unsafe environment detection APIs: Google SafetyNet's "basicIntegrity" [5] and ishumei [60]. They can recognize the environment of Android emulators, app-level virtualization, API hooking, and rooted device, but they fail to detect VPBox. At last, we evaluate VPBox's resilience against 1,961 environment-sensitive malware, including samples that try to detect Google phones.

Threat Model. We assume the apps running in the VP are unprivileged user-mode programs. This assumption is also held by bare-metal malware analysis frameworks such as BareBox [39] and BareDroid [48]. That being said, a skilled attacker may exploit a Linux kernel zero-day vulnerability to compromise VPBox. For this reason, we disable the loading of arbitrary kernel modules and prevent user-level apps from accessing kernel memory. §9 will discuss whether VPBox introduces the new artifacts (if any) that can be exploited by adversaries. In a nutshell, our research makes the following contributions.

- **A Transparent Android Container Framework.** VPBox represents the latest progress in mobile container-based virtualization. Our "out-of-the-box" design advances state of the art in transparent device virtualization. Our user-level virtualization solution offers a flexible and stealthy alternative to virtualize new hardware devices without compromising transparency.
- **Device Attribute Customization.** In VPBox, each VP's artifacts are highly customizable without user-level hooking. This new feature offers a cost-effective way to simulate more diversified VPs on a single device. To the best of our knowledge, VPBox offers the most comprehensive device-attribute editing options so far.
- **Open-source Implementation.** VPBox reveals a strong resilience against virtual-machine detection heuristics and device-consistency checks, as well as native performance. VPBox's demo video is available at https://youtu.be/TpGD_jjxSqc. To foster more research on the VPBox platform, we

[1] "VPBox" means running **V**irtual **P**hones as an OS-level Sand**Box**.

release VPBox's source code at (https://github.com/VPBox/Dev).

2 BACKGROUND AND RELATED WORK

We first summarize two common Android virtualization techniques that do not rely on new hardware features (e.g., TrustZone [35]): Android emulators and app-level virtualization. Their deficiencies in lacking transparency and stealthiness have been utilized as effective evasion methods. Next, we discuss the status quo of Android container-based virtualization. It is a new style, lightweight virtualization technique, but the weaknesses of existing containers severely limit their adoptions in security applications. Our project unleashes the power of container-based virtualization to foster strengths and circumvent weaknesses of the current work. At last, we introduce the background about Binder. Our Binder virtualization enables having no in-guest virtualization component at user-level.

2.1 Android Emulators

Android sandboxes based on full-system emulation provide an isolated environment to collect app behaviors [41, 43, 53, 64, 75]. Upon analysis completion, the virtual environment can be restored to a clean snapshot in a matter of seconds. Security analysts typically run Android malware in an emulator to observe malicious behaviors. However, a long-standing challenge of emulators is to virtualize various hardware device effects realistically. It is fundamentally infeasible to make hardware emulation and native hardware indistinguishable [31]. Researchers have proposed a set of detection heuristics to find the hardware-related discrepancies caused by non-transparent system emulation techniques [16, 30, 34, 45, 50, 55, 69], and many of them have been adopted by malware [3]. For example, due to the performance slowdown in graphics rendering emulation, Android emulators typically exhibit a low video frame rate [69]. Petsas et al. detect QEMU-based emulators by checking the virtual program counter update and cache consistency [50]. Bordoni et al. find that sensor-related APIs' return values are different between mobile emulators and real devices [16]. Sahin et al. uncovered instruction-level discrepancies between software-based emulators and real ARM CPUs [55].

In contrast, VPBox's container-based virtualization has a unique advantage in the transparent virtualization effect: VPBox does not have software-emulated hardware, and the foreground VP can always access hardware devices and run apps at native speed.

2.2 App-Level Virtualization

The recent app-virtualization development (e.g., VirtualApp [11], DroidPlugin [51], and Parallel Space [42]) provides a more lightweight option to run multiple copies of the same app on a single device, such as accessing Facebook simultaneously with two different accounts. The key idea is that a host app provides a virtual environment on top of the Android framework, and it creates system service proxies to launch arbitrary guest apps from their APK files without installation. Due to the dynamic proxy hooking, the actions from a guest app will be treated by the Android system as the host app's actions. In this way, two copies of the same app are able to bypass the UID restriction and execute at the same time.

Despite the growing popularity of app-virtualization-based apps in the Android market, researchers have realized the security problems caused by this new technical progress [20, 56, 57, 77, 78]. As all guest apps share the same UID with the host app, the current design introduces a serious "shared-everything" threat to guest apps [56], which has made malicious attacks such as permission escalation and privacy leakage tremendously easier. Although guest apps can also directly access the Android device that installs the host app, the host app has to hook API invocations of the guest app so that the Android system thinks that all API requests and components are from the host app. However, the hooking mechanism leaves many host app's signatures in the guest app's call stack and memory; DiPrint [56] utilizes these signatures to detect the presence of an app-virtualization environment.

By contrast, VPBox can achieve the same goal of running multiple instances of the same app simultaneously, but with a stronger isolation mechanism among virtual phones and the host device. Furthermore, VPBox's virtualization and customization do not adopt user-level API hooking and thus have better stealthiness than app-level virtualization.

2.3 Android Container-Based Virtualization

The container-based virtualization reveals distinct benefits in performance and transparent hardware virtualization effects. Initial investigations on Linux Containers [18, 36] and Docker [37] have shown that container-based virtualization is very promising to defeat emulator-aware malware. However, these works [18, 36, 37] did not deliver a functional mobile virtualization platform, and many important topics, such as how to hide new artifacts introduced by containers and an extensive evaluation with existing anti-virtual-machine heuristics, are still missing. Our research bridges this gap.

Challenges. Compared with the rise of container-based virtualization in PC and server platforms [2, 26, 72], Android container's development has to overcome the challenge of hardware resource multiplexing. Especially, many mobile devices are physically not designed for multiplexing (e.g., WiFi and Bluetooth). For the Android OS, at least the devices and pseudo-device drivers listed in Table 1 must be fully supported. However, none of the existing Android containers can meet this goal. Besides, to facilitate a rapid transplantation and upgrade of the Android system, Android 8.0 re-architected the vendor interface in the Android OS framework [46]. This new update invalidates existing virtualization methods on multiple devices, such as Telephony, Display, Network, and Binder.

Another take-away message from Table 1 is that none of the existing work can meet the "out-of-the-box" design; that is, all of them have in-guest virtualization components that run at the same privilege level as the VP's apps. As a result, it is trivial to detect whether an app is running in these containers by scanning suspicious non-system files and processes. Furthermore, as shown in Table 1's gray color row, no existing work can customize the VP's device attributes. As we will present in §7, stealthy customization is impossible without the "out-of-the-box" virtualization design.

Cells [10]. Cells is the pioneering work of mobile container-based virtualization. Limited by the small-scale touchscreen, Cells introduces a usage model of having one foreground VP and other VPs running in the background. The VP running in the foreground

Table 1: The comparison of devices, pseudo-device drivers, and services among five Android container-based virtualization solutions. Cellrox [17] is Cells's commercial version. The labels ○/● indicate the virtualization is missing (○) or enabled (●). ◐ means the virtualization has been outdated in mainstream Android versions. The marks ✓/✗ indicate the device virtualization meets the "out-of-the-box" design (✓) or not (✗).

	Description	Cells [10]	Cellrox [17]	Condroid [73]	VMOS [71]	VPBox
Device/Pseudo Device[1]						
Display	Display screen graphics	◐, ✓	◐, ✓	◐, ✗	◐, ✗	●, ✓
Filesystem	SD card partition virtualization	◐, ✓	◐, ✓	◐, ✓	◐, ✓	●, ✓
Power	Power management	◐, ✓	◐, ✓	◐, ✗	◐, ✗	●, ✓
Binder	Inter-process communication	◐, ✓	◐, ✓	◐, ✗	◐, ✗	●, ✓
Input	Touchscreen and input buttons	●, ✓	●, ✓	◐, ✗	◐, ✗	●, ✓
Network	Core network resources	◐, ✓	◐, ✓	◐, ✗	◐, ✗	●, ✓
WiFi	Wireless connection configuration	◐, ✗	◐, ✗	◐, ✗	○	●, ✓
Telephony	Incoming/outgoing calls	◐, ✗	◐, ✗	○	○	●, ✓
GPU	Graphics processing unit	◐, ✓	◐, ✓	○	◐, ✗	●, ✓
Sensors	Light sensor and accelerometer	●, ✓	●, ✓	○	◐, ✗	●, ✓
Camera	Video and still-frame input	○	○	◐, ✗	◐, ✗	●, ✓
Audio	Speakers, microphone	○	○	◐, ✗	○	●, ✓
GPS	Global positioning system	○	○	○	○	●, ✓
Bluetooth	Short-range wireless communication	○	○	○	○	●, ✓
ADB	Command-line utility for debugging	○	○	○	○	●, ✓
Service						
Multiple virtual phones		●	●	●	○	●
Reduce memory consumption		◐	◐	◐	○	●
Security-Enhanced Linux in Android		○	○	○	○	●
Device attribute customization		○	○	○	○	●
The latest Android version supported		4.0.3	5.1	4.4.2	5.1	10.0

[1]Pseudo-device drivers (e.g., Binder) are parts of the kernel that act like device drivers but do not correspond to any actual hardware.

is displayed at any time and is always given direct access to hardware devices. Cells invents a new `device` namespace mechanism to support efficient hardware resource multiplexing, and each VP is associated with a unique `device` namespace for device interactions. In addition to the kernel-level virtualization, Cells also integrates user-level device virtualization methods to handle proprietary devices with closed software stacks. Unfortunately, many of Cells's device virtualization methods are either incompatible with new Android versions or leave in-guest components. Furthermore, it also lacks some essential device virtualization solutions that are indispensable to a malware sandbox. For example, as no existing work can virtualize Bluetooth because of its complexity, malware can easily check Bluetooth profiles (e.g., Bluetooth MAC Address) to differentiate a sandbox from a real machine [4, 69].

Condroid [73] & VMOS [71]. Condroid and VMOS, two follow-up Android containers, share similar limitations with Cells in transparency and customization. Condroid transplants the Linux Container tools [44] to Android and makes the most of the modifications at the Android framework layer; it ensures the isolation of containers by leveraging namespaces and cgroups. VMOS runs another Android OS as the guest operating system by mounting the virtual root file system and virtualizing the JAVA runtime. VMOS's virtual system and the host phone share the host's native libraries to access hardware devices. Compared with Cells, VMOS's implementation is simpler, but at the cost of a weaker container isolation mechanism. Except for the mount namespace, VMOS's virtual system and the host device have the same namespaces, which cannot isolate operating system resources.

Summary. The existing Android containers are not qualified to be an OS-level sandbox for security applications. Their limitations on outdated/incomplete device virtualization, having in-guest components, and a lack of device customization, can all be exploited by attackers as new fingerprints to detect the presence of Android containers. Our work delivers a novel Android container platform with strong anti-evasion capability, even to a dedicated adversary.

2.4 Binder

Binder is the Android-specific inter-process communication (IPC) mechanism and the remote method invocation system. Binder consists of Binder driver, ServiceManager, server, and client. The Binder driver is a pseudo device in the kernel and does not correspond to the physical device. Userspace processes supporting the Binder communication will create corresponding Binder data structures (e.g., `binder_proc`, `binder_node`) in the kernel to maintain the process state. The server is the Binder service. ServiceManager is a special Binder service. The Android kernel creates a global `binder_node` object `binder_context_mgr_node` in the Binder driver to indicate that it is the Binder service manager and set `handler=0` for other client processes to call. When the client (e.g., App) requests the Binder service, it will obtain the ServiceManager service by calling the handler with a value of 0. ServiceManager finally queries the `binder_node` from the server according to the list of services it maintains and binds it to `binder_node` from the client to implement the client-to-server binder interface call.

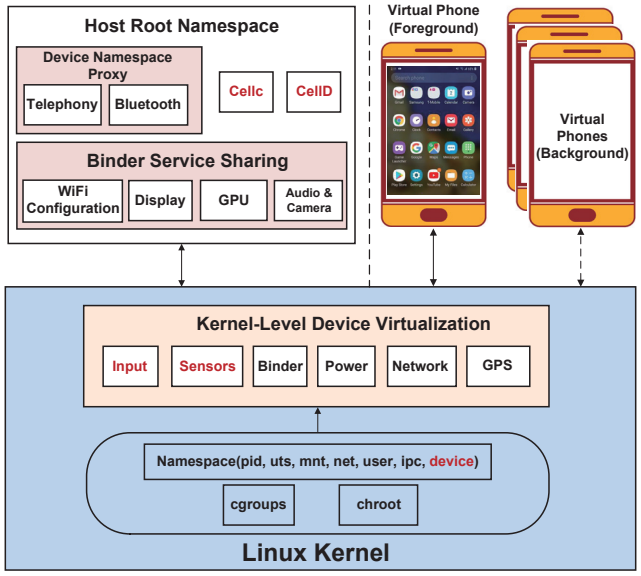


Figure 1: Overview of VPBox's architecture. The names in red represent Cells's modules reused by VPBox.

3 VPBOX SYSTEM OVERVIEW

VPBox is a transparent and stealthy Android OS-level sandbox via a novel, customizable container-based virtualization technique. To enable the application to security related tasks, VPBox's design is capable of meeting the following two progressive requirements:

(1) **Transparency.** This requirement involves two aspects: a) the virtualized device exhibits the same hardware effects as the underlying physical device; b) complete virtualization support for all devices and services listed in Table 1.
(2) **Stealthiness.** On top of the transparency, this requirement ensures a dedicated adversary in the VP is difficult to fingerprint the presence of the container, including the presence of virtualization components and the particular mobile device that runs the container.

Existing Android containers partially meet the transparency requirement due to their incomplete device virtualization, but no one satisfies the stealthiness requirement. The last column of Table 1 shows VPBox's advantages. All of the devices and services listed in Table 1 are fully supported by VPBox, including hardware devices, pseudo-device drivers, and necessary services to the Android system (e.g., Bluetooth, ADB, and SELinux). Security analysts are free to configure different device artifacts and then boot up diversified virtual environments. To achieve the goal of stealthiness, we enable our device virtualization and the customization of device-specific attributes to be executed outside of VPs.

Figure 1 provides an overview of VPBox's device virtualization. VPBox retains the foreground-background VP usage model of Cells [10]. Each isolated VP runs a stock Android userspace environment. The VP running in the foreground can always have direct access to hardware devices. VPBox utilizes Linux namespaces and the `device` namespace introduced by Cells to remap OS resource identifiers to VPs. Each VP has its private namespace so that it does not interfere with the other VPs and the host. The names in red in Figure 1 represent Cells's modules reused by VPBox. We reuse Cells's kernel-level virtualization methods that still work in the latest Android version, including Input (e.g., touchscreen and input buttons) and Sensors (e.g., accelerometer and light sensors). The virtualization of Input and Sensors is to modify a device subsystem to be aware of the `device` namespace. We also keep two custom processes, "CellC" and "CellD," in the host device's root namespace; they manage the service of booting up a VP and switching VPs between the foreground and background. CellD also coordinates our ADB virtualization. We add a control center app for VPBox users to start and switch VPs swiftly. More importantly, we improve Cells in four significant ways to meet our requirements on transparency and stealthiness.

(1) We design kernel-level device virtualization to be compatible with device changes in the new Android systems. Our method makes it possible to have no in-guest virtualization component for our user-level device virtualization(§4).
(2) We propose a novel user-level virtualization mechanism, which offers a flexible and stealthy solution to virtualize new hardware devices without compromising transparency (§5).
(3) We take new measures to reduce memory consumption and enable SELinux settings for each VP (§6).
(4) We provide a broad spectrum of options to customize the VP's device attributes stealthily. This enables us to simulate more diversified VPs on a single device (§7).

4 KERNEL-LEVEL DEVICE VIRTUALIZATION

Kernel-level device virtualization provides efficient hardware resource multiplexing, and it is also transparent to user-mode apps running in VPs. Our kernel-level mechanism enables the virtualization of Binder, power management, core network resource, and GPS on mainstream Android versions. Our key method is to rewrite the source code of kernel drivers to be aware of the `device` namespace. Next, we use Binder and GPS as examples to present the strategy of our kernel-level device virtualization. We put core network resource and power management virtualization in Appendix A.

Binder. Binder allows high-level framework APIs to cross process boundaries and interact with Android system services. The Binder driver consists of three pseudo-device drivers. In addition to the traditional "/dev/binder" driver, the Android system adds another two Binder drivers since Android 8.0: "/dev/hwbinder" and "/dev/vndbinder"; they are used for IPC between framework/vendor processes and IPC between vendor/vendor processes [8]. Without Binder virtualization, Binder's IPC feature can be abused by different container processes, violating the system isolation between containers. In VPBox, we have modified all Binder drivers' data structures to enable IPC between two processes that share the same `device` namespace. Binder driver virtualization is the foundation of our new user-level device virtualization technique (see §5), which allows a service process in the VP to share the corresponding service in the host system and leaves no virtualization component in the VP's userspace.

GPS. GPS provides a more accurate positioning service than network positioning, but existing Android containers do not support GPS virtualization. GPS relies on a physical chip for location tracking. In the Android framework layer, the GPS provider,

GpsLocationProvider, calls the hardware abstraction layer (HAL) interface via Java Native Interface methods. The HAL interface interacts with the GPS chip through "/dev/gss" driver. The GPS chip is an active tracking device. After a user's first request, the GPS chip will continue to report the location information to GpsLocationProvider without interruption. However, the GPS chip only supports one connection. Our virtualization of GPS is to rewrite "/dev/gss" driver to support multiple connections. We modify "gss_open" and "gss_event_output" functions so that the location information received from the chip is forwarded to multiple clients simultaneously. The location information goes through HAL and eventually reaches GpsLocationProvider in the Android framework layer of the host and virtual phones, respectively.

5 USER-LEVEL DEVICE VIRTUALIZATION

User-level device virtualization is necessary because some hardware vendors provide proprietary software stacks that are completely closed source. Without hardware vendor's support, it would be difficult, if not impossible, to virtualize them in the kernel. VPBox's user-level virtualization achieves the goal of *having no in-guest virtualization component* by developing two new methods, which enable the VP space to retain native-like system components.

1. Binder Service Sharing For the system services registered in ServiceManager (e.g., WifiService & SurfaceFlinger), we propose a new, general virtualization technique via Binder service sharing. We first modify the Binder-driver data structure (e.g., context_mgr_node, procs, and dead_nodes) to ensure that each VP has its own Binder-driver data structure. Next, we create a new specific handler in Binder's data structure and let it point to the host's context_mgr_node. As context_mgr_node is associated with ServiceManager, with this handler, the VP can access the host phone's ServiceManager node. Therefore, this mechanism allows a VP'S service process to share the corresponding service in the host system. Then, we leverage the SELinux technique to enforce which services the VP can share in the host system. In VPBox, we use Binder service sharing to virtualize WiFi configuration, Display, GPU, Audio, and Camera.

2. Device Namespace Proxy We cannot apply Binder service sharing to the anonymous services not registered in ServiceManager, such as telephone and Bluetooth, because their binder_node and binder_ref kernel structures are missing. Therefore, we develop a new device namespace proxy to virtualize telephone and Bluetooth, leaving no in-guest virtualization component. Cells's user-level virtualization is not stealthy. It creates each VP's own proxy, connecting to CellD running in the host's root namespace. CellD, in turn, communicates related hardware vendor libraries to respond to the VP's requests. However, Cells's proxy layer is located at the VP's application framework layer. Like API hooking, apps running in the VP can easily detect the presence of Cells's proxy layer because they share the same privilege level. We address the stealthiness concern by creating a device namespace proxy in the host's userspace only.

Next, §5.1 and §5.2 explore the method of Binder service sharing, and §5.3 takes Bluetooth as an example to present the method of device namespace proxy. We put the details of other user-level devices' virtualization (telephony, filesystem, and ADB) in Appendix B and Appendix C.

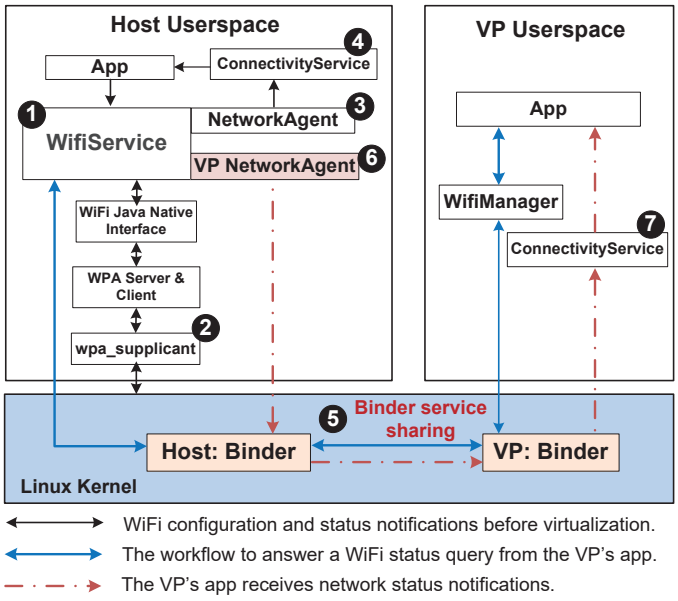


Figure 2: VPBox's WiFi configuration virtualization.

5.1 WiFi Configuration

WiFi configuration and status notifications occur at the userspace. We use Binder service sharing for its virtualization. Compared with Cells's method, our approach is simpler and leaves no virtualization component in the VP's userspace. Cells's WiFi virtualization is not stealthy because it adds a WiFi poxy inside each VP. In contrast, our virtualization occurs at the host's userspace and the kernel. Figure 2 illustrates our design. In the Android system, before virtualization, WifiService (❶) calls the library of "wpa_supplicant" (❷) to detect WiFi connections. The "wpa_supplicant" library is a user-level library that contains wireless network service code. The WiFi-connection information is sent through NetworkAgent (❸) to ConnectivityService (❹), which answers app queries about the state of network connectivity. To virtualize WiFi, we use the binder service sharing mechanism (❺) to bridge WifiService between the VP and the host. The blue double arrow line in Figure 2 shows the workflow to respond to a WiFi status query from the VP's app. In addition, we create a new NetworkAgent in the host and bind it to the VP's device namespace (❻). As shown by the red dotted line, we also use Binder service sharing to connect the new NetworkAgent (❻) with the VP's ConnectivityService (❼). The purpose of doing so is to automatically forward network status notifications (e.g., WiFi signal strength) to the VP.

5.2 Display, GPU, Audio, and Camera

Display, GPU, Audio, and Camera are all virtualized via the Binder service sharing mechanism while ensuring the isolation between VPs. We use Display and GPU as examples to describe the design. The virtualization methods of Audio and Camera are similar by sharing the Binder service of AudioFlinger and CameraService.

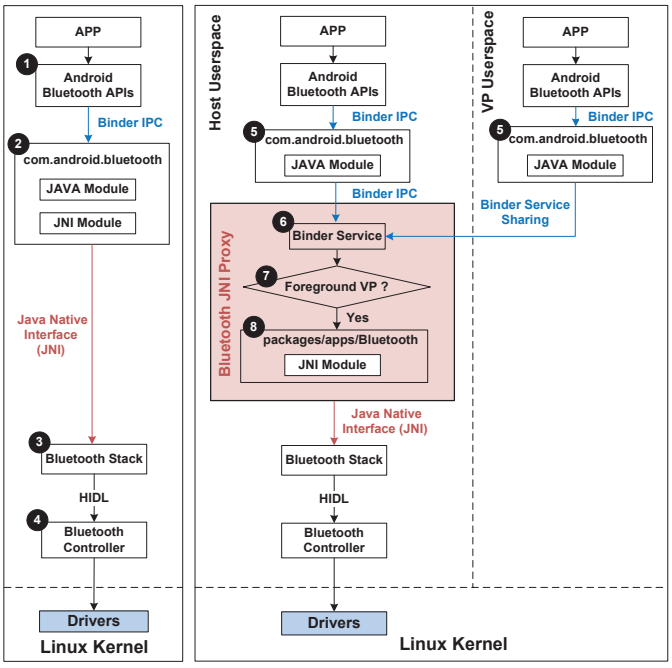


Figure 3: VPBox's Bluetooth virtualization.

The Display is an essential device in smartphones, and the GPU provides hardware display acceleration. Before Android 6.0, the Android system takes Linux framebuffer (FB) as an abstraction to a physical display and screen memory. Cells virtualizes FB by multiplexing FB's device driver. However, Android 6.0 and later versions have switched to the ION driver for managing the screen memory. Modifying the ION driver to virtualize FB is error-prone and complicated.

To solve this issue, we take advantage of the Binder service sharing to enable each VP to multiplex an essential graphics service—SurfaceFlinger of the host system. SurfaceFlinger is responsible for compositing all of the application and system surfaces into a single framebuffer for a final display. Also, we adapt related data structures, graphics rendering APIs, and interfaces for virtualization. 1) We add the system tag field in the Layer data structure to detect to which system (VP or host) the Layer belongs. 2) With the added system tag, we identify the foreground system layer from SurfaceFlinger's APIs, such as layer cropping and compositing, to display the final image on the screen. 3) To switch the screen between the VP and host, we add new interfaces for clearing and redrawing images in SurfaceFlinger. Our design is two-birds-one-stone because no additional measures are needed for GPU virtualization. Since the VP multiplexes the host system's screen memory buffer, the host's GPU can directly work on it for display acceleration. Additionally, to properly support the foreground-background usage model, we limit the SurfaceFlinger service to only respond to the request from the foreground VP, ignoring the requests from background VPs.

5.3 Bluetooth

None of the existing Android emulators or containers can virtualize Bluetooth. Each smartphone manufacturer provides its own proprietary Bluetooth vendor code that is entirely closed source. Without the hardware vendor's support, the kernel-level virtualization of Bluetooth would be very challenging. Figure 3(a) shows the Android Bluetooth architecture since Android 8.0. To use the Bluetooth service, an app first calls Android Bluetooth APIs (❶), which further sends the request to the Bluetooth service process (❷) via Binder IPC. Next, the Bluetooth service process connects to the Bluetooth stack (❸) via Java Native Interface (JNI). Then, the Bluetooth stack interacts with the Bluetooth controller (❹) using Hardware Interface Design Language (HIDL).

Bluetooth service process (❷) only provides the anonymous Binder service externally, which does not submit the registered Binder to the ServiceManager. This means we cannot apply the binder service sharing mechanism to ❷. Instead, we implement a new service proxy to virtualize Bluetooth. Figure 3(b) illustrates our workflow. We modify the Bluetooth app ("packages/apps/Bluetooth") and embed a Bluetooth JNI proxy. After our modification, the Bluetooth service process now only contains the JAVA module (❺), and the original JNI module is put into the newly added Bluetooth JNI proxy (❽) in the host. Now, it is our Bluetooth JNI proxy to interact with the Bluetooth stack and the Bluetooth controller.

Furthermore, to enable our proxy to communicate with new Bluetooth service processes (❺) in the host and each VP, we also build a binder service in the Bluetooth JNI process (❻). In this way, each VP can finally access the Bluetooth driver in the host device. Please note that the Bluetooth driver does not support multiplexing. An exception will happen if multiple connections are established with the Bluetooth driver at the same time. Therefore, we add a namespace check in our proxy (❼): we only forward the foreground VP's Bluetooth service request. Our SELinux policy specifies user-level apps in each VP have no privilege to access the new Bluetooth service process (❺) to detect our change.

6 SCALABILITY AND SELINUX

When running multiple VPs on VPBox, memory usage becomes the scalability bottleneck. We modify related kernel functions and data structures to support three memory optimization techniques: advanced multi-layered unification filesystem (AUFS) [49], Linux kernel same-page merging (KSM) [65], and Android low memory killer [6]. We use the AUFS mechanism to mount the read-only partition of the VP system to reduce the load storage of the device. However, there is no AUFS module in the Android system. We first transplant the AUFS module from its `git` repository to the "/fs" directory of the Android kernel source code. Then, we modify the file operation interface of the kernel, such as `d_walk`, `setfl`, `sync_filesystem`, and related data structures, to complete the adaption. KSM is a module in the Android kernel, but we need to activate it to support multiple container instances. KSM is a memory-saving de-duplication feature. The major modifications we made include enabling "/sys/kernel/mm/ksm/run=1" and "CONFIG_KSM=y", configuring the values of `sleep_millisecs`, `pages_to_scan`, and other parameters in "kernel/mm/ksm.c" module based on the terminal hardware configuration and the number

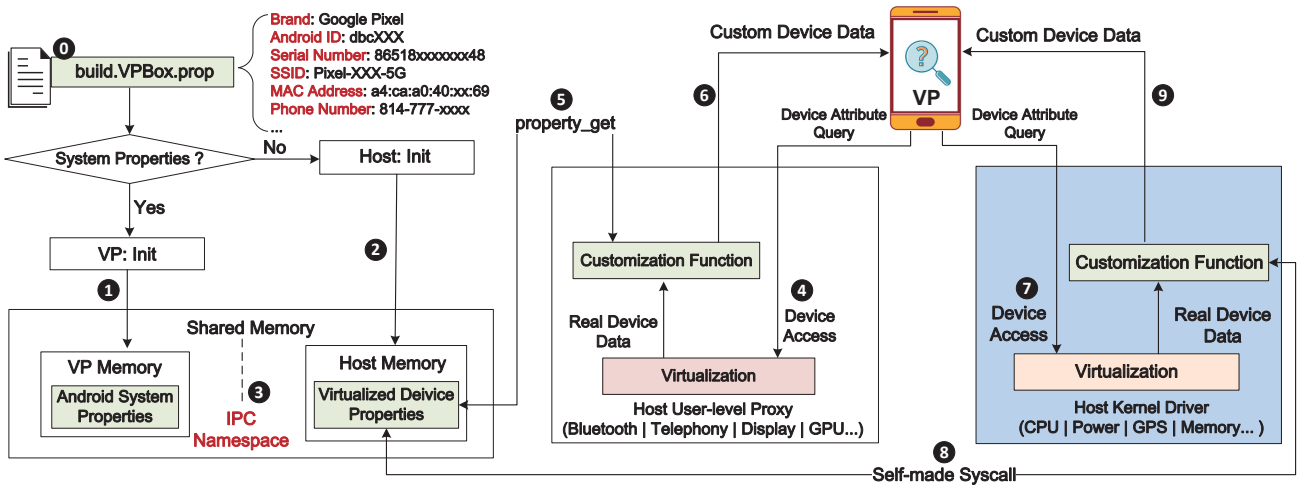


Figure 4: VPBox's workflow of customizing the VP's device attributes.

of container instances. We virtualize the kernel driver of Low Memory Killer so that each VP can independently use this mechanism to manage the process memory of its namespace. We mainly modified the process `task_struct` data structure to bind the device namespace to identify the VP that different processes belong to. In particular, we modified "kernel/drivers/staging/android/lowmemorykiller.c" module, registered a `lowmem_shrinker` memory callback for each VP, and configured the scheduling strategy so that the background VP's `lowmem_shrinker` has more execution opportunities than the foreground VP's `lowmem_shrinker`.

Moreover, we also provide an optional, "screen off" function for background VPs to further improve scalability. In this way, when a VP is switched to the background, its power model will become the same as pressing the power button of the native Android system. Turning off the screen causes each component to stop unnecessary services, processes, and threads, which further reduces memory consumption.

To isolate all VPs from the host machine and one another, we utilize three kinds of namespaces (UID, `device`, and mount) to enforce the access control on user credentials, data, device state, and filesystem. Also, we disable the capability of creating device nodes inside a VP. Furthermore, we add a fine-grained permission strategy that monitors a VP's internal processes in real time. We modify the host's SELinux policy to take different VP's namespaces as new labels and create new SELinux access control strategies for each VP's internal processes. In this way, we can prevent untrusted apps from abusing the VP's device access permissions.

Inspired by BareDroid [48], we also take advantage of SELinux to record the system calls invoked during app execution. By default, only denied operations are recorded by SELinux. We modified the SELinux policy by adding an `auditallow` tag to each authorized operation. In this way, we can collect complete operations performed by a user app.

7 DEVICE ATTRIBUTE CUSTOMIZATION

Our virtualization techniques attempt to provide VPBox users with the same experiences as using a physical smartphone. However, a dedicated adversary can still detect the particular device running VPBox, such as the Google phones we used. Even bare-metal malware analysis frameworks, such as BareBox [39] and BareDroid [48], are still susceptible to ad-hoc fingerprinting techniques by examining specific software/hardware environment features. To address this issue, we go one step further to enable customizing VP's device attributes. Our "out-of-the-box" virtualization design enables the device attribute customization to preserve stealthiness. This new feature offers a cost-effective way to simulate more diversified virtual environments (e.g., Xiaomi Redmi series and Huawei Honor series) on a single device. Figure 4 shows the workflow of our proposed device-attribute customization. VPBox users provide a configuration file "build.VPBox.prop" in advance (❶ in Figure 4), which stores device-specific attributes in the form of key-value pairs. We classify these key-value pairs into three categories: Android system properties, user-level-virtualized device properties, and kernel-level-virtualized device properties. Each category has a different customization method. The strategy and advantages of our customization are explored next.

7.1 Android System Property Customization

Android system properties are const values that describe the mobile device's configuration information, such as brand, model, serial number, IMEI, and manufacturer. These properties are stored in the init process's shared memory, but they are independent of our device virtualization. This shared memory is typically used to store some system and hardware information when the system is being initialized. Other processes enquire about Android system properties at runtime by calling "property_get" (❺ in Figure 4), an API for other processes to read the data stored in the shared memory space. Therefore, during the process of booting up the VP, its init process will call "load_system_props" to load the custom Android system properties from "build.VPBox.prop" into the VP's shared memory space (❶). Then, the custom system properties are ready for apps running in the VP to access and inquire.

7.2 User-level Customization

The second category of "build.VPBox.prop" contains the device attributes that we customize for user-level-virtualized devices, such

as Bluetooth, WiFi, and telephony. The customized data in the second category will be loaded into the host init process's shared memory (❷). We enforce the IPC namespace to isolate the host's and VP's shared memory (❸). We embed a customization function in the place where we perform user-level device virtualization, such as Bluetooth JNI proxy and Telephony RilD proxy. In particular, the customization function takes effect after the virtualization function has responded to the app's device attribute query request (❹). The customization function first determines whether the current query request is from the VP by checking the associated `device` namespace. If the query is from the VP's app, it calls "property_get" (❺) to get the custom data from the shared memory that maps "build.VPBox.prop", and then it returns the custom device data to the VP (❻).

7.3 Kernel-level Customization

The third category contains key-value pairs used to customize kernel-level-virtualized devices, such as Power and GPS. Besides, certain kernel drivers contain basic device attributes (e.g., kernel version and memory/processor information), which are included in the third category of our customized data as well. These kernel-related configuration data are also stored in the host init process's shared memory.

In the kernel driver, we embed a customization function at the place where our kernel-level virtualization function has responded to the app's device access request (❼). The customization functions need to interact with the shared memory of the host's init process. However, the customized data in the init process have no privilege to enter the kernel space. To overcome this obstacle, we create a new system call to copy data from the userspace to the kernel space (❽). All of our customization functions in the kernel drivers work similarly. For example, we customize the battery-related profiles (e.g., battery level) in the kernel power driver and use the `device` namespace to determine whether the query request is from the VP or the host. If the request is from the VP, the Power's customization function will call our created syscall to extract the custom data and then return them to the VP (❾).

However, we have to take special measures to customize kernel version information, of which two attributes are defined in the UTS namespace data structure ("UTS_RELEASE" and "UTS_VERSION"). We need to modify the UTS namespace data structure to embed our customization function. Interested readers are referred to Appendix D for more details.

7.4 Advantages of VPBox's Customization

VPBox now provides 150 device configuration options, which span a broad spectrum of device attributes. Appendix Table 6 lists customizable device-attribute options. We collect them from 1) the related work on Android sandbox detection, and 2) existing device-attribute editing tools. To the best of our knowledge, VPBox's device-attribute customization options are the largest and most comprehensive so far.

Existing Android device-attribute editing tools [15, 74] are built on Xposed [54] by hooking APIs. Compared to them, VPBox reveals two distinct advantages. **First**, our customization methods are more stealthy, because they occur at internal data structures or internal

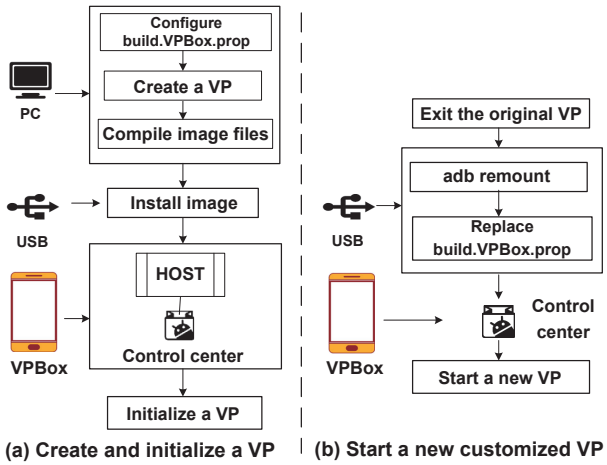


Figure 5: The workflow of starting a custom virtual phone.

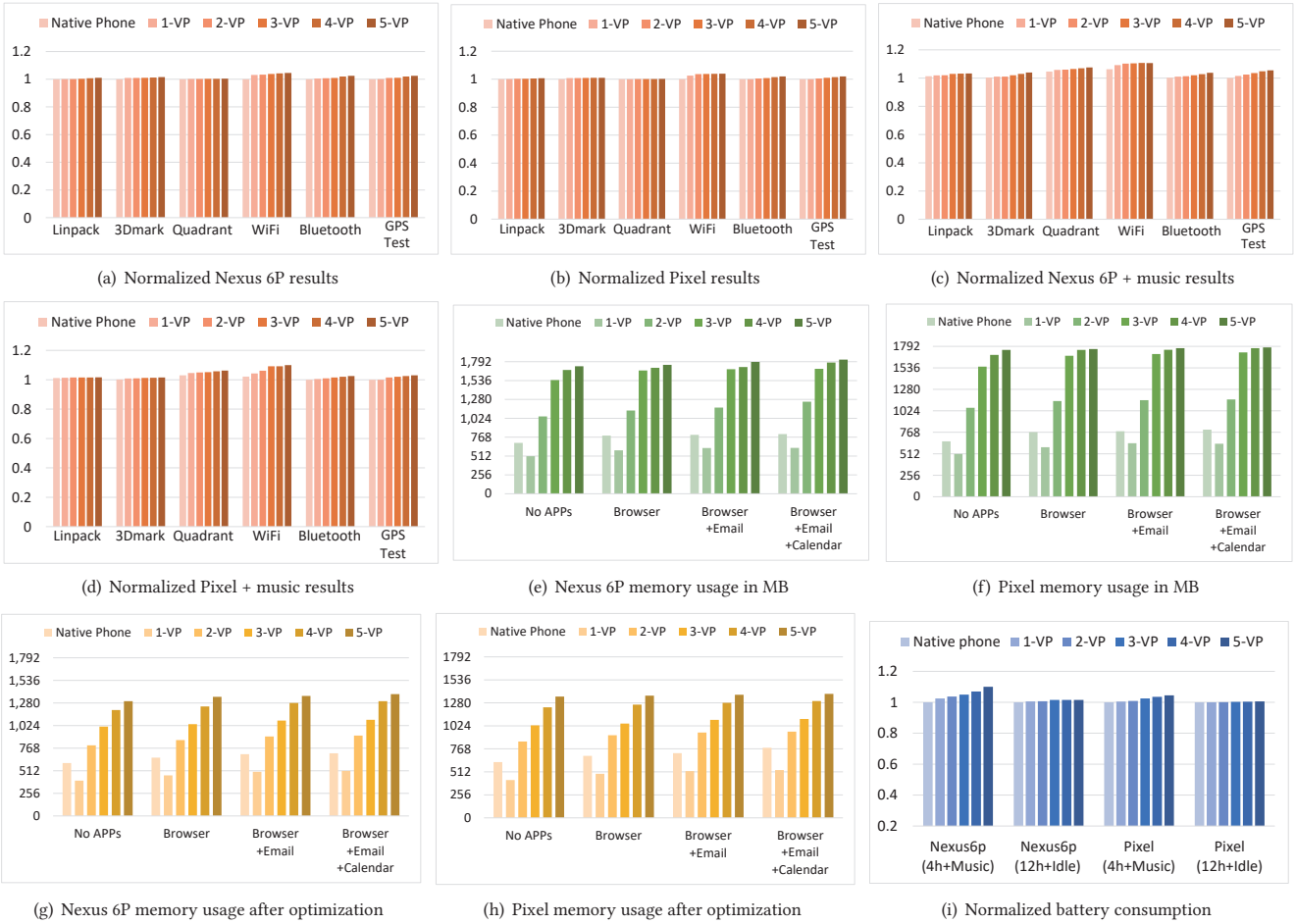
interfaces that are inaccessible to the virtual phone. Besides, they do not rely on user-level API-hooking, which means our customization does not leave footprints in the VP's runtime environment.

Second, our VP's customization does not interfere with normal operations on the host device. System modifications without leveraging container virtualization lack flexibility and compatibility, because only changing return values of APIs or syscalls is likely to result in system crashes or exceptions. For example, blindly editing Bluetooth attributes would cause the Bluetooth system service to keep restarting, affecting the app that is using the Bluetooth service. In VPBox's customization functions, we do not use the custom device data to respond to all device access requests. Instead, we analyze the data flow of the VP interface that accesses the device. Only if the device data obtained by the VP interface finally flows into the VP's process, and the process UID is a user app, we send the custom device data to the VP interface. Powered by our "out-of-the-box" virtualization design, VPBox can gracefully decouple device-attribute editing operations from normal operations on the host device and solve incompatibility issues.

8 EVALUATION

VPBox Usage The VP images are created and configured on a PC and downloaded to the host device via USB. We provide a control center app for VPBox users to switch between the host system and VPs swiftly. To start a new custom VP, a user takes the following three steps: 1) exit the original VP; 2) update and replacing a new "build.VPBox.prop" configuration file; 3) stat a new VP via the control center app. Figure 5 shows how to start a new custom VP.

We evaluate VPBox from three dimensions. First, we provide performance measurements to show that VPBox reveals native performance. Second, we compare existing Android sandboxes in evading various virtual environment detection heuristics. The third experiment evaluates VPBox's device customization using environment-sensitive malware. Please note that we are unable to compare VPBox with other peer Android containers in the performance test. The complete source code download links of Cells [10], Cellrox [17], and Condroid [73] have been out of work for a while, so we cannot compile and run their virtual phones. We can only



(a) Normalized Nexus 6P results (b) Normalized Pixel results (c) Normalized Nexus 6P + music results

(d) Normalized Pixel + music results (e) Nexus 6P memory usage in MB (f) Pixel memory usage in MB

(g) Nexus 6P memory usage after optimization (h) Pixel memory usage after optimization (i) Normalized battery consumption

Figure 6: VPBox's performance measurements on Google Nexus 6P and Pixel 3a XL phones.

run one VP using VMOS [71] on Android 5.1, but VMOS's 1-VP and VPBox's performance data on multi-VPs are not comparable.

8.1 Performance Measurements

We measure runtime overhead, memory usage, and power consumption using two different Google Phones: Nexus 6P (1.55 GHz Cortex-A53, Adreno 430 GPU, 3G RAM, and 32G ROM) and Pixel 3a XL (2.15 GHz Kryo, Adreno 530 GPU, 4G RAM, and 32G ROM). We follow similar experimental settings as Cells's paper in SOSP'11 [10]. We measured the performance of VPBox when running 1VP, 2VPs, 3VPs, 4VPs, and 5VPs, each with a fully booted Android environment. Our runtime overhead measurement contains two scenarios. The first one is running a set of benchmark apps on VPBox's VPs and a native phone, respectively. The second one is running the same benchmark apps on VPs and the native phone, but simultaneously with an additional background music player workload. The benchmark application is always run in the foreground VP; if the background workload is used, it runs in a single background VP when multiple VPs are started. The results of runtime overhead are normalized against the performance of running the same benchmark apps on the latest manufacturer stock image available for two Google phones, but without the background workload.

Benchmark Apps. Each benchmark app is designed to stress some aspect of the system performance: Linpack (v1.1) for CPU; 3DMark (v2.0.4646) for 3D graphics; Quadrant advanced edition (v2.1.1) for 2D graphics and file I/O; WiFi using BusyBox wget (v1.21.1) to download a 409M video file through a PC's WiFi hotspot; Bluetooth measurement is the time that the Bluetooth module takes to transfer a 1M file between two paired Bluetooth devices; and GPS performance is measured by the time that the GPS Test app (v1.6.3) takes to acquire the GPS location.

Runtime Overhead. Figure 6(a) & 6(b) show the normalized runtime overhead on two Google phones with no background workload running. The deviations between "n-VP" and "Native Phone" represent the additional overhead caused by VPBox's device virtualization. The negligible deviations in most cases indicate no user-noticeable performance difference between running in VPBox and running natively on a real phone, even with up to five VPs running simultaneously. The WiFi benchmark shows the largest overhead—it introduces about 3%~6% additional slowdowns on two phones. In addition to VPBox's virtualization, we argue that WiFi variability levels could also affect network performance. Figure 6(c) & 6(d) show the normalized runtime overhead when running the background music player. As would be expected, running

the background workload causes additional overheads relative to our baseline. Among all of our benchmarks, 3DMark shows the least overhead because playing music does not involve 3D rendering. Compared to Cells's performance data [10], VPBox reveals the same level of variability in runtime overhead.

Memory Usage. Figure 6(e) & 6(f) show the default memory usage (without memory optimization) on two phones. The "No Apps" measures the memory usage after booting each VP but running no apps. Then, we measure the memory usage after starting an instance of Chrome browser, Gmail client, and Google Calendar in each running VP. Apparently, after starting multiple VPs, memory usage becomes the scalability bottleneck. Note that VPBox requires incrementally less memory when starting more VPs. The reason is Android low memory killer; even without specific memory optimization methods, it automatically takes effect to kill background processes and free memory for new applications. Figure 6(g) & 6(h) show results after we apply kernel same-page merging, file system unioning, and "screen off" for background VPs. With these memory optimization methods enabled, we can further reduce memory consumption by 100MB to 600MB.

Power Consumption. Figure 6(i) shows the normalized power consumption on two Google phones; the larger value means more power consumption. The "4h + Music" measures the power consumption after playing the music repeatedly for 4 hours. When multiple VPs exist, we run the music player in the foreground VP. Compared with the native phone, the power consumption results from 1-VP to 3-VP increase by less than 5%, and the power consumption results from 4-VP to 5-VP increase by less than 10%. The Pixel 3a XL phone's power measurement is better than the Nexus 6p phone, because Pixel 3a XL phone's power management has been improved. The "12h + Idle" measures the power consumption after 12 hours in an idle state. Compared to the native phone, VPBox's numbers in "12h + Idle" show no measurable increase.

Conclusion. VPBox's superior performance data indicate that it is immune to the evasions that measure the performance gap between virtual machines and real devices.

8.2 Security Analysis

Our second experiment evaluates the resilience against virtual environment detection heuristics proposed by the previous work [16, 34, 50, 55, 56, 69, 78]. The superset of them covers the mainstream Android virtual environment detection heuristics. Table 3 presents the results under Android emulators, app-level virtualization, and Android container environments. Row 2 to 10 are nine types of Android virtual environment detection heuristics, and their descriptions are presented in Table 2. We first run these detection heuristics in a physical device and save their results as Output 1. Then, we install seven different virtualization environments listed in Row 1 on this device and then rerun these detection methods in seven virtual environments, respectively. After that, we compare their outputs with Output 1. A transparent virtual environment should show no appreciable difference with its underlying device.

VPBox meets this goal from two aspects: 1) the virtualized device exhibits the same hardware effects as the underlying physical device; 2) VPBox's virtualization supports all devices and services listed in Table 1. Besides, to achieve the goal of stealthiness, we enable our device virtualization and the customization of device specific attributes to be executed outside of VPs. It ensures a dedicated adversary in the VP is diffcult to fingerprint the presence of VPBox, including the presence of virtualization components.

Table 2: Android virtual environment detection heuristic types and their descriptions.

Type	Description
Emulated Network [69]	The emulated network environment is typically different from that of physical devices, such as IP address, virtual router, and host loopback.
CPU & Graphical Performance [69]	1) Calculate 1,048,576 digits of Pi; 2) measure video frame rate
Hardware Components [69]	E.g., Bluetooth, Radio, and Power management.
Sensor Events [16]	Detect accelerometer API return values.
Hypervisor Heuristics [50]	1) Virtual program counter update; 2) cache consistency
Instruction-level Profiles [55]	Software-based emulators reveal different instruction-level behaviors from real ARM CPUs when processing undefined instructions.
Android APIs [34, 69]	Many APIs return unique device identifiers.
System Properties [34]	Android system configurations and status.
Shared UID & Hooking [56, 78]	In app-level virtualization, the host app shares the same UID with all guest apps and relies on hooking to hide guest apps' API requests.

Hardware-related Discrepancies. Row 2 to Row 7 focus on detecting the hardware-related discrepancies caused by virtualization. Android emulators are easy to be detected because of software-emulated hardware and slow performance. VirtualApp and Parallel Space reveal the same hardware effects because guest apps can still directly access the underlying Android device's hardware. VMOS fails five times in the category of "Hardware Components." The reason is VMOS lacks virtualization support for WiFi, Telephony, Audio, GPS, and Bluetooth. VMOS's results imply that incomplete device virtualization can also be exploited by adversaries to fingerprint the presence of a virtual phone.

Device Artifacts. Row 8 and Row 9 represent the detection of device artifacts. Software-based emulators exhibit different values in some Android system properties, and many APIs return unique device identifiers. For app-level virtualization, to run multiple copies of the same guest apps simultaneously, the host app (e.g., VirtualApp) has to *intentionally* reveal some different device artifacts (e.g., Android ID) to each guest app instance. As VMOS's device virtualization is not complete, it also returns ten different API values about device identifiers, such as `TelephonyManager.getLine1Number()`. In contrast, as VPBox's foreground VP can directly access the hardware, it reveals the same device artifacts as the physical device.

App-level Virtualization. Row 10 detects two characteristics of app-level virtualization [56, 78]: 1) Shared UID between the host app and guest apps; 2) API-hooking mechanism. The three Android emulators also adopt API-hooking as an analysis approach. By contrast, only VPBox succeeds in bypassing all detection heuristics. Because each VP has its private namespace so that it does not interfere with the other VPs and the host. Besides, VPBox does not rely on user-level API-hooking, which means our virtualization does not leave hook footprints in the VP's runtime environment.

Table 3: The results of anti-virtual-environment detection. For the results like "X/Y", Y is the total number of detection heuristics, and X is the number of effective ones. For the results of SafetyNet and ishumei, ○ means a tool successfully detects this virtual environment, and ● means it treats this virtual environment as a genuine Android device. For each evasive malware family, the value in "()" is the number of samples, and we represent the number of file operations generated by each evasive malware family as (min, max, median).

Detection Heuristics	Android Emulator			App-Level Virtualization		Android Container	
	DroidScope [75]	CuckooDroid [53]	DroidBox [41]	VirtualApp [11]	Parallel Space [42]	VMOS [71]	VPBox[1]
Emulated Network [69]	5/5	5/5	5/5	0/5	0/5	0/5	0/5
Performance [69]	2/2	2/2	2/2	0/2	0/2	0/2	0/2
Hardware Components [69]	11/13	13/13	13/13	0/13	0/13	**5/13**	0/13
Sensor Events [16]	9/9	9/9	9/9	0/9	0/9	0/9	0/9
Hypervisor Heuristics [50]	2/2	2/2	2/2	0/2	0/2	0/2	0/2
Instruction-level Profiles [55]	6/6	6/6	6/6	0/6	0/6	0/6	0/6
Android APIs [34, 69]	38/47	47/47	40/47	**22/47**	**16/47**	**10/47**	0/47
System Properties [34]	10/10	10/10	10/10	0/10	0/10	0/10	0/10
UID & Hooking [56, 78]	1/4	1/4	1/4	**4/4**	**4/4**	0/4	0/4
SafetyNet-bI [5]	○	○	○	○	○	○	●
ishumei [60]	○	○	○	○	○	○	●
Evasive Malware (1, 961)							
Rotexy (273) [58]	(9, 77, 25)	(9, 73, 10)	(9, 73, 16)	(18, 24, 19)	(0, 0, 0)	(24, 113, 43)	(50, 170, 74)
Ashas (152) [63]	(0, 27, 0)	(0, 20, 0)	(0, 23, 0)	(0, 46, 29)	(0, 0, 0)	(51, 92, 63)	(65, 99, 75)
HeHe (145) [23]	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 6, 0)	(0, 0, 0)	(0, 31, 17)	(26, 44, 30)
Ztorg (143) [67]	(0, 37, 25)	(0, 30, 20)	(0, 30, 20)	(0, 31, 26)	(0, 0, 0)	(0, 59, 3)	(40, 63, 45)
Andr RuSms-AT (217) [62]	(0, 29, 0)	(0, 20, 0)	(0, 25, 0)	(4, 41, 10)	(0, 0, 0)	(17, 79, 36)	(48, 157, 74)
OBAD (290) [66]	(26, 52, 40)	(20, 40, 30)	(25, 49, 35)	(16, 50, 26)	(0, 6, 0)	(38, 87, 70)	(78, 101, 94)
Android.BankBot (290) [40]	(0, 176, 50)	(0, 118, 30)	(0, 149, 24)	(1, 224, 71)	(0, 0, 0)	(6, 211, 93)	(64, 250, 101)
GhostClicker (442) [25]	(24, 94, 41)	(12, 47, 23)	(24, 65, 35)	(56, 192, 92)	(0, 59, 35)	(85, 273, 97)	(108, 392, 125)
G-Ware[2] (9) [9]	(0, 10, 8)	(0, 5, 3)	(0, 9, 7)	(0, 24, 3)	(0, 0, 0)	(5, 31, 7)	(127, 160, 150)

[1]All VPBox's anti-virtual-machine detection experiments are performed in the foreground virtual phone.
[2]In addition to detecting virtual environments, G-Ware family also detects Google phones.

Commercial Detection Tools. Google SafetyNet [5] and ishumei [60] are two representative anti-abuse/anti-fraud APIs. They help developers to determine whether their apps are running on a genuine Android device. The "SafetyNet-bI" in Table 3 represents SafetyNet's "basicIntegrity" verdict. SafetyNet's "basicIntegrity" and ishumei can identify the signs of a rooted device, emulator, and API hooking. Our results show that both of them are able to recognize all of the tested Android emulators, VirtualApp, Parallel Space, and VMOS, but they fail to detect VPBox. Please note that SafetyNet also provides another more stringent Android compatibility testing, called "ctsProfileMatch." It detects genuine but uncertified devices, certified devices with an unlocked bootloader, and devices with custom ROM. VPBox does not pass the verdict of "ctsProfileMatch," because we have to unlock the bootloader to flash VPBox's image. We argue that this is not a specific limitation caused by our virtualization system. We download the complete Android 6.0-10.0 system source code, compile them in Ubuntu, and then flash them on mobile devices—all of them cannot pass the check of "ctsProfileMatch" either. Also, many top phone manufacturers run a custom ROM in their products, such as the MIUI system in Xiaomi smartphones.

Evasive Malware. We collect 1, 961 environment-sensitive malware samples (9 families) from our industry collaborator. Security analysts have confirmed that all of these samples try to detect virtual machines. For example, HeHe [23] variants detect whether they are being run in an emulator by checking the IMEI, phone number, operator, and phone model. Like BareDroid [48], we also use the number of file operations as a quantitative measurement for the amount of malicious behaviors. Under each virtual environment, we follow CopperDroid's targeted stimulation strategy [64] to stimulate a running malware sample for 1 minute and monitor its behaviors. The last nine rows of Table 3 show the number of file operations generated by each evasive malware family—they are strikingly different between VPBox and the others. The number "0" indicates that the sample crashed upon start; we attribute this to the successful detection of virtual environment. Apparently, evasive malware samples either crashed upon start or performed a very small number of file operations under emulators and app-level virtualization tools. In contrast, no evasive malware sample crashed under VPBox, and malware exhibits much more behaviors than other sandboxes. For example, we find most malware crashed in Parallel Space [42], and the variants of HeHe [23] and Ztorg [67] can detect all virtual environments except for VPBox.

8.3 VP Customization Evaluation

We conduct a separate experiment to compare evasive malware behaviors in VPBox and physical Google phones. Although most malware samples reveal the same behaviors in VPBox and physical Google phones, we do find an exception for G-Ware [9]. Upon further investigation, we find that, in addition to evading virtual machines, G-Ware family also avoids running in Google phones. G-Ware samples first retrieve the device's system property. If the string of "http.agent" or "Manufacturer" contains "Pixel," "Nexus,"

Table 4: Number of file operations generated by G-Ware in Google phones (Pixel 3a XL and Nexus 6p) and four different custom VPs. We customize the four VPs as Xiaomi RedmiNote 4 (VP1), Xiaomi Redmi Note 4x (VP2), Huawei Honor 6x (VP3), and Huawei Honor 8 (VP4).

Samples	Real Devices	VP1	VP2	VP3	VP4
G-Ware1	11	147	143	139	141
G-Ware2	10	157	152	142	157
G-Ware3	8	139	134	141	149
G-Ware4	8	153	149	150	151
G-Ware5	7	133	127	132	131
G-Ware6	26	152	146	147	153
G-Ware7	9	146	147	141	157
G-Ware8	23	160	159	147	151
G-Ware9	12	141	157	150	146

or "google," G-Ware's malicious activities will not be triggered. We construe this behavior as a way to evade Android's built-in Application Sandbox or bare-metal analysis framework, because Android's built-in sandboxing environments are usually named as "Nexus XXX" or "Pixel YYY," and the bare-metal analysis framework like BareDroid [48] is also built on a specific Google Nexus phone. We list G-Ware samples' MD5 values in Appendix Table 7.

Our device attribute customization functionality prevents malware from fingerprinting the underlying mobile device that runs VPBox. We configure our VPs as four different phones: Xiaomi RedmiNote 4 (VP1), Xiaomi Redmi Note 4x (VP2), Huawei Honor 6x (VP3), and Huawei Honor 8 (VP4). In particular, we edit customizable device-attribute options (shown in Appendix Table 6) as the same values of the target phone. After that, we rerun G-Ware malware in custom VPs to monitor their behaviors. As shown in Table 4, all G-Ware samples exhibit much more file operations in VPs than in physical Google phones. Besides, we observe the same behaviors across the four VP environments, such as calling "setComponentEnabledSetting" to hide the current App icon and then stealthily downloading new malicious packages.

9 DISCUSSION AND FUTURE WORK

A natural question to VPBox is whether a skilled attacker can easily detect the presence of the new Android container once it is publicly known. We do not assume that evading our approaches is strictly impossible, but they can prohibitively increase attackers' cost. We acknowledge that VPBox introduces some specific artifacts, such as never-changing geographical location and `device` namespace. However, these artifacts can be hidden by VPBox's unique feature on device attribute customization and its fine-grained SELinux policy. As some devices' virtualization methods happen at the host userspace, if an app in the VP has the root privilege, it can find out the corresponding service processes are incomplete. For example, the VP's Bluetooth service process does not interact with its own Bluetooth stack and Bluetooth controller. However, our design pushes attackers from attempting to fingerprint a virtual machine or a very specific mobile device, to attempting to exploit privilege escalation vulnerabilities to root devices. We believe this to be a non-trivial task even for skilled attackers.

VPBox now provides 150 device customization options, but we cannot guarantee that our list is complete. The arms race here is that an attacker could detect the existence of VPBox's underlying Google phone by checking the consistency of some obscure device properties, but finding all of them is an open problem. It is worth noting that only the foreground VP shows the full strength in bypassing virtual-machine detection heuristics. Some devices (e.g., Bluetooth and ADB) in background VPs are disabled because they are physically not designed for multiplexing. Therefore, the best strategy to run untrusted apps or evasive malware is executing them in the foreground VP.

Reverse Turing Test. A new trend of evading virtual environment is the so-called "Reverse Turing Test" by detecting human interactions [19, 24, 47]. For example, Miramirkhani et al. [47] propose using the "wear and tear" artifacts that typically occur on devices of real users, but not virtual devices, to detect malware sandboxes. The authors [47] also developed a statistical model to help build virtual machine images that exhibit more realistic "wear-and-tear" characteristics. Their findings help further improve the fidelity of VPBox by customizing the VP with the "wear-and-tear" artifacts.

Dynamic Malware Analysis. VPBox shows promise as a sandbox for dynamic malware analysis. Currently, system call invocation tracking is ready via SELinux virtualization. With the `device` namespace and our custom SELinux policy, we can capture system calls pertaining to the malware process. However, system calls alone have been questioned to depict high-level Android-specific semantics [64, 75]. Next, inspired by CopperDroid's out-of-the-box approach [64], we will reconstruct malware behaviors from low-level system events, leaving no in-guest behavior analysis components. We always perform malware analysis in the foreground VP, and all background VPs are customized in a clean state. Upon analysis completion, a background VP is switched to the foreground to start the next round of malware analysis.

10 CONCLUSION

In this paper, we characterize, research, and evaluate VPBox, a new Android container-based virtualization framework. VPBox provides a transparent virtual phone environment and allows users to customize the virtual phone's device attributes stealthily. Currently, VPBox is the only Android container framework that can work on mainstream Android versions. Our experiments demonstrate that VPBox introduces negligible runtime overhead and reveals strong resilience against various virtual machine detection heuristics. VPBox has been deployed into a production environment to assist security professionals in identifying evasive malware.

ACKNOWLEDGMENTS

We sincerely thank CCS 2021 anonymous reviewers for their insightful comments and Dr. Srdjan Capkun for helping us improve the paper throughout the shepherding process. This research was supported in part by the National Natural Science Foundation of China (62172308, U1626107, 61972297, 62172144), and Jiang was supported by the National Science Foundation (NSF) under grant CNS-1850434 and CNS-2128703.

REFERENCES

[1] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. 2018. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS'18)*.

[2] Theodora Adufu, Jieun Choi, and Yoonhee Kim. 2015. Is Container-based Technology a Winner for High Performance Scientific Applications?. In *Proceedings of the 17th Asia-Pacific Network Operations and Management Symposium*.

[3] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. 2019. Malware Dynamic Analysis Evasion Techniques: A Survey. *Comput. Surveys* 52, 6, Article 126 (November 2019), 28 pages.

[4] Vitor Afonso, Anatoli Kalysch, Tilo Müller, Daniela Oliveira, André Grégio, and Paulo Lício de Geus. 2018. Lumus: Dynamically Uncovering Evasive Android Applications. In *Proceedings of the 21st International Conference on Information Security (ISC'18)*.

[5] Android Developers. [online]. SafetyNet Attestation API. https://developer.android.com/training/safetynet/attestation.

[6] Android Open Source Project. 2019. Low Memory Killer Daemon (lmkd). https://source.android.com/devices/tech/perf/lmkd.

[7] Android Open Source Project. [online]. Supporting Multiple Users. https://source.android.com/devices/tech/admin/multi-user/.

[8] Android Open Source Project. [online]. Using Binder IPC. https://source.android.com/devices/architecture/hidl/binder-ipc.

[9] androidcentrol. 2018. G-Ware Virus. https://forums.androidcentral.com/ask-question/885223-g-ware-virus-app-not-deleting.html.

[10] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. 2011. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*.

[11] asLody. 2015. VirtualApp. https://github.com/asLody/VirtualApp.

[12] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. 2015. Boxify: Full-fledged App Sandboxing for Stock Android. In *Proceedings of the 24th USENIX Conference on Security Symposium (USENIX Security'15)*.

[13] Rafael Ballagas, Michael Rohs, Jennifer G Sheridan, and Jan Borchers. 2004. BYOD: Bring Your Own Device. In *Proceedings of the 6th International Conference on Ubiquitous Computing (UbiComp'04)*.

[14] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. 2010. The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket? *ACM SIGOPS Operating Systems Review* 44, 4 (2010).

[15] Antonio Bianchi, Eric Gustafson, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2017. Exploitation and Mitigation of Authentication Schemes Based on Device-Public Information. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC'17)*.

[16] Lorenzo Bordoni, Mauro Conti, and Riccardo Spolaor. 2017. Mirage: Toward a Stealthier and Modular Malware Analysis Sandbox for Android. In *Proceedings of the 22th European Symposium on Research in Computer Security (ESORICS'17)*.

[17] Cellrox ltd. [online]. Cellrox Mobile Virtualization. https://www.cellrox.com/.

[18] Ngoc-Tu Chau and Souhwan Jung. 2018. Dynamic analysis with Android Container: Challenges and Opportunities. *Digital Investigation* 27 (2018).

[19] Valerio Costamagna, Cong Zheng, and Heqing Huang. 2018. Identifying and Evading Android Sandbox Through Usage-Profile Based Fingerprints. In *Proceedings of the First Workshop on Radical and Experiential Security*.

[20] Deshun Dai, Ruixuan Li, Junwei Tang, Ali Davanian, and Heng Yin. 2020. Parallel Space Traveling: A Security Analysis of App-Level Virtualization in Android. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies (SACMAT'20)*.

[21] Christoffer Dall and Jason Nieh. 2014. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

[22] David Pierce. 2018. Your Smartphone Is the Best Computer You Own. The Wall Street Journal, http://tiny.cc/cqsnpz.

[23] Hitesh Dharmdasani. 2014. Android.HeHe: Malware Now Disconnects Phone Calls. https://www.fireeye.com/blog/threat-research/2014/01/android-hehe-malware-now-disconnects-phone-calls.html.

[24] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. 2016. Evading Android Runtime Analysis Through Detecting Programmed Interactions. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec'16)*.

[25] Echo Duan and Roland Sun. 2017. GhostClicker Adware: a Phantomlike Android Click Fraud. http://tiny.cc/7w5ctz.

[26] Michael Eder. 2016. Hypervisor- vs. Container-based Virtualization. In *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications*.

[27] Eric Enge. 2019. Mobile vs. Desktop Usage in 2019. https://www.perficient.com/insights/research-hub/mobile-vs-desktop-usage-study.

[28] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software*.

[29] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. TriggerScope: Towards Detecting Logic Bombs in Android Applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*.

[30] Jyoti Gajrani, Jitendra Sarswat, SMeenakshi Tripathi, Vijay Laxmi, M.S. Gaur, and Mauro Conti. 2015. A Robust Dynamic Analysis System Preventing SandBox Detection by Android Malware. In *Proceedings of the 8th International Conference on Security of Information and Networks (SIN'15)*.

[31] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. 2007. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HOTOS'07)*.

[32] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. 2016. Smart Locks: Lessons for Securing Commodity Internet of Things Devices. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIACCS'16)*.

[33] John Høegh-Omdal. 2020. StrandHogg 2.0 - The 'evil twin', New Android Vulnerability Even More Dangerous, With Attacks More Difficult to Detect Than Predecessor. https://promon.co/strandhogg-2-0/.

[34] 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'14)*.

[35] Uri Kanonov and Avishai Wool. 2016. Secure Containers in Android: The Samsung KNOX Case Study. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'16)*.

[36] Alexander Kedrowitsch, Danfeng (Daphne) Yao, Gang Wang, and Kirk Cameron. 2017. A First Look: Using Linux Containers for Deceptive Honeypots. In *Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense*.

[37] Ayrat Khalimov, Sofiane Benahmed, Rasheed Hussain, S.M. Ahsan Kazmi, Alma Oracevic, Fatima Hussain, Farhan Ahmad, and Chaker Abdelaziz Kerrache. 2019. Container-Based Sandboxes for Malware Analysis: A Compromise Worth Considering. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC'19)*.

[38] I Luk Kim, Yunhui Zheng, Hogun Park, Weihang Wang, Wei You, Yousra Aafer, and Xiangyu Zhang. 2020. Finding Client-side Business Flow Tampering Vulnerabilities. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*.

[39] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2011. BareBox: Efficient Malware Analysis on Bare-Metal. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)*.

[40] Mohit Kumar. 2019. New Android Malware Apps Use Motion Sensor to Evade Detection. https://thehackernews.com/2019/01/android-malware-play-store.html.

[41] Patrik Lantz. 2015. Dynamic Analysis of Android Apps. https://github.com/pjlantz/droidbox.

[42] LBE Tech. [online]. Parallel Space: Run Multiple Social and Game Accounts in Your Phone Simultaneously. http://parallel-app.com/.

[43] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. 2014. ANDRUBIS – 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*.

[44] Linux Containers. [online]. Infrastructure for Container Projects. https://linuxcontainers.org/.

[45] Dominik Maier and Mykola Protsenko. 2014. Divide-and-Conquer: Why Android Malware Cannot Be Stopped. In *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES'14)*.

[46] Iliyan Malchev. 2017. Here comes Treble: A modular base for Android. https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html.

[47] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. 2017. Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*.

[48] Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. 2015. BareDroid: Large-Scale Analysis of Android Apps on Real Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*.

[49] Junjiro R. Okajima. [online]. Advanced Multi Layered Unification Filesystem. http://aufs.sourceforge.net/.

[50] 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 7th European Workshop on System Security (EuroSec'14)*.

[51] Qihoo360. 2015. DroidPlugin. https://github.com/DroidPluginTeam/DroidPlugin.

[52] Paul Ratazzi, Yousra Aafer, Amit Ahlawat, Hao Hao, Yifei Wang, and Wenliang Du. 2014. A Systematic Security Evaluation of Android's Multi-User Framework. In *Proceedings of the Mobile Security Technologies (MOST'14)*.

[53] Idan Revivo and Ofer Caspi. 2016. CuckooDroid - Automated Android Malware Analysis. https://github.com/idanr1986/cuckoo-droid.

[54] rovo89. [online]. Xposed Module Repository. https://repo.xposed.info/.

[55] 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'18)*.

[56] Luman Shi, Jianming Fu, Zhengwei Guo, and Jiang Ming. 2019. "Jekyll and Hyde" is Risky: Shared-Everything Threat Mitigation in Dual-Instance Apps. In *Proceedings of the 17th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys'19)*.

[57] Luman Shi, Jiang Ming, Jianming Fu, Guojun Peng, Dongpeng Xu, Kun Gao, and Xuanchen Pan. 2020. VAHunt: Warding Off New Repackaged Android Malware in App-Virtualization's Clothing. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS'20)*.

[58] Tatyana Shishkova and Lev Pikman. 2018. The Rotexy Mobile Trojan —- Banker and Ransomware. https://securelist.com/the-rotexy-mobile-trojan-banker-and-ransomware/88893/.

[59] Junaid Shuja, Abdullah Gani, Kashif Bilal, Atta Ur Rehman Khan, Sajjad A. Madani, Samee U. Khan, and Albert Y. Zomaya. 2016. A Survey of Mobile Device Virtualization: Taxonomy and State of the Art. *Comput. Surveys* 49, 1 (April 2016).

[60] shumei. [online]. ishumei Android Device Security Threat Detection SDK. https://www.ishumei.com/product/bs-post-sdk.html.

[61] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*.

[62] SophosLabs. 2017. Android Malware Anti-emulation Techniques. http://tiny.cc/s416tz.

[63] Lukas Stefanko. 2019. Tracking down the developer of Android adware affecting millions of users. https://www.welivesecurity.com/2019/10/24/tracking-down-developer-android-adware/.

[64] Kimberly Tam, Salahuddin J. Khan, Aristide Fattoriy, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS'15)*.

[65] The kernel development community. [online]. Kernel Samepage Merging. https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html.

[66] Roman Unuchek. 2013. The most sophisticated Android Trojan. https://securelist.com/the-most-sophisticated-android-trojan/35929/.

[67] Roman Unuchek. 2017. Ztorg: money for infecting your smartphone. https://securelist.com/ztorg-money-for-infecting-your-smartphone/78325/.

[68] Steven J. Vaughan-Nichols. 2009. Will Mobile Computing's Future Be Location, Location, Location? *Computer* 42, 2 (2009).

[69] 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'14)*.

[70] Timothy Vidas, Daniel Votipka, and Nicolas Christin. 2011. All Your Droid Are Belong to Us: A Survey of Current Android Attacks. In *Proceedings of the 5th USENIX Conference on Offensive Technologies (WOOT'11)*.

[71] VMOS Inc. [online]. Virtual Android on Android . http://www.vmos.com/.

[72] Miguel G. Xavier, Marcelo V. Neves, Fabio D. Rossi, Tiago C. Ferreto, Timoteo Lange, and Cesar A. F. De Rose. 2013. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*.

[73] Lei Xu, Guoxi Li, Chuan Li, Weijie Sun, Wenzhi Chen, and Zonghui Wang. 2015. Condroid: A Container-Based Virtualization Solution Adapted for Android Devices. In *Proceedings of the 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud'15)*.

[74] XxsqManage. 2019. The Best Tool to Change Android Phone's Configuration. http://www.javaer.xyz/XxsqManager/html/index.html.

[75] Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*.

[76] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. 2017. Show Me the Money! Finding Flawed Implementations of Third-party In-app Payment in Android Apps. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS'17)*.

[77] Lei Zhang, Zhemin Yang, Yuyu He, Mingqi Li, Sen Yang, Min Yang, Yuan Zhang, and Zhiyun Qian. 2019. App in the Middle: Demystify Application Virtualization in Android and its Security Threats. In *Proceedings of the 45th International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'19)*.

[78] Cong Zheng, Wenjun Hu, and Zhi Xu. 2018. Android Plugin Becomes a Catastrophe to Android Ecosystem. In *Proceedings of the 1st Workshop on Radical and Experiential Security (RESEC'18)*.

APPENDIX

A CORE NETWORK RESOURCE AND POWER MANAGEMENT

We reuse most of Cells's kernel-level work to virtualize core network resources such as network adapters, IP addresses, and port numbers. However, the Android system has been updated significantly since Android 6.0. It adopted the so-called "policy routing" to work with multiple routing tables and rules. Policy routing defines which traffic a specific routing table is used for. Therefore, we need to come up with a new virtualization method. We extend Cells by configuring `ndc` and `iptables` commands to add new rules for policy routing. As WiFi configuration management happens in the userspace, we adopt the binder service sharing to virtualize WiFi configuration (see §5.1).

In power management virtualization, VPBox reuses Cells's solution in *wake-locks* virtualization but manages *early suspend* completely differently. Since Android 6.0, the *early suspend* subsystem has been replaced by SurfaceFlinger's `setPowerMode` interface to manage display's on/off-screen, which invalidates Cells's virtualization that modifies the *early suspend* subsystem to recognize device namespaces. By contrast, we virtualize SurfaceFlinger service at the user level (see §5.2). We only need to prevent background VPs from putting the foreground VP into a low power mode via the `setPowerMode` interface.

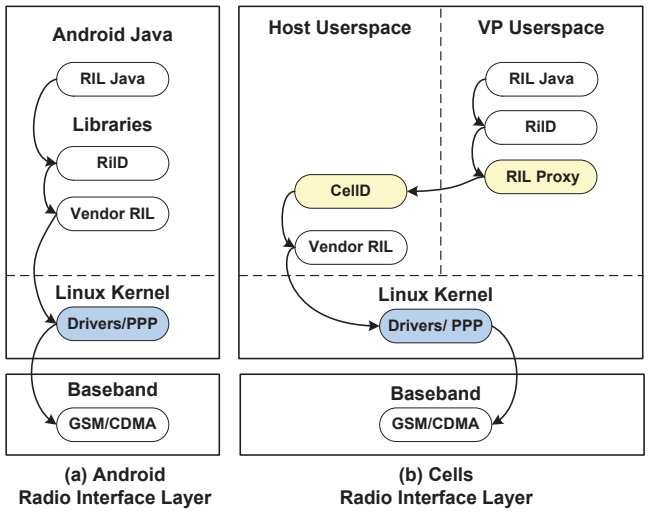


Figure 7: Cells's Radio Interface Layer (RIL). Cells's RIL proxy is visible to apps running in the VP.

B TELEPHONY VIRTUALIZATION VIA RILD PROXY

As smartphone vendors customize their own proprietary radio stack, we adopt a user-level device namespace proxy to virtualize the telephony in the VP. The previous solution proposed by Cells is not stealthy, because its proxy is located in the VP's userspace and visible to apps running in the VP. We show Cells's Radio Interface Layer in Figure 7. By contrast, we design a socket-interface-based

proxy that only presents at the host userspace. As shown in Figure 8, in the host's Radio Interface Layer, we create a Radio Interface Layer Daemon (RilD) proxy between the communication flow of Android telephony Java libraries (RIL Java) and RilD. Then, we create two standard Unix Domain sockets in the proxy. One socket connects to the RIL Java of each VP; the other connects to the RIL Java of the host system. The RIL Java in each VP communicates with the proxy of the host system, and the proxy passes the communication data (e.g., dial request and SIM) to the host system's RilD. The RilD proxy also passes the VP-related arguments (e.g., call ring and signal strength) to the VP's RIL Java over a socket. In this way, we provide a separate telephony functionality for each VP. In addition, we customize the SELinux-based device access control strategy to ensure that private call data (e.g., incoming/outgoing call information and voice data) pertaining to a specific VP cannot be accessed by other VPs.

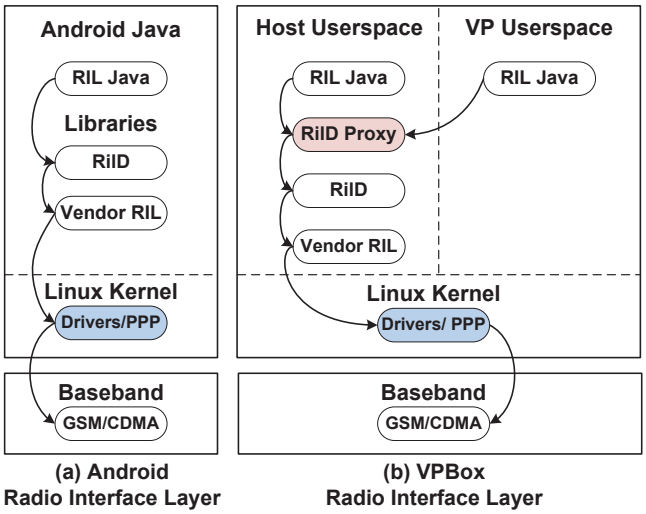


Figure 8: VPBox's Radio Interface Layer (RIL).

Table 5: Common Radio Interface Layer(RIL) commands.

Type	Commands
Solicited	dial request, get current calls, SIM I/O, set screen state, set radio state
Unsolicited	signal strength, call ring call state changed

Once a VP's phone function is enabled, the VP can make/receive calls and access phone hardware information, such as international mobile subscriber identity (IMSI) and mobile equipment identifier (MEID). VPBox disables the telephony functionality for VPs that have no telephony access. In addition, when the foreground VP is making or receiving calls, other background VPs cannot make/receive calls even if they have the telephony functionality. To properly support the foreground-background usage model, RIL commands shown in Table 5 require filtering from background VPs or special handling. We take the same special handling with Cells.

C FILESYSTEM AND ANDROID DEBUG BRIDGE

Existing Android containers' SD card partition virtualization does not comply with the new SD card access management starting from Android 6.0, which introduces Filesystem in Userspace (FUSE) technology to manage the SD card partition. Recent Android versions directly fork a process in the Volume Daemon (Vold) subsystem and start the sdcard process to mount the FUSE filesystem. Because the FUSE module supports file system creation in userspace, and the VP in VPBox runs complete userspace, we take the following two steps to virtualize SD card partition: 1) open a "dev/fuse" node in the VP's Vold process and fork a sdcard process; 2) mount FUSE filesystem to the "dev/fuse" node.

ADB is a command-line utility that can debug apps, transfer files back and forth with a PC, and run shell commands. Enabling ADB for a VP facilitates app security testing [70]. ADB includes three components: a client, a server, and a daemon (adbd). Usually, the ADB server and ADB client are located in one device, and they communicate with adbd process in another device. The cross-device communication performed by ADB complicates its virtualization. If the host and the VP are running ADB command at the same time, we must virtualize the two ends of ADB protocol to avoid conflict.

We build a mutual exclusion mechanism in the Android framework layer. When switching a system to the foreground, we terminate the adbd process in the other ones. In this way, only the foreground VP can use ADB exclusively. This mechanism is simple to implement, but the side effect is that the host and background VPs' ADB do not work. We argue that this trade-off is acceptable, as the VP is always activated when using ADB. Besides, as the ADB protocol partition can only be mounted for one time, we also solve the difficulty of sharing the ADB protocol partition with the VP. In the CellD process, we intentionally mount "/dev/usb-ffs/adb", the ADB protocol partition's mount point, to the VP's system directory. As a result, the ADB protocol partition is visible to the VP.

D KERNEL VERSION CUSTOMIZATION

```
const char linux_banner[] =
    "Linux version " UTS_RELEASE " (" LINUX_COMPILE_BY "@"
    LINUX_COMPILE_HOST ") (" LINUX_COMPILER ") " UTS_VERSION "\n";
```

Listing 1: Kernel version information.

```
const char linux_proc_banner[] =
 "%s version %s"
 " (" LINUX_COMPILE_BY "@" LINUX_COMPILE_HOST ")"
 " (" LINUX_COMPILER ") %s\n";
```

Listing 2: linux_proc_banner.

Customizing kernel version information is a little bit tricky. Listing 1 shows the kernel version information. It consists of two objects defined in the UTS namespace data structure ("UTS_RELEASE" and "UTS_VERSION "), as well as linux_proc_banner information (as shown in Listing 2). The customization of linux_proc_banner information (Listing 2) is similar to other kernel-related profile customizations, but we need to take special measures for "UTS_RELEASE" and "UTS_VERSION." These two objects are bound to the UTS namespace, and the only place that we can edit them is in the function "clone_uts_ns", which creates a new UTS namespace when booting the VP. Therefore, we embed a customization function in

Table 6: VPBox's customizable device-attribute options (total number: 150).

Type	Customizable Device-Attribute Options	Number
System Property	SECURITY_PATCH, RESOURCES_SDK_INT, BASE_OS, Gsm.version.baseband, Product Name, Useragent, PREVIEW_SDK_INT, CODE NAME, Description, Secure, USER, Brand, Specific Version Number, Hardware Serial Number, Device Fingerprint, Device Version Number, Product Local region, Device Model, Device TAGs, Manufacturer, Device Version Type, Version ID, Product Device, Httpagent, Device Bootloader, Product Board, Product Locale Language, User Key, RADIO, Compile machine name, Compiler, SDK, SDK_INT, Version increment, Compile time, Compile type	36
Kernel Version	UTS_RELEASE, UTS_VERSION, LINUX_COMPILE_BY, UTS_VERSION, LINUX_COMPILE_HOST, UTS_MACHINE	6
Memory	Heapsize, Heapgrowthlimit, AvailROMSize, TotalROMSize, AvailRAMSize, TotalRAMSize	6
CPU	CPUFreq, CPUHardware, CPU Cores, CPU Model, CPU Hardware, CPU Architecture, CPU Version, CPUTemp, CPUABI, CPU Variant, CPU Part, Feature, CPU Serial Number, CPU Vendor	14
Network	MAC address, SSID, BSSID, RSSI, IP Address, DNS1, DNS2, Gateway, Available Networks, NetRate, Netmask, WiFiState, NetworkInerfaces, TypeName, NetworkId, NetworkType, Network Capabilities, Throttling	18
Power	Battery Scale, Battery Plugged, Battery Temperature, Battery health, Battery Voltage, Battery Level, Battery Status, Battery Technology, Battery Type	9
Bluetooth	Bluetooth Name, Bluetooth MAC Address, Connected Devices, ProfileConnectionState, Available Devices, Bluetooth Scanmode, Bluetooth Version, Bluetooth State, Bonding State, Device Alias, Profiles (e.g., Contact Sharing), Rssi, ScanResultType, ManufacturerData	14
Location	Accuracy, Speed, GPS Status, Location Type, Best Providers, Base Station Signal Strength, NetworkId, Longitude, Latitude, Bearing, Altitude, Location Area Code, Cell Identity, SystemId, BaseStationId	15
Telephony	SubId, ImsRegistrationState, MMS_USER_ANENT, MMS_UA_PROF_URL, Mobile Network Cod, IMEI1, IMEI2, MEID, IMSI, IMEISV, ESN, ICCID, Phone Number, SIMState, SIMCountryIso, Carrier_name, Mobile Country Code, SIMOperatorName, Phone Type, SIMOperator	20
Display & GPU	GPU Version, Vendor, Density, Renderer, Resolution, ScaledDensity, Extensions, Touch Screen Type, Brightness, x_px/y_px, x_dpi/y_dpi, GPU Extension	12

Table 7: G-Ware samples' MD5 values.

Sample	MD5
G-Ware1	B7494A6879FD107FC0910D9F6B7F49B2
G-Ware2	AE2437BC6B21D83A9262A752CD56E678
G-Ware3	BB878E32E75D1136CC10D89619C64E37
G-Ware4	6F46F37EFACE7E6ED38306DA9536A9E5
G-Ware5	5B6614A0E3A824DE836B5D86919F37DA
G-Ware6	8FDFD410B35B356EE2D67828A6A2F05C
G-Ware7	5F62A64CCA5E5CA87C36D3FC6D2FC986
G-Ware8	F9265AA20E6D53C680B9A76E4CFC9F28
G-Ware9	1F66A7A83A331C4DA8FF9EB55C7B317C

"clone_uts_ns" to 1) access our customized "UTS_RELEASE" and "UTS_VERSION" via our created syscall; 2) update the data structure "new_utsname" that defines these two objects.

,