VPDroid System Design

Anonymous Author(s)

1 VPDROID SYSTEM DESIGN

We develop a lightweight Android OS virtualization architecture, *VPDroid*, to assist data-clone attacks. With VPDroid, attackers can configure different device settings according to the victim phone's attributes and then boot up a virtual phone (VP) environment that closely approximates the victim's device. To deceive the cloned apps into thinking the smartphone is not changed, VPDroid has to meet two requirements (**RQ1** & **RQ2**):

(1) **RQ1**: the VP always gets direct access to hardware devices and thus revels native performance; this design ensures VPDroid can evade the detection of Android emulators.
(2) **RQ2**: the auto-login functions of cloned apps are imperceptible to the change of device; this requires our virtualization and the editing of device-specific attributes are invisible to cloned apps.

VPDroid is built on top of Cells [1], because its foreground VP design meets **RQ1**. However, Cells fails to meet **RQ2**: it is not designed to edit device attributes, and its user-level device virtualization modifies the VP's application framework layer, which can be detected by cloned apps. Besides, Cells's kernel-level device virtualization to many hardware devices are not compatible with Android 6.0 and later versions any more. We extend Cells in many ways to achieve our requirements on mainstream Android versions.

1.1 Overview

Figure 1 provides an overview of VPDroid's device virtualization. We keep Cells's foreground VP design but remove the support to background VPs. For data-clone attacks, maintaining one customizable VP will suffice. This design choice also simplifies our virtualization implementation and device attribute customization. The isolated VP runs a stock Android userspace environment. VPDroid utilizes Linux namespaces as well as the `device` namespace introduced by Cells to transparently remap OS resource identifiers to the VP. The VP has its private namespace so that it does not interfere with the host smartphone.

We retain Cells's virtualization solutions on Input (e.g., touchscreen and input buttons), Sensors (e.g, accelerometer and light sensors), and Audio. We also keep the custom process, "CellD", in the host device's root namespace. CellD manages the starting and switching of VPs, and it also coordinates our virtualization to ADB, which is used for copying data to the VP. §1.2 ~ §1.5 present our updates and new additions to Cells's device virtualization solutions. Among them, VPDroid improves Cells in two major ways. First, we design a new user-level device virtualization solution, which has better portability and transparency than Cells (see §1.2). Second, VPDroid is able to customize the VP's device attributes, but this function is not offered by Cells (see §1.5).

1.2 New User-level Device Virtualization

For Cells's obsoleted device virtualization solutions, rewriting every kernel driver to adapt to new Android versions is error-prone and complicated. Especially, some hardware vendors provide proprietary software stacks that are completely closed source (e.g., telephone & Bluetooth). Without hardware vendor's support, it would be difficult if not impossible to virtualize them in the kernel. VPDroid's user-level virtualization offers a flexible and portable alternative without compromising transparency. In particular, our mechanism contains two methods to fit different devices.

Binder service sharing. For the system services that are registered in ServiceManager (e.g, WifiService & SurfaceFlinger), we develop a new way to virtualize them. Binder is the interprocess communication (IPC) mechanism in Android. The Binder driver is a custom pseudo driver with no corresponding physical device. We first modify the Binder-driver data structure (e.g., `context_mgr_node`, `procs`, `dead_nodes`, etc) to ensure that the VP has its own Binder-driver data structure. In addition, we create a new specific handler in Binder's data structure and make 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 service process in the VP to share the corresponding service in the host system. Furthermore, we leverage SELinux technology to enforce which services in the host system can be shared by the VP. In VPDroid, we use Binder service sharing to multiplex Wi-Fi configuration, display, and GPU.

Device namespace proxy. For the anonymous services that are not registered in ServiceManager such as telephone and Bluetooth, as the kernel does not have their `binder_node` and `binder_ref` structures, we cannot apply binder service sharing to them. Instead, we modify Cells's device namespace proxy to virtualize telephone and Bluetooth. The core of Cells's design is creating its own proxy for each VP, and this proxy in turn interacts with related hardware vendor library running in the host through CellD. However, like API hooking, Cells's proxy in the VP can be detected by cloned apps. §1.3.4 and §1.4.1 describe how we address the transparency issue by creating device namespace proxy in the host only.

1.3 VPDroid's Updates to Cells's Virtualization

As labeled as light grey boxes in Figure 1, we update Cells's implementation in following modules by modifying both kernel and user-level virtualization methods.

1.3.1 Network. Networking virtualization involves two steps to multiplex: 1) core network resource, and 2) wireless configuration management. In the first step, we reuse most of Cells's kernel-level work to virtualize core network resources such as network adapters, IP addresses, and port numbers. The only exception happens when handling multiple routing tables. Since Android 5.0, Android system has adopted so-called "policy routing" to work with multiple routing tables and rules. Policy routing defines for which traffic a specific routing table is used. We extend Cells by configuring `ndc` and `iptables` commands to add new rules for policy routing.

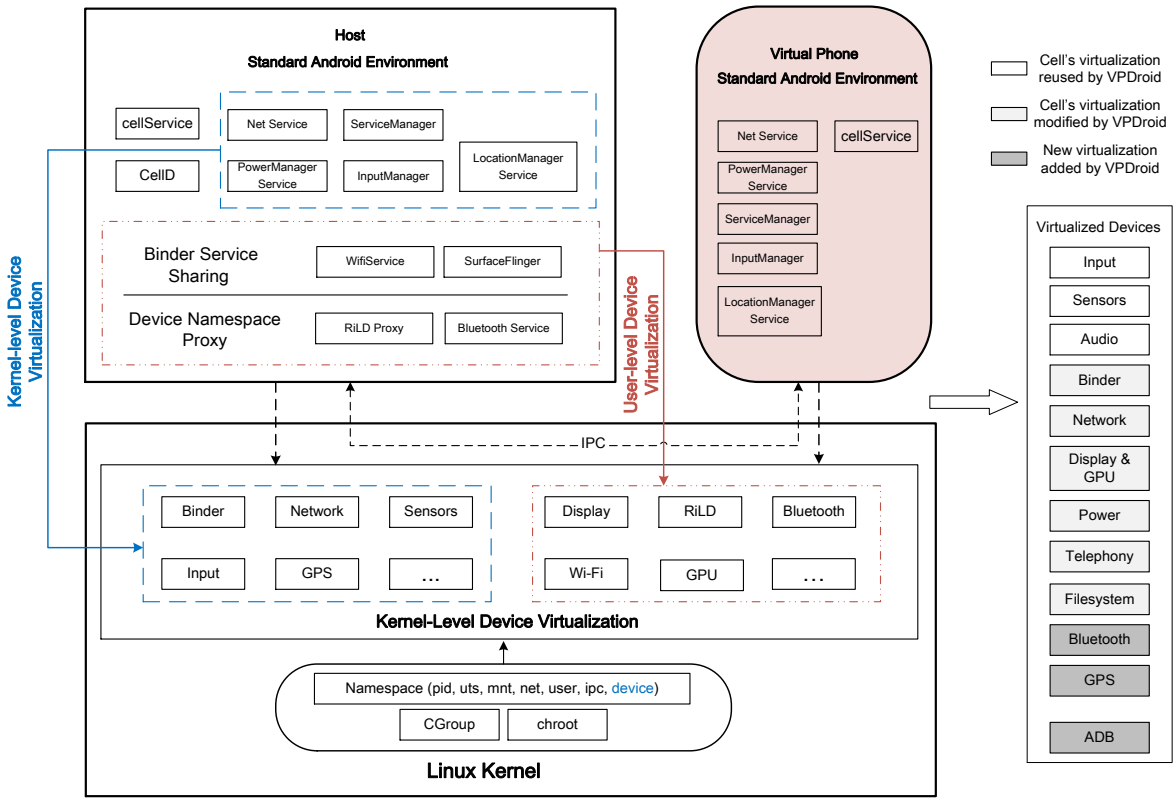

Figure 1: VPDroid's device virtualization architecture (kernel-level & user-level) and our changes to Cells.

The second step is to virtualize wireless configuration and status notifications, which occur in user space. Cells's device namespace proxy forwards all configuration requests from the foreground VP proxy to the host's "wpa_supplicant", which is a user-level library that contains wireless network service code. Cells replaces "wpa_supplicant" inside the VP with a Wi-Fi proxy. In contrast, we leverage our proposed binder service sharing to achieve the same goal, but leaving no change in the VP's userspace. We illustrate our approach in Figure 2. In Android system, WifiService calls the library of "wpa_supplicant" to detect Wi-Fi connections, and such information is sent through NetworkAgent to ConnectivityService, which answers app queries about the state of network connectivity. We use the binder service sharing mechanism to share WifiService between the VP and the host system. The blue two-way line in Figure 2 shows the workflow to answer a Wi-Fi status query from the VP's app. In addition, we create a new NetworkAgent in the host system and bind it to the VP's `device` namespace. Like sharing WifiService, we also use binder service sharing mechanism to transfer the new NetworkAgent to the VP's ConnectivityService; as shown in Figure 2's red line, the purpose of doing this is to automatically forward network status notifications to the VP.

1.3.2 Display & GPU. The display is one of the most important devices in modern smartphones, and GPU provides hardware display acceleration. Android system takes Linux framebuffer (FB) as an abstraction to a physical display and screen memory. Cells virtualizes FB by multiplexing FB device driver. However, Android 6.0 and later versions have switched to the ION driver for managing the screen memory. Instead of modifying the ION driver, we take a much simpler way via our user-level device virtualization.

W still use the binder service sharing mechanism to make the VP share 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 that is finally to be displayed. Besides, we also adapt related data structures, graphics rendering APIs, and interfaces for our needs: 1) we add system tag field in the Layer data structure to detect which system (VP or host) the Layer belongs to; 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 offers another advantage: we do not need to takes special measures for GPU virtualization. Since the VP actually multiplexes the host system's screen memory buffer, the host's GPU can directly work on it for display acceleration.

1.3.3 Power. In power management virtualization, VPDroid inherits Cells's solution in *wake locks* virtualization, and the major difference is how to manage *early suspend*. After Android 5.0, the management of display's on/off screen has been replaced by SurfaceFlinger's `setPowerMode` interface rather than using *early suspend*

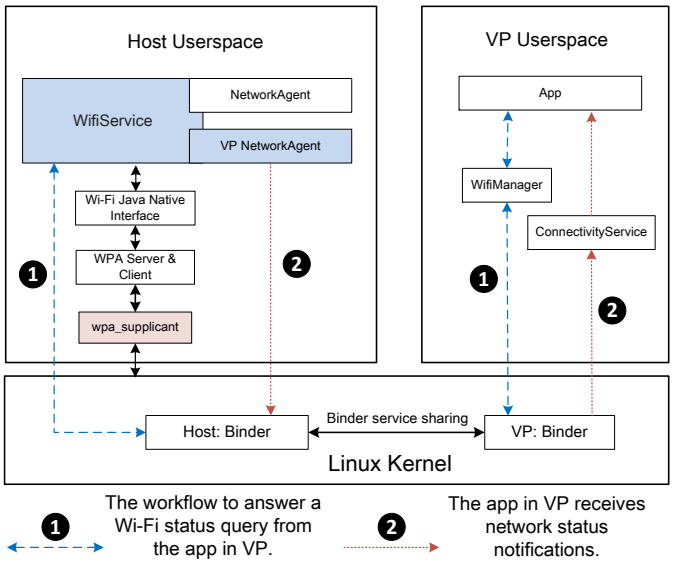

Figure 2: VPDroid leverages binder service sharing mechanism to virtualize wireless configuration management.

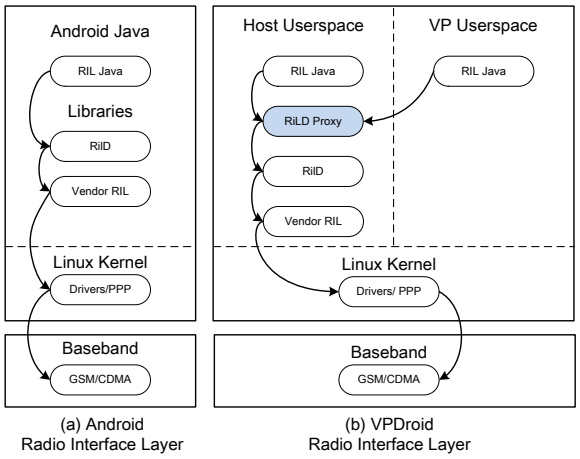

Figure 3: VPDroid's Radio Interface Layer (RIL).

subsystem. As we have already virtualized SurfaceFlinger service, we just need to prevent the background system from using the `setPowerMode` interface. In this way, the background system cannot put the foreground system into a low power mode.

1.3.4 Telephony. As smartphone vendors customize their own proprietary radio stack, Cells adopts user-level device namespace proxy to provide a separate telephony functionality for each VP. Each VP has its own proxy Radio Interface Layer (RIL) library. The RIL proxy is loaded by Radio Interface Layer Daemon (RilD) and connects to CellD running in the host's root namespace, and CellD in turn communicates the hardware vendor library to respond to the VP's requests. However, the RIL proxy in the VP, like API hooking, is not invisible to cloned apps, which does not meet our **RQ2**.

As shown in Figure 3, we implement a socket-interface based proxy scheme only in the host userspace, and it does not require the assistance of CellD. In the host's Radio Interface Layer, we create a RiLD proxy between the communication flow of Android telephony Java libraries (RIL Java) and RilD. Then we create another two standard Unix Domain sockets in the proxy. One socket connects to the RIL Java of the VP, and the other one connects to the RIL Java of the host system. The RIL Java in the 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. In turn, the RilD proxy passes the VP-related arguments (e.g., call ring and signal strength) to the VP's RIL Java over a socket.

1.3.5 Filesystem. Cells's SD card partition virtualization does not comply with the new SD card access management starting from Android 4.4, in which Filesystem in Userspace (FUSE) technology kicks in to manage the SD card partition. Recent Android versions directly fork a process in 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 VPDroid runs a 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.

1.4 VPDroid's New Additions to Cells's Virtualization

Cells did not virtualize Bluetooth and GPS. However, accessing Bluetooth and GPS in the VP is indispensable to evading Android sandbox detections and device-consistency checks. Therefore, we add virtualization support to Bluetooth, GPS, as well as Android Debug Bridge (ADB). After an attacker customizes a VP environment and then boot it up, ADB helps the attacker copy the victim's private data into the VP and then launch a data-clone attack.

1.4.1 Bluetooth. Bluetooth virtualization is the most challenging implementation in VPDroid. The Bluetooth vendor library is like a black box and entirely closed source. As shown in Figure 4(a), to require Bluetooth service, other apps first call Android Bluetooth APIs (❶), which further send the request to Bluetooth service process (❷) via Binder IPC. After that, Bluetooth service process connects to Hardware Abstraction Layer (HAL) via Java Native Interface (JNI) to interact with Bluetooth stack and the hardware vendor library (❸). However, each smartphone manufacturer provides its own proprietary Bluetooth vendor library. In addition, the number of Bluetooth profiles that complete different short-range communication functions is also increasing.

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 binder service sharing mechanism to ❷. Instead, we implement a new service proxy to virtualize Bluetooth. Figure 4(b) illustrates our workflow. We modify the Bluetooth app ("packages/apps/Bluetooth") and embed a Bluetooth JNI proxy. After our modification, Bluetooth service process now only contains the JAVA module (❹), and the original JNI module is put into the newly added Bluetooth JNI proxy (❺). Therefore, now it is the Bluetooth JNI proxy to interact with HAL, Bluetooth stack, and the hardware vendor library.

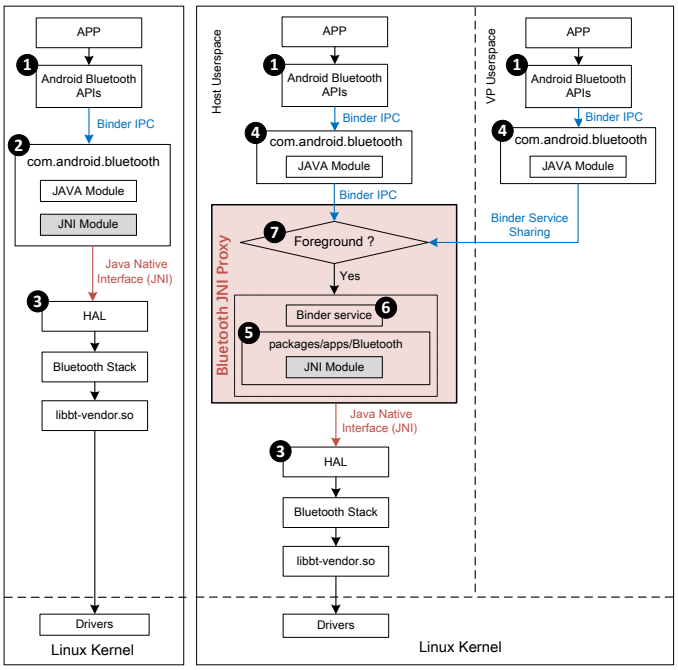

Figure 4: VPDroid's Bluetooth virtualization.

Furthermore, to enable our proxy to communicate with the new Bluetooth service process (❹) in the host or the VP, we also build a binder service in the Bluetooth JNI process (❻). In this way, the VP can finally access the Bluetooth driver in the host device. However, the Bluetooth driver does not support multiplexing, and an exception will happen if both the host and the VP establish a connection with the driver. Therefore, we add an additional namespace check in our proxy (❼): we only forward Bluetooth service requests for the foreground system. Note that the user apps in the VP have no privilege to access ❹ to detect our change unless they perform process injection or modify the related SELinux policy.

```
1 void dump_stack(void)
2 {
3   dump_stack_print_info(KERN_DEFAULT);
4   show_stack(NULL, NULL);
5 }
```

Listing 1: Dump the current task's stack trace.

The app running in the VP may find the signs of virtualization from a full stack trace. We take Bluetooth virtualization (Figure 4(b)) as an example. When an app in the VP initializes a Bluetooth request, it can print a stack trace and find "/system/lib64/libbinder.so" in the call chain—we use Binder service sharing to connect the VP's Bluetooth service process with our Bluetooth JNI proxy in the host. To hide virtualization footprints from a full stack trace, we customize Linux kernel function "dump_stack" (Listing 1): when it receives a stack trace dump request from the VP, it will return a normal stack trace just like the one dumped from a physical phone.

1.4.2 GPS. GPS relies on a physical chip for location tracking. `GpsLocationProvider` is the GPS provider in the Android framework layer. It calls HAL interface via JNI methods. The HAL interface interacts with GPS chip through "/dev/gss" driver. GPS chip is an active tracking device. This means after a user's first request, GPS chip will continue to report the location information to `GpsLocationProvider` without interruption. However, GPS chip only supports one connection.

Our virtualization of GPS is to rewrite '/dev/gss" driver. We modify "gss_open" and "gss_event_output" functions to support: 1) multiple connections; and 2) the location information received from the chip is forwarded to multiple clients at the same time. The location information goes through HAL and eventually reaches `GpsLocationProvider` in the Android framework layer of the host and the VP, respectively.

1.4.3 ADB. ADB is a command-line utility that can debug apps, transfer files back and forth with a PC, and run shell commands. ADB includes three components: a client, a server, and a daemon (adbd). Usually, the ADB server and ADB client in one device 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 one. In this way, only the foreground system can use ADB exclusively. This mechanism is simple to implement, but the side effect is the host system's ADB does not work when it is displayed in the background. We argue that this tradeoff does not affect us, as the VP is always activated during virtualization-assisted data-clone attacks. Besides, as the ADB protocol partition can be only mounted for one time, we also solve the difficulty of sharing the ADB protocol partition with the VP. In 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.

1.5 Virtual Phone Customization

The virtualization solutions from §1.2 to §1.4 attempt to provide VPDroid users the same experiences as using a physical smartphone. Our design ensures that Android emulator detection heuristics are in vain for VPDroid. Now we present how we achieve our **RQ2**—evading device-consistency checks, so that cloned apps are unaware of the change of device even they have the root privilege. Figure 5 shows the workflow of customizing the VP's device attributes. VPDroid users provide a configuration file "build.VPDroid.prop" in advance, 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. Besides, we incorporate multiple namespaces to isolate our customization.

1.5.1 Android System Properties. Android system properties are const values that describe the configuration information of the mobile device, including brand, model, serial number, IMEI, android_id,

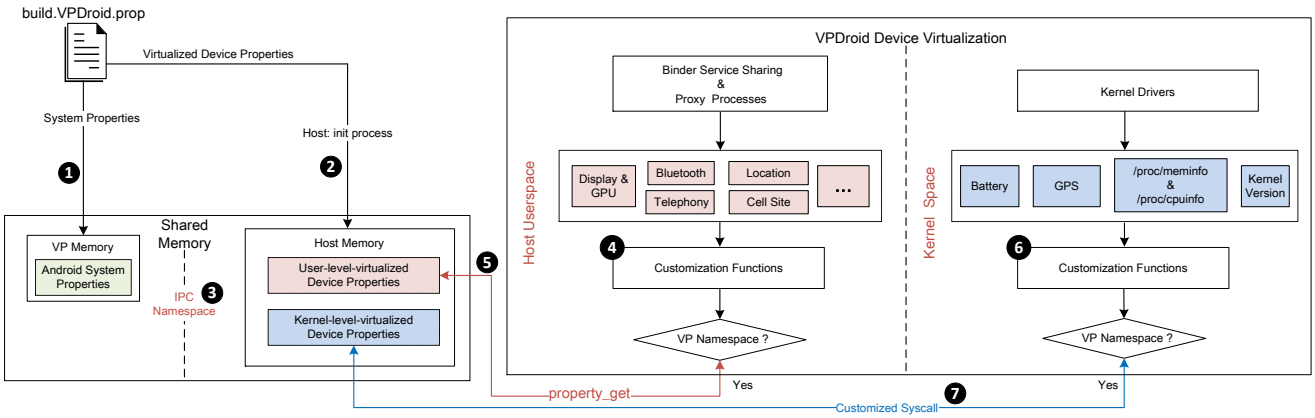

Figure 5: VPDroid users provide device-specific attributes in "build.VPDroid.prop" configuration file. The VP's customization happens either in the kernel drivers or the host's userspace. We incorporate multiple namespaces to isolate our customization.

etc. They are stored in the init process's shared memory but not related to 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 run time by calling "property_get", an API for native code to read the data in the shared memory space from other processes. Therefore, during the process of booting up the VP, its init process will call "load_system_props" to load the customized Android system properties in "build.VPDroid.prop" into the VP's shared memory space (❶). Then the customized system properties are ready for other apps running in the VP to access and inquire.

1.5.2 User-level Customization. Recall that we virtualize some devices such as Bluetooth, Wi-Fi, and telephony at the host userspace. The second category of "build.VPDroid.prop" just contains the device attributes that we customize for these user-level-virtualized devices. The customized data in the second category will be loaded into the host init process's shared memory (❷). We enforce the IPC namespace for the host and VP shared memory isolation (❸). We embed customization functions (❹) in the places where we perform user-level device virtualization such as Bluetooth JNI proxy and RiLD proxy. The customization function first determines whether the current query request is from the VP by checking the associated `device` namespace. If yes, it calls "property_get" to get the customized data from the shared memory that maps "build.VPDroid.prop" and then returns the fake data to the VP (❺).

1.5.3 Kernel-level Customization. The third category contains key-value pairs that are used to customize for kernel-level-virtualized devices, such as power and GPS. In addition, certain kernel drivers contain device basic 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.

We embed customization functions (❻) in these kernel drivers to interact with the shared memory of the host's init process. However, the customized data loaded into 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 (❼). All of our customization functions in the kernel drivers work in a similar style. For example, we customize the battery-related profiles (e.g, battery level, temperature, and voltage) 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, it will call our created syscall to extract customized data. Due to the isolation of `device` namespace, the cloned app in the VP is by no means to bypass our customization.

Customizing kernel version information is a little bit tricky. Listing 2 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 3). The customization of linux_proc_banner information (Listing 3) 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 "clone_uts_ns" to: 1) access our customized "UTS_RELEASE" and "UTS_VERSION" via our created syscall; 2) update the data structure "new_utsname" which defines these two objects.

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

Listing 2: Kernel version information.

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

Listing 3: linux_proc_banner.

1.5.4 The Advantages of VPDroid Customization. Compared with existing Android device-attribute editing tools, our customization solution revels distinct advantages. All of our editings do not rely on any user-level hooking mechanism and happen out of the VP's runtime environment, which means our device customization is

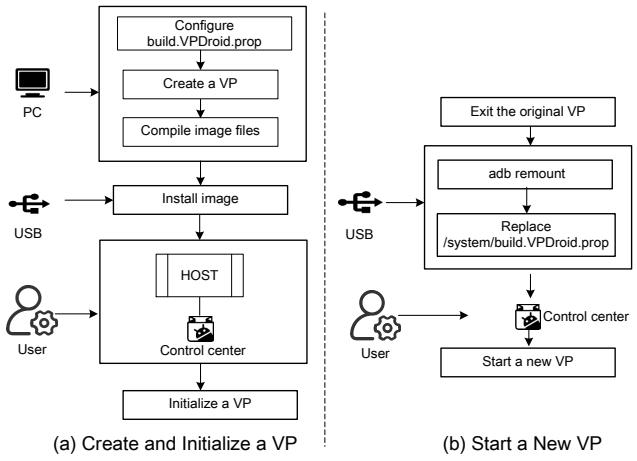

Figure 6: The workflow of creating and starting a VP.

invisible to cloned apps. Although our user-level device virtualization allows the VP's process to share certain services in the host system, with the `device` namespace isolation, the app running in the VP is still unaware of any device-specific differences even it has the root privilege, and we will confirm this in our evaluation.

VPDroid now provides 101 device configuration options, which span a wide spectrum of device attributes. We collect them from the existing Android sandbox detection work and our reverse engineering of the apps that perform device-consistency checks. To the best of our knowledge, VPDroid offers the largest and most comprehensive Android device-attribute editing options so far.

2 VPDROID IMPLEMENTATION

VPDroid's prototype contains 11, 674 new lines of C/JAVA code to Cells's codebase.[1] 10% of lines of code are used to update kernel drivers; 83% of lines of code work at the host phone's native C/C++ libraries, and the left are Java code working at the host phone's application framework layer. Currently, VPDroid is compatible with Android 6.0 to 10.0. As shown in Figure 6(a), 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 VPDroid users to switch between the VP and the host system swiftly. Figure 6(b) shows how to start a new VP to simulate a different device: 1) exit the original VP; 2) update and replace a new "build.VPDroid.prop" configuration file; 3) stat a new VP via the control center app.

REFERENCES

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

[1] Cells's lines of code is 8, 028.