A Lightweight Virtualization Solution for Android Devices

Wenzhi Chen, *Member, IEEE*, Lei Xu, *Student Member, IEEE*, Guoxi Li, *Student Member, IEEE*, and Yang Xiang, *Senior Member, IEEE*

Abstract—Mobile virtualization has emerged fairly recently and is considered a valuable way to mitigate security risks on Android devices. However, major challenges in mobile virtualization include runtime, hardware, resource overhead, and compatibility. In this paper, we propose a lightweight Android virtualization solution named *Condroid*, which is based on container technology. *Condroid* utilizes resource isolation based on namespaces feature and resource control based on cgroups feature. By leveraging them, *Condroid* can host multiple independent Android virtual machines on a single kernel to support mutilple Android containers. Furthermore, our implementation presents both a system service sharing mechanism to reduce memory utilization and a filesystem sharing mechanism to reduce storage usage. The evaluation results on Google Nexus 5 demonstrate that *Condroid* is feasible in terms of runtime, hardware resource overhead, and compatibility. Therefore, we find that *Condroid* has a higher performance than other virtualization solutions.

Index Terms—Container, virtualization, android, security

1 Introduction

1.1 Motivation

Smart mobile devices have already been an omnipresent part of our daily lives. By the end of 2013, Android claimed 61.9 percent market share of all smart devices and nearly 79 percent market share of smartphones [1]. In the second quarter of 2014 Android continued to dominate the global smartphone market with nearly 85 percent of the market share [2].

As smart devices become increasingly common, they have also become an active area of research. In recent years, the interest of the research community has focused on two topics regarding smart mobile devices: 1) Security threats, and 2) bring your own device (BYOD).

1.1.1 Security Threats

While Android has become popular, it has also become an attractive target for malware because of its openness. The report of F-Secure states the number of malicious software on the Android platform accounts for 97 percent of the overall number of mobile malware. In 2013, the malicious deduction virus ranks in first place with 23 percent, while fraud and process control takes second and third place respectively with 21 and 16 percent [3].

Viruses, Trojan horses and malware from all kinds of external attackers have attracted attention. However, deploying a security environment (such as encryption, digital signature, safety audit, access control, and digital certification) on a mobile device is very complicated for everyday users. People need an innovative solution that can offer a secure and credible execution environment when using some critical applications (mobile payment, mobile banking), or when accessing sensitive data (SMS, contacts) [4].

Virtualization [5], [6] can offer a secure zone to store sensitive Apps, data and private information, and prevents malware from infiltrating the secure zone from other insecure environments [7], [8], [9], [10].

Moreover, virtualization can create an isolated environment [11], [12], [13], [14], [15] that can be used in a safe way to run programs that may ruin the host OS, other app or users' data or private information. If the environment does crash or become compromised, the rest of the host OS is not affected because the isolated environment created by virtual technology cannot directly access the host resources.

1.1.2 BYOD

The concept of BYOD [16] refers to the policy of permitting employees to bring personally owned mobile devices to their workplace, and access privileged company information and applications with these devices. BYOD increases employee morale and convenience by allowing employees to use their own devices and it also makes the company look a flexible and attractive employer [17], [18], [19], [20], [21]. It seems the answer to BYOD is mobile virtualization [19]. Mobile virtualization enables a single device to offer two or more personas with different system settings and user profiles and totally different operating environments [22].

Some other scenarios may include: (1) a company needs to monitor and remotely manage the devices of employees but employees do not want to be monitored or controlled when they use their devices for personal use (telephony, gaming, web browsing); (2) a company needs

- *W. Chen, L. Xu, and G. Li are with the School of Computer Science and Technology, Zhejiang University, Hangzhou 310027, P.R. China. E-mail: {chenwz, leixu, guoxili}@zju.edu.cn.*
- *Y. Xiang is with the School of Information Technology, Deakin University, Burwood, Vic. 3125, Australia. E-mail: yang@deakin.edu.au.*

Manuscript received 4 Nov. 2014; revised 29 Dec. 2014; accepted 1 Jan. 2015. Date of publication 8 Jan. 2015; date of current version 10 Sept. 2015. Recommended for acceptance by Y. Pan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2015.2389791

to backup the workspaces of employees, destroy them after work and then restore them on the next workday. Mobile virtualization can make all these possible and easy to achieve.

However, the concept of mobile virtualization is not simply applying the current virtualization technology to the current mobile devices. When mobile devices meet virtualization, some new challenges do emerge [23], [24], [25], [26], [27], [28], [29], [30], [31]:

Challenge 1: Keeping the device's native performance. Nobody wants to try new technology yet sacrifice performance. If so, they would rather buy another phone than run two phones on the one physical phone.

Challenge 2: Keeping the native user-experience. The phone should maintain the same look and feel. Users should not be aware of virtualization when it comes to notifications, smart switching, links or sharing.

Challenge 3: Supporting more than two personas. This is the most complex challenge. The person who owns the phone is not limited to two personas. He may have different personas for his children, parents, or colleagues etc.

1.2 Our Contributions

This paper aims to develop mobile virtualization architecture for Android, called *Condroid*. *Condroid* will enable a single device to run several virtual Android phones in a simultaneous, independent, isolated and secure manner. Our architecture can also meet the recent security and BYOD requirements and challenges mentioned above (our project is also open source on Github at http://condroid.github.io).

Considering the power and performance limits of common mobile devices, our architecture has to run in a stable and endurable manner. The weight of virtualization is mainly responsible for power and performance. This explains why we need to employ some kind of lightweight virtualization solution for our architecture.

Our contributions are summarized as follows:

- *Porting Linux container (LXC)* [32] *to Android*—We port the Linux Container tools to Android while fixing problems with compatibility. This equips Android OS with its own lightweight virtualization capacity.
- *Full-featured container virtualization architecture*—We design efficient container virtualization architecture with several device virtualization models, such as Binder, Display and Input. This allows a single set of device resources to be shared among multiple Android environments.
- *Efficient service sharing mechanism*—We present a service sharing and filesystem sharing mechanism to reduce memory and storage utilization. This significantly improves system performance.

The rest of this paper is organized as follows. Section 2 describes related work. In Section 3, we describe the architecture of *Condroid*, and Section 4 details the implementation of each subsystem. Finally, in Section 5, we use a series of benchmarks to evaluate the performance of *Condroid*. A summary and plan of our future work are given in Section 6.

2 Related Work

Isolation mechanisms that enhance security for Android can be classified into three types: ARM-based system virtualization, user-level isolation and OS-level isolation.

2.1 ARM-Based System Virtualization

The first approach to isolate runtime environments is system virtualization. This technology was originally developed for servers and desktops. We discovered some research transplanted typical x86 system virtualization platforms to an ARM platform:

KVM/ARM [33] is the first full system ARM virtualization solution that can run unmodified guest operating systems on ARM multicore hardware. KVM/ARM leverages existing Linux hardware support and functionality to simplify hypervisor development and maintainability while utilizing recent ARM hardware virtualization extensions to run virtual machines. However, KVM was not originally designed for ARM architecture, and this solution is neither mature nor stable. There is a lengthy process to modify KVM to be adaptable to ARM hardware. EmbeddedXEN [34] is a para-virtualization hypervisor specifically for ARM embedded systems. In particular, EmbeddedXEN supports heterogeneous ARM cores and keeps execution overhead as low as possible. However, solutions based on para-virtualization are not fit for mobile devices. It has a complex configuration that is not easy for common users and the guest OS code needs to be modified, which means it can't support the latest OS nor the commercially closed-source operating systems. The OKL4 microvisor [35] is designed to serve as a hypervisor as well as a replacement for the microkernel. OKL4 is a third generation microkernel of L4 heritage for the large-scale commercial deployment of mobile virtualization platforms. However, the microvisor has to work with device support and emulation, which is an onerous requirement for mobile devices that contain increasingly diverse hardware devices.

2.2 User-Level Isolation

Isolation based on user-level, known as *sandbox*, is a traditional way to confine malware. This solution uses different user identifiers per application group to implement sandboxing. The communication between applications and core Android components is restricted and based on permissions requested during the installation of applications. Drawbridge [36] is such a system designed for Windows. There is no such a library for Android currently and there are several significant challenges due to the vast architectural differences between Windows and Android. These differences include: (1) Remote Desktop Protocol (RDP) does not support virtualized applications to render graphics in Android; (2) It does not support a shared state between applications in different sandboxes; (3) It also doesn't support multiple processes bound to a single OS library.

2.3 OS-Level Isolation

Isolation based on OS-level virtualization [9], [37], as used in our solution, is a common concept for containers today. *Cells* introduced in [38] is also an Android container solution. *Cells* is virtualization architecture that enables multiple virtual smartphones to run simultaneously on the same physical cellphone in an isolated, secure manner. *Cells* introduces a usage model of one foreground virtual phone and multiple background virtual phones. In contrast to *Cells*, our approach expends effort to virtualize the Binder subsystem in Android to gain a higher performance, which is a primary Android-specific IPC framework used ubiquitously by all Android processes. In addition, *Cells* makes the most of modifications in the Linux kernel layer, and these are unlikely to merge into the mainline because this feature is not the emphasis of a standard kernel, and therefore may not capture the attention of the kernel. Most of our modifications are in the Android framework layer, and these are likely to be collected in the Android Open Source Project (AOSP) if Google wishes Android to support virtualization.

3 System Architecture

The main approach adapted for *Condroid* is the Linux kernel technology of Containers. Containers are illusions of controlling system resources so lightweight virtualization can isolate processes and resources without the complexities of full virtualization.

As mentioned above, several related technologies have been developed which range from *sandbox*, to hypervisor and container based separation. After analyzing the specialty of mobile devices and evaluating the performance of these solutions in our previous work [4], we designed container-based architecture.

Condroid uses a single OS kernel across all containers that virtualizes identifiers and hardware resources. This means *Condroid* does not require running multiple complete Android instances, rather it provides virtual environments in which multiple containers can run on a single Linux kernel. *Condroid* ensures the containers are individual, completely independent, and secure from one another in order to prevent bugs or malicious applications running in one container and adversely impacting the operation of other containers. This is done by leveraging namespaces [39] and cgroups [40].

The purpose of each namespace (currently a Linux kernel implements six different types of namespaces: Mount, UTS, IPC, PID, Network, and User) is to wrap a particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. We use the cgroups (control groups) feature to limit, account and isolate the resource usage of process groups. Cgroups provides a mechanism to partition sets of tasks and all future children into hierarchical groups with specialized behavior. We also transplant the LXC toolkit to the Android platform. To do this, we made several modifications: replace several functions the Bionic library does not support, replace some syscalls because of the difference in yaffs2, cross-compile the Android NDK toolchain, and recompile kernels to enable cgroups and namespaces to support kernel configurations. However, basic OS virtualization is insufficient to run a complete Android user space environment. Virtualization mechanisms have primarily been used in headless server environments with relatively few devices, such as networking and storage, which can already be virtualized

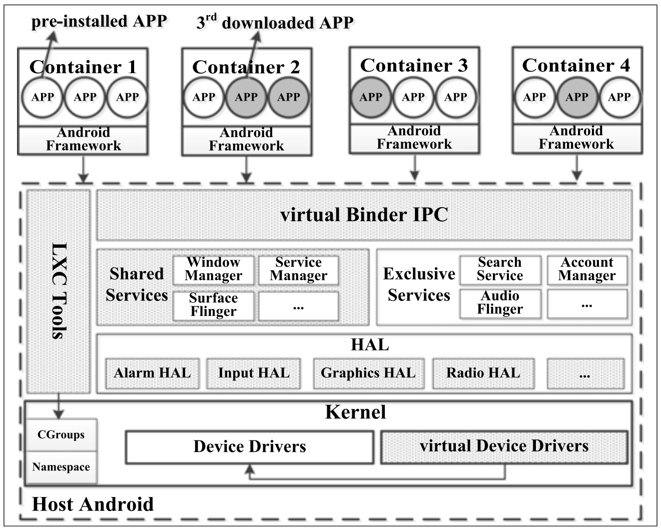


Fig. 1. Overview of *Condroid* architecture.

in commodity OS platforms such as Linux. Android applications, however, are expected to interact with a variety of hardware devices, many of which were not originally designed to be multiplexed. Therefore, mobile virtualization is not existent. In Android, certain devices must be fully supported, including both hardware devices and pseudo devices unique to the Android environment.

Condroid is a solution adapted for Android devices. It does so by integrating both the kernel-level and user-level device virtualization methods to present a complete virtual Android OS environment. Fig. 1 shows the relationship between the host and containers. The host is a control center and never installs downloaded apps despite being a complete Android OS. This design can ensure the safety of the host the most when all apps run in containers. A container may be associated with one or more apps, downloaded apps or pre-installed apps. We assume a container is secure when the container only includes pre-installed apps and trustful apps. *Condroid* can efficiently offer several secure and insecure containers.

Condroid's design provides two novel mechanisms to improve performance and user experience: (1) a system service sharing mechanism is used to reduce memory utilization. Acquiescently, multiple Android systems run multiple system services. However, some of these system services are duplicated. We offer a user-configurable way to determine which services can be shared among all the Android instances through an interface in a `/proc` filesystem; (2) a read-only filesystem sharing mechanism is proposed to reduce storage usage. Normally, people are concerned about storage usage while multiple whole Android instances exist on a single device. This mechanism means the `/system` partition of an Android system is shared among all containers.

3.1 General Description

Fig. 1 shows the basic *Condroid* architecture. The design of our prototype system is inspired by the common Linux Container architecture.The modifications we make are shown as grey in the figure. This design can offer a better user experience because it can enable a user to create, start, shutdown, and manage all the containers in the device and enable them to switch containers more conveniently. Most modifications are located in the Android framework layer and all of these will be packaged as ROM firmware.

Firstly, we think it is necessary to briefly describe the purpose of each component in Fig. 1:

- *Host android*—Our implementation needs a complete Android system as the host platform to initialize and handle *Condroid*. We treat it as a virtual machine monitor with no untrusted application ever executed in this domain. Each complete Android OS consists of a Linux kernel and Android framework.
- *Linux kernel*—Our solution needs only a single Linux kernel even though we have to run multiple Android containers. We reconfigure this kernel to enable the cgroups and namespaces feature, and we also need to make some other modifications in the kernel, such as creating some virtual devices, and writing several virtual device drivers, etc.
- *Android framework*—This is one maintained by Google that consists of system services, libraries, Dalvik runtime and some other application frameworks. We have to make some modifications in the host's Android framework to cooperate with the containers and also make a few modifications in the container's Android framework.
- *LXC tools*—LXC is a userspace interface for the Linux kernel containment features. Through a powerful API and some simple tools, it allows Linux users to easily create and manage containers. However, LXC was originally designed for Linux and what we have to do is transplant LXC to an Android runtime environment.
- *Container*—This is a virtual machine or a virtual phone that runs an isolated Android system. The Android version in different containers can be different and the communication between different containers is handled by the kernel. Each container includes several pre-installed APPs developed by Google and it can also run the downloaded APPs.

In Fig. 1, we can see the LXC toolkit is the console of *Condroid* which is a user interface to manage containers. LXC combines cgroups and namespaces support to provide an isolated environment for applications. To make it possible to use LXC in an Android system, we have to make a number of modifications. These modifications include: (1) replacing several functions that Bionic library doesn't support (such as `setenv()`, `tmpfile()`, etc.); (2) replacing some syscalls because of the difference in Android (such as `pivot_root`, `umount_oldrootfs`, etc.); (3) cross compiling using the Android NDK toolchain; (4) re-configuring the Linux kernel with cgroups and the namespaces feature so it is enabled.

In Fig. 1, we can also see there are other modifications in both the user and kernel space (shown in grey). We designed a virtual Binder IPC mechanism, which is the main communication channel between apps, even across container boundaries. We designed a service sharing mechanism by making use of the Linux *proc* filesystem interface. We also designed a device virtualization mechanism by creating several virtual device drivers and modifying the Hardware Abstract Layer (HAL).

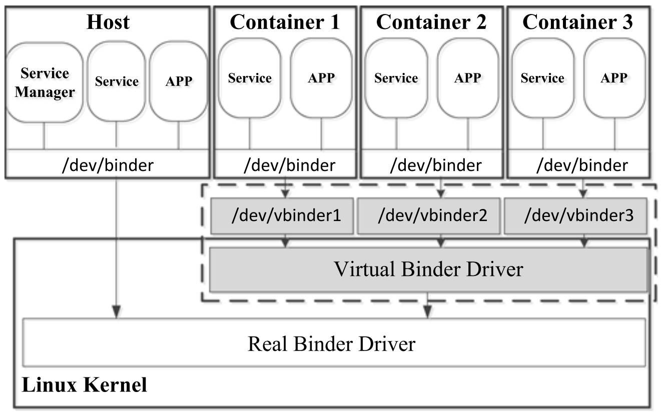


Fig. 2. Binder virtualization architecture.

4 Implementation Details

In this section, we concentrate on the details of *Condroid*. Because there are too many modifications, optimizations and adaptations in this system, we are unable to cover all of them in this paper. We will describe the implementation of the main components in *Condroid*. These include the Binder System, Display System, Input System, Service Sharing and Filesystem Sharing.

4.1 Binder System Virtualization

Binder is a system for Inter-Process Communication (IPC) used in the Android operating system. Binder is a primary subsystem used ubiquitously by all Android processes, which is why we should virtualize Binder first of all.

In order to provide a fundamental and convenient mechanism for the other subsystems, we need to find a way to share the single Binder framework (a single Linux kernel can only support one Binder framework) between the host and all containers. In Android, the Binder driver is a bridge among the ServiceManager, Service and Apps. They transfer requests and responses by using syscalls on `/dev/binder`, such as `open`, `ioctl` and `mmap`. Binder system virtualization means the host provides the main IPC components (Binder driver, ServiceManager), and that containers don't need to have these. Apps in containers communicate with the host's Binder through a virtual Binder device.

As shown in Fig. 2, we add a virtual Binder driver in the Linux kernel. The functions of this virtual driver include: (1) forwarding the operation that Apps make the on virtual binder device to the real Binder driver; (2) if the operation is `ioctl` and the target is ServiceManager (e.g. registering service or requesting service), the virtual driver will modify the name of a service using a hash function. Function (1) mentioned above makes it possible for the virtual binder to respond to all requests Apps send. Function (2) solves name conflict problems by modifying the name of the services registered in the ServiceManager. This ensures the same services running in different containers can be labelled with different names. In turn, this means the virtual driver can deliver the requests from Apps in each container to the services in the corresponding container.

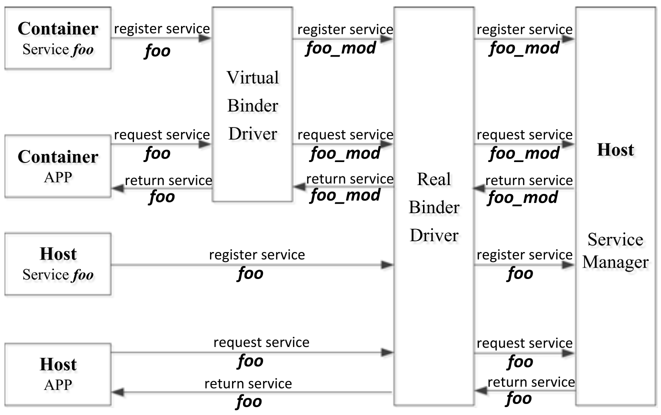


Fig. 3. The workflow of *Condroid* with binder virtualization.

After creating the virtual driver, we use this to register a set of virtual Binder devices in the kernel initialization process. The kernel will then automatically create a set of corresponding device files (`/dev/vbinder1,/dev/vbinder2...`). As shown in Fig. 2, we bind the Binder device file (`/dev/binder`) in a container to one of the virtual device files in the host before launching the containers. This means accessing the `/dev/binder` in its own root filesystem means to access the virtual binder device equivalently. All operations will then be redirected to the real binder driver. The real binder driver will think all operations are from the common processes running in the host without feeling the existence of the containers.

In Fig. 3, we described the workflow of *Condroid*. We assume the host and container have the same service *foo*. Firstly, the service *foo* needs to be registered in the ServiceManager which only runs in Android as a host. While the service *foo* in the container registers, the virtual Binder driver intercepts the registering operation and modifies the name of *foo* using a hash function. The registering operation from the host will not be intercepted. Then, while an app in the container requests service *foo*, the virtual driver will also modify the name of *foo* using the hash function and search for it in the ServiceManager. The ServiceManager then returns an object reference of *foo_mod*. As described above, we can see this mechanism solve the name conflict problem through the Binder subsystem in the Android virtualization environment.

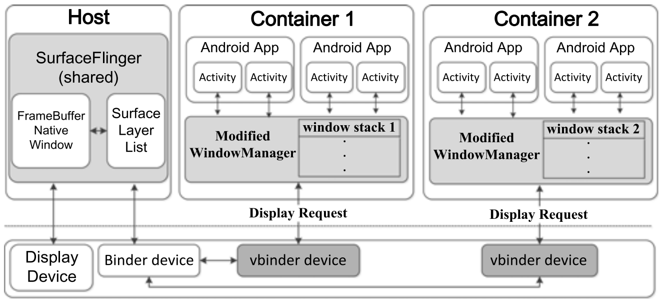


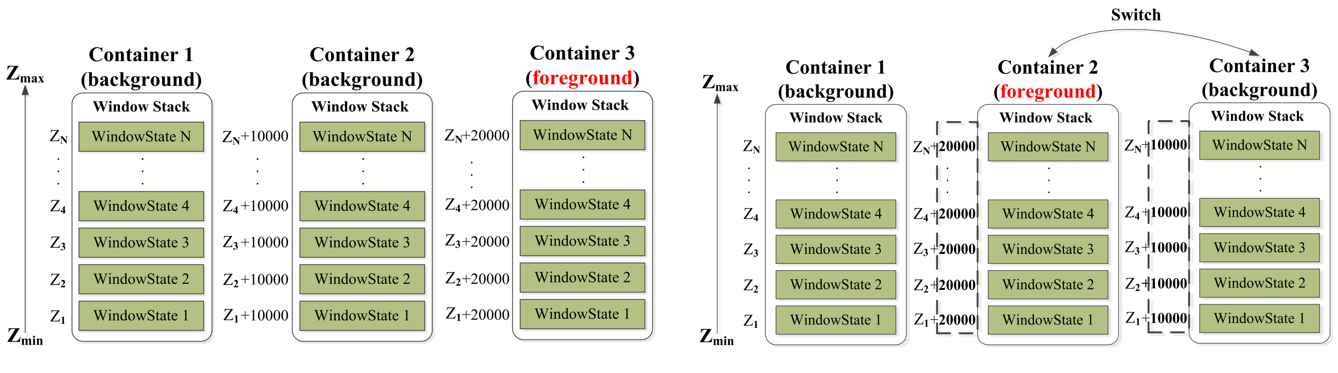
Fig. 4. The architecture of display system virtualization.

4.2 Display System Virtualization

In order to share the unique screen among all containers, we should find a way to virtualize the display system. Unlike *Cells*, we make these modifications in the Android framework rather than virtualizing the framebuffer device in the Linux kernel. It can greatly reduce memory usage without maintaining the virtual hardware state and renders any output to a virtual screen memory buffer in RAM. It is also very hard to debug when you make modifications in the kernel. Importantly, our solution has more flexibility and portability than *Cells* as our solution does not need to create any new virtual devices (*Cells* need to create a `mux_fb` device).

All modifications we made are in the WindowManager, which is Android's system service that controls window lifecycles, input events, screen orientation, position, z-order, and many other aspects of a window. The WindowManager sends all window metadata to the SurfaceFlinger, which is Android's system service that composites the visible surface onto the display. Fig. 4 shows the architecture of display system virtualization.

In Android, the WindowManager maintains a window stack which is very important for SurfaceFlinger to decide which windows to be drawn on the screen. Each item in the window stack is a WindowState object, and WindowManager calculates a Z-index for each item through a mapping function. SurfaceFlinger chooses the max Z-index value of the WindowState to draw on the screen. Here, we should modify the mapping function of the WindowManger in each container. The Z-index value of the N_{th} container should add (N-1)*10,000 to avoid repetition of the Z-index value as shown in Fig. 5a. Therefore, the container which starts last



(a) The mapping relation when 3 containers run. (b) The mapping relation after switching.

Fig. 5. The mapping function of the modified WindowManager in *Condroid*.

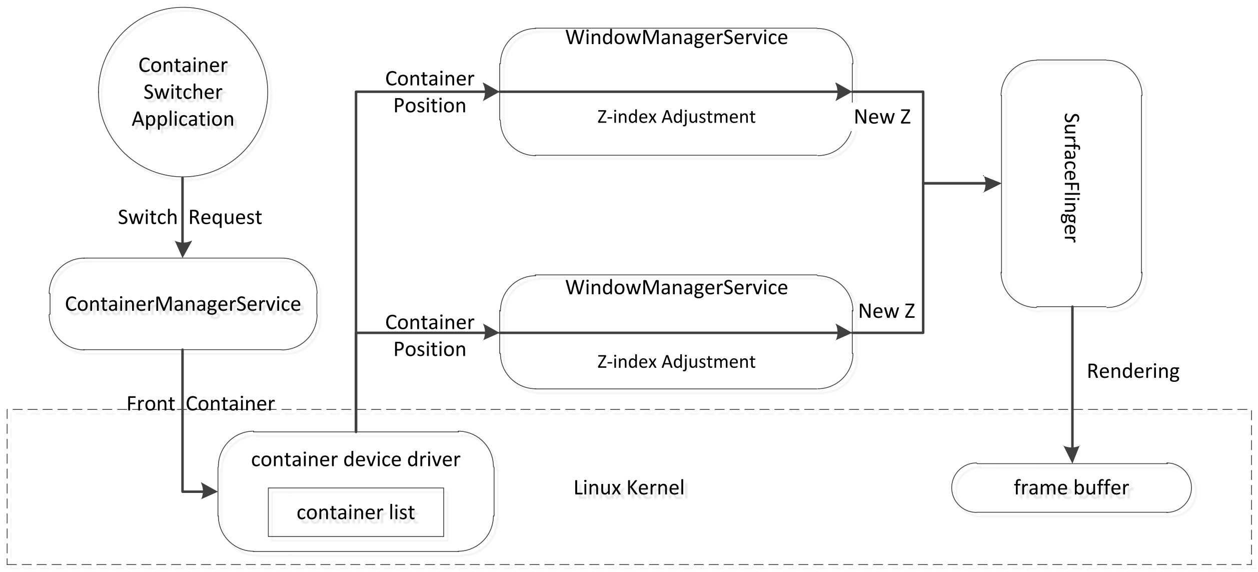


Fig. 6. The Container Switch in the *Condroid.*

has the max Z-index value and SurfaceFlinger would draw its windows on the screen. This container is now in the foreground and the others are in the background.

4.2.1 Container Switch

In order to switch between containers, we add a kernel virtual device and a framework system service. The flow chart of the container switch is shown in Fig. 6.

- *The virtual device /dev/container*—We implement a new virtual device in the Linux kernel, `/dev/container`, which provides the services of container register, container switch and some get-and-set utilities. The virtual device can also be manipulated through the system call *ioctl*. It maintains a vector that stores all containers currently running and the positon of the vector represents the order of containers booting up. We name the container in one end of the vector the front container, which is on top of the window stack, and the applications belonging to it can be seen through the screen. Input events from users can also be received.
- *The framework system service ContainerManagerService*—In the application framework of the Android OS, we add another system service, ContainerManagerService. This service maintains the adjustment value of the Z-index of the current Container it belongs to. Its client, the ContainerManager in the original system service WindowManagerService, communicates with it for managing the containers.

However, while we may want to switch container 2 to the foreground, we need to put all of its windows to the top of the window stack. This has been done by swapping the Z-index value of WindowStates of container 2 with the one which is now the foreground container. For example, swap container 2 with container 3 as in Fig. 5b.

For the user of the Android OS, one can complete the container switch through an authenticated application as shown in Fig. 6. The application sends a Switch Request to the ContainerManagerService and changes the front container to another container through the virtual device in the Linux kernel. This makes the two containers' WindowManagerService change the Z-order index of the WindowStates. Eventually, SurfaceFlinger, the shared system service, renders the new surface to the frame buffer and the container switch is completed.

Because of one shared SurfaceFlinger (details in Section 4.4) which maintains Z-index value of windows from all Containers, the extra communication of new Z values among SurfaceFlingers from different Container is eliminated.

4.3 Input System Virtualization

In Android, input events are all handled by the InputManager, which is shown in Fig. 7. The input system virtualization is done by modifying the InputManager to let the current foreground container respond to input events, which background containers will ignore. Originally, there are two member variables in the InputManager: *mInputDispatcher* and *mInputReader*. These point to an InputDispatcher object

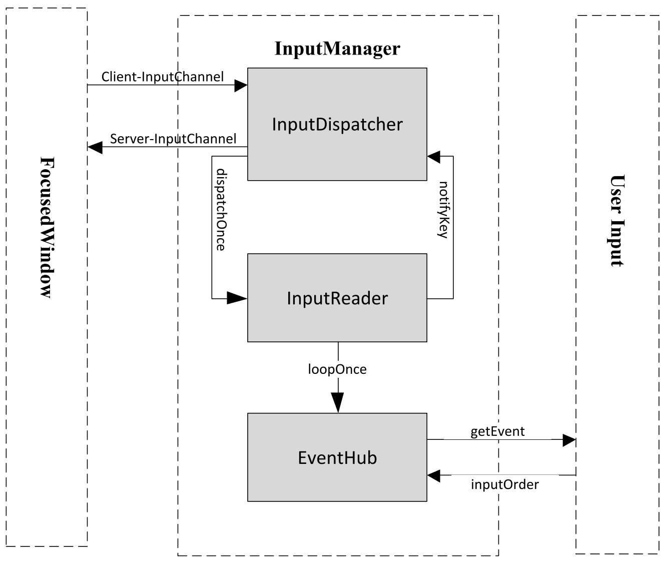


Fig. 7. The input subsystem virtualization in *Condroid.*

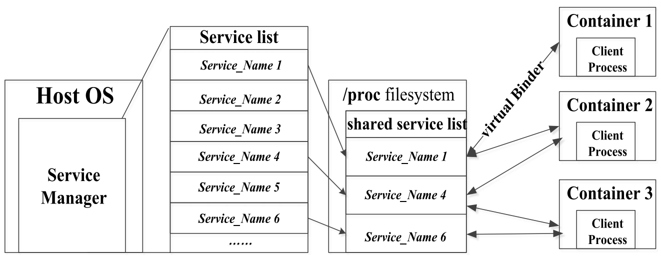


Fig. 8. The service sharing mechanism in *Condroid*.

and an InputReader object respectively. The InputDispatcher object is responsible for dispatching input events to the current activated windows and the InputReader object is in charge of monitoring input events. They run in separate threads. Acquiescently, the InputDispatcher thread will continually call its member function *dispatchOnce* to check whether the InputReader dispatches input events. If not, the InputDispatcher thread will go to sleep until roused by the InputReader. The InputReader thread will continually call its member function *loopOnce* to check whether a user issues an input order.

Under implementation, the *loopOnce* function invokes getEvent, the EvenHub's method whose responsibility is to read input events from the kernel by opening the *evdev* driver associated with each input device.

If the InputReader receives input events, it will call the function *notifyKey* of the InputDispatcher to wake up the InputDispatcher. The InputDispatcher has a member variable *mFocusedWindow* which can retrieve an InputChannel associated with the current focused window so it can dispatch the input events to the activated windows.

In *Condroid*, we modified the InputReader of the InputManager. We maintained a variable *num_Foreground-Container* standing for the number of current foreground containers. This means each container can know whether it is the foreground one. If not, the modified InputReader will stop calling the *loopOnce* function to monitor whether there are any input orders. It means the InputReader in background containers will shield all input events and only the foreground container will respond to input orders.

4.4 Service Sharing Mechanism

It is important this mechanism is created because it can reduce the memory footprint. As we know, each container is a stock Android system that contains many system services. In view of the containers, these services are duplicated. However, it is not necessary to run every service in every container, such as LightsService, BatteryService, WifiService, or SurfaceFlinger, etc. A service sharing mechanism allows a device to run a single service that can be shared among all containers instead of this service being run in every container.

In stock Android, each service should register itself in the ServiceManager after it begins so the ServiceManager can maintain a global service list. In *Condroid*, we implement an interface to allow users to custom share services through the `/proc` filesystem because containers can receive the reference object of shared services in a host's `/proc` temporary filesystem. As shown in Fig. 8, users can share some unsecure services among hosts and containers. Our virtual binder driver will direct access requests from the client process in containers to the corresponding shared service in the host's `/proc` filesystem.

4.5 Filesystem Sharing Mechanism

This mechanism allows hosts to share directories of the filesystem with containers which can significantly reduce the storage usage. In Android, the filesystem can be grouped into two categories: temporary filesystem (tmpfs) and nonvolatile filesystem (nonvolatilefs). The tmpfs is a kind of memory filesystem that is dynamically created when a system is booting. However, the nonvolatilefs contain some read-only directories that can be shared among containers.

In *Condroid*, the nonvolatilefs contains two subdirectories: `/data` and `/system`. In particular, `/system` has many read-only subdirectories, such as: `/app`, `/fonts`, `/framework`, and `/lib`, etc. We offer a method where all read-only subdirectories in containers are linked to the host. This reduces storage usage and will not introduce any security issues. The size of these subdirectories is relatively large and because of this we believe the filesystem sharing mechanism is necessary when a user needs to run many containers in a single device.

5 EVALUTAION

We have implemented a prototype of Android virtualization named *Condroid* using container technology, and transplanted it to the latest Google devices, the **Nexus 5** smartphone. In order to evaluate the usability, scalability, robustness, efficiency and stability of our prototype, we carried out experiments in regards to performance impact, power consumption, booting up time, memory utilization and so on.

5.1 Methodology

We chose *Cells* as a comparison, which is the most famous solution based on container virtualization technology presented by Columbia University at SOSP'11. However, there are many differences between *Condroid* and *Cells*, and also many improvements especially in the implementation concepts of many subsystems, such as IPC, Display, and Input, etc.

It has been proven that *Condroid* works successfully with many versions of Android, however in this paper all of our experimental results presented have been collected from Nexus 5 running with Android 4.4.2 and the Linux kernel 3.4.0. To date, *Cells* can only support Nexus S with Android 4.1.2. For fair comparison, all results have been normalized to the result of the manufacturer's unmodified Android OS.

5.2 Evaluation Results

5.2.1 Booting Up Time

Booting up time is usually an important factor of the user experience. We measure the time one container spends from receiving the start command until it is ready to receive user inputs.

Fig. 9a shows the result with one container, two containers, three containers, and four containers running in the background versus that of *Cells* in the same configuration respectively.

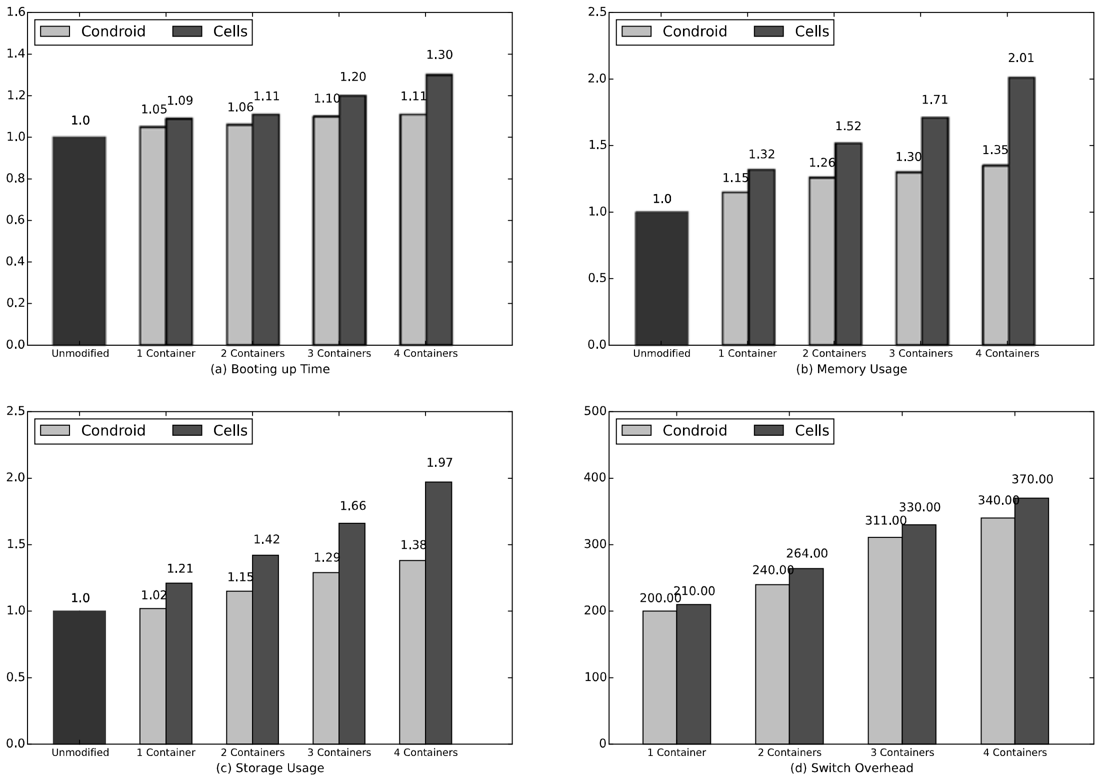


Fig. 9. Results of booting up, storage usage, memory usage and container switch.

In Fig. 9a, the x-axis of N container(s) represents the N*th* container booting up while N–1 containers run in the background. The unmodified has no containers, which makes it the mean of the device booting up.

As *Condroid* and *Cells* are running on different devices, it is pointless to directly compare the booting up time directly. Therefore, all the results are normalized to the result of the manufacturer's unmodified Android OS whose value is 1. Because *Condroids* and *Cells* both compare to their own manufacturer's unmodified Android OS, the comparison makes sense.

The observations of Fig. 9a are:

1. We measure from the time the `init` process starts to the time the Android framework application `Launcher2` starts. In this way, the unmodified Android OS takes about 15.31 seconds.
2. With an increasing number of containers booting up, time becomes larger. However, our prototype incurs no more than 11 percent overhead in all cases while *Cells* incurs an overhead of 30 percent at most.

5.2.2 Memory Usage

In *Condroid*, some system services are shared among containers in order to reduce the memory usage. This experiment measures the memory utilization of *Condroid* and *Cells* when one container, two containers, three containers and four containers run respectively. In our tests, the unmodified Android OS occupies about 297 MB memory on average.

As *Condroid* and *Cells* use different versions of the Android OS, it is pointless to directly compare memory usage. Therefore, all results are normalized to the result of the manufacturer's unmodified Android OS whose value is 1. This is something we also did in the booting up time experiment.

The observations of Fig. 9b are: our prototype incurred no more than a 35 percent overhead in all cases, while *Cells* memory utilization doubled in the worst case scenario. *Condroid* does not virtualize a framebuffer by a multiplexing framebuffer device driver which is needed to render any output to a virtual screen memory buffer in a system's RAM. In addition, *Cells* does not solve the services sharing problem.

5.2.3 Storage Usage

In our prototype many containers share several read-only directories that significantly reduce the usage of storage. The unmodified Android occupies about 619 MB storage on average.

In Fig. 9c, the y-axis scale is with the normalized unit length according to the manufacturer's unmodified Android OS as 1 unit. We also make that normalization in the experiment of booting up and memory usage.

The observations of Fig. 9c are that our prototype incurs no more than 38 percent overhead while *Cells* almost doubled in the worst case scenario. This may be because *Cells* also offers a kind of filesystem sharing mechanism and because it shares less files with each other. *Cells* may only share some configured files and some apk files.

5.2.4 Container Switch Overhead

As is already known, containers will be switched frequently in daily use which makes switching time a critical part of runtime overhead. Switching time is also an important factor of the user experience and can determine whether users are willing to use the product.

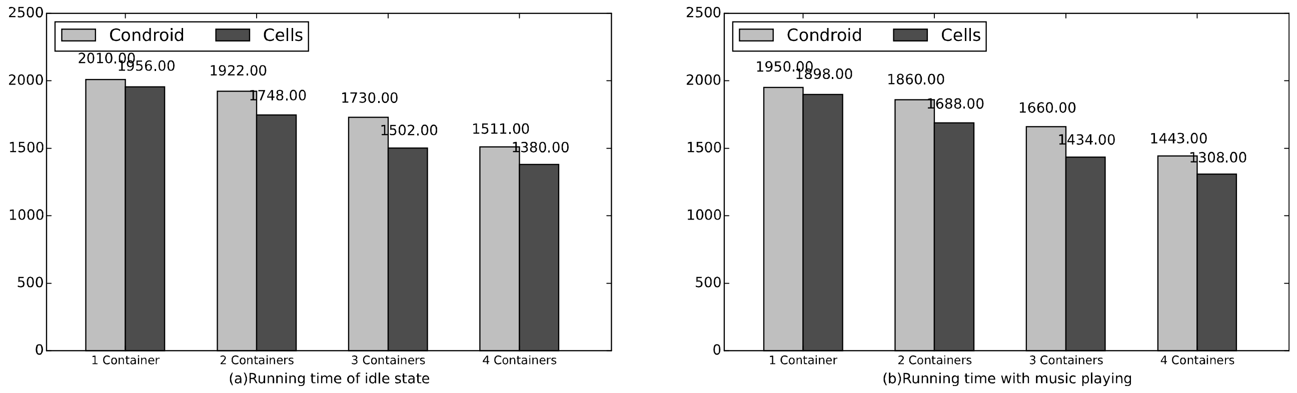


Fig. 10. The results of power consumption.

In addition, we implement our own container switch mechanism in *Condroid*, whose message goes deep into the kernel and back to the framework layer of the Android OS. With such a long message path, we intend to measure whether its overhead is heavy.

In order to achieve container switch overhead, we measure the time elapsed during one container in foreground switching compared to the background which results in another container returning to the foreground.

In Fig. 9d, the y-axis scale is one unit length representing 1 ms. Although we can achieve precision of μs, according to the results, the precision of ms is enough to tell the difference.

Fig. 9d shows the switching time of one container from the foreground to the background. From the figure, we can see our prototype incurs no more than 140 ms extra overhead. The result also shows that our prototype is better than *Cells* by an average of 20 ms in all cases.

5.2.5 Power Measurements

There are two frequently used usage scenarios to measure power consumption in mobile device benchmarks:

1) The device runs continuously in the idle state without communication over Wi-Fi or cellular and with the display backlight turned off.
2) A music player runs in the foreground with the display turned off.

Fig. 10a shows power consumption in the first scenario, which lets the phone sit idle in a low power state. While Fig. 10b shows music playing with the standard Android music playing continuously. We measure when the battery dies in these two usage scenarios.

In both figures, the y-axis scale is one unit length represents 1 minute. The results of experiments are rounded to a unit of minutes without losing the comparability of different configurations of devices.

Observations from Fig. 10 include: even though running more containers increased, the endurance time became smaller. Another observation was that our prototype incurred no more than 26 percent loss in endurance time. This is better than *Cells* with the same configuration respectively.

In addition, our new container switch mechanism proved good enough on performance.

5.2.6 Micro Benchmarks

We used the benchmark programs, which are basically equivalent to the fork + exec, fork + exit, pipe and syscall programs included in the LMbench [41] benchmark suite. We show the results of comparing the execution speed in Table 1.

To see the impact of virtualization on common operations in mobile phones, we compared UI loading time, codec performance and image file saving time. For the UI loading test, we used Qtopia [42] installed at the NOR flash memory. We prepared 100 files whose size was distributed from 10 KB to 5 MB to test the image file saving and we measured the time taken to save all image files from a NFS server to NAND flash memory. For codec tests, the WMV stream encoder/decoder was used.

Table 1 shows the performance of executing a simple syscall is the one most severely impacted because its execution path is very simple. The other benchmark programs involve fair amounts of work executed in the guest operating systems, thus the performance degradation is a little severe.

5.2.7 Macro Benchmarks

To measure performance in a macro way, we selected five benchmarks designed for measuring different aspects of an Android OS: AnTuTu v4.5.2 [43]; Quadrant Standard Edition v2.1.1 [44]; SunSpider v0.9.1 JavaScript benchmark [45]; Passmark PerformanceTest Mobile v1.0.4000 [46] and Vellamo v2.0.3 [47]. These benchmarks are designed to test 2D and 3D graphics performance, Disk I/O, Memory I/O and CPU (Framework layer, Native layer) performance. The scores they give are very powerful in explaining how well a device runs.

TABLE 1
Results of Micro Benchmarks

	Condroid		Cells	
	1 VM	2 VMs	1 VM	2 VMs
fork + exit (μs)	4,012.38	4,328.53	5,117.75	5,332.65
fork + exec (μs)	5,984.14	6,211.51	7,463.90	8,577.90
pipe (μs)	201.64	273.30	1,190.35	2,254.30
syscall (μs)	13.74	17.21	19.93	21.22
UI loading (s)	12.32	13.45	10.17	14.32
Image saving (s)	45.17	54.23	40.32	50.30
Encoding rate (fps)	5.67	5.76	7.21	7.43
Decoding rate (fps)	20.41	23.14	24.13	26.55

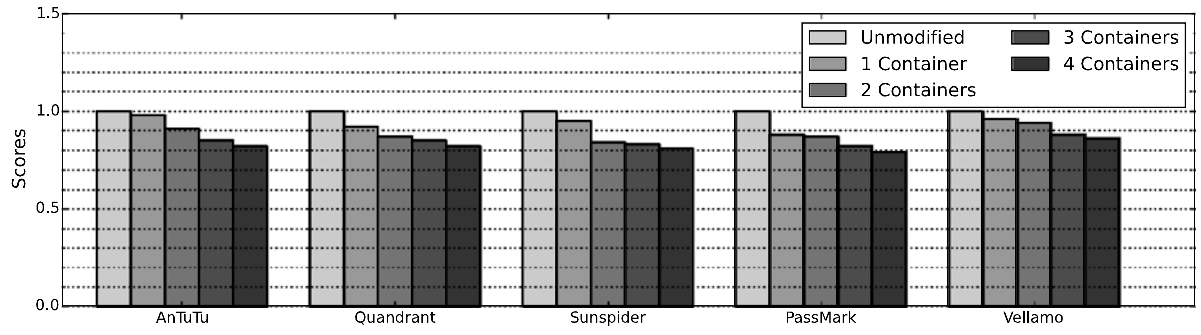


Fig. 11. The scores of five acknowledged benchmarks.

Fig. 11 shows the scores given by the five macro benchmarks with *Condroid* when one container, two containers, three containers, and four containers are already running.

As different benchmarks give scores according to different standards, the scores range from thousands of points to tens of thousands of points. Therefore, all the results are normalized to the result of the manufacturer's unmodified Android OS whose score of five benchmarks is 1. This makes the results easy to read and comparable with other configurations of Android OS.

From Fig. 11 we observe that when the number of Containers increase, worse scores are achieved, however our prototype incurs no more than a 46 percent difference in scores between the unmodified Android OS.

6 Conclusion and Future Work

In this paper, we present *Condroid*, a lightweight solution based on container virtualization technology. Unlike *Cells*, we make the most of modifications at the Android framework layer in order to achieve good portability. Our solution supports all mobile devices on the market that can run the AOSP Android system. The main contributions of this paper include: (1) we verify the feasibility of using cgroups and namespaces through LXC in an Android environment; (2) we design an efficient container virtualization prototype with several device virtualization models, such as Binder, Display and Input; (3) we present a service sharing mechanism and filesystem sharing mechanism to reduce the amount of memory and storage used. A series of experiments on the latest Nexus 5 running with *Condroid* and Nexus S running with *Cells* tells us that *Condroid* incurs near zero performance overhead, and in most experiments *Condroid* achieves better performance than *Cells*.

Future work includes supporting telephony virtualization that can provide containers with independent phone numbers. In addition, various sensors virtualization (Bluetooth, GPS, NFC, etc.) will be explored in the future.

Acknowledgments

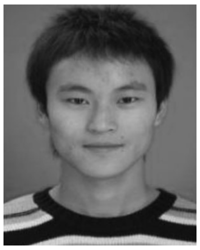
Xiaoqian Fan and Pengfei Jiang helped with running benchmarks to obtain many of the measurements in this paper. Qian Chen helped to promote this project on source on Github (**http://condroid.github.io/**). The authors wish to thank the reviewers for their comments about improving the presentation and structure of this paper. This work was funded by the National Science and Technology Major Project of the Ministry of Science and Technology of China under Grant NO. 2013ZX03003010-002 and supported partly by the key Science and Technology Innovation Team Fund of Zhejiang under Grant NO. 2010R50041. ARC Projects DP150103732, DP140103649, and LP140100816. Yang Xiang is the corresponding author.

References

[1] N. Swanner. (2014). Android tablet market share jumps to nearly 62% [Online]. Available: http://androidcommunity.com/android-tablet-market-share-jumps-to-nearly-62–20140303/
[2] International Data Corporation (IDC). (2014). Smartphone OS market share, Q2 [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp
[3] F-Secure. (2013) [Online]. Available: http://www.f-secure.com/en/web/labs_global/white-papers/reports
[4] L. Xu, W. Z. Chen, and Z. H. Wang, "Research about virtualization of ARM-based mobile smart devices," in *Proc. Int. Conf. Multimedia Ubiquitous Eng.*, 2014, pp. 259–266.
[5] R. Robert, "Survey of system virtualization techniques," *Oregon State Univ. Technical Report*, 2004.
[6] T. King, Samuel, G. W. Dunlap, and P. M. Chen, "Operating system support for virtual machines," in *Proc. USENIX Ann. Tech. Conf.*, 2003, pp. 71–84.
[7] S. Crosby and D. Brown, "The virtualization reality," *IEEE Trans. Queue*, vol. 4, no. 10, pp. 34–41, Dec./Jan. 2006.
[8] S. J. Vaughan-Nichols, "Virtualization sparks security concerns," *Computer*, vol. 41, no. 8, pp. 13–15, Aug. 2008.
[9] O.. Laadan and J. Nieh, "Operating system virtualization: Practice and experience," in *Proc. ACM 3rd Annu. Haifa Exp. Syst. Conf.*, 2010, p. 17.
[10] G. Heiser, "The role of virtualization in embedded systems," in *Proc ACM 1st Workshop Isolation Integr. Embedded Syst.*, 2008, pp. 11–16.
[11] K. L. Kroeker, "The evolution of virtualization," *Commun. ACM*, vol. 52, no. 3, pp. 18–20, 2009.
[12] M. Rosenblum Mendel and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *IEEE Trans. Comput.*, vol. 38, no. 5, pp. 39–47, May 2005.
[13] P. M. Chen and B. D. Noble, "When virtual is better than real [operating system relocation to virtual machines]," *Proc. IEEE 8th Workshop Hot Topics Operating Syst.*, 2001, pp. 133–138.
[14] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A survey on concepts, taxonomy and associated security issues," in *Proc. IEEE 2nd Int. Conf. Comput. Netw. Technol.*, 2010, pp. 222–226.
[15] M. Carpenter, T. Liston, and E. Skoudis, "Hiding virtualization from attackers and malware," *IEEE Security Privacy*, vol. 5, no. 3, pp. 62–65, May/Jun. 2007.
[16] R. Ballagas, M. Rohs, J. G. Sheridan, and J. Borchers, "Byod: Bring your own device," in *Proc. Workshop Ubiquitous Display Environ.*, 2004, vol. 2004, pp. 1–8.
[17] J. P. Shim and D. Mittleman, "Bring your own device (BYOD): Current status, issues, and future directions," in *Proc. 19th Am. Conf. Inf. Syst.*, 2013, pp. 595–596.
[18] K. W. Miller, J. Voas, and G. F. Hurlburt, "BYOD: Security and privacy considerations," *IT Prof.*, vol. 14, no. 5, pp. 53–55, Sep./Oct. 2012.

[19] S. Antonio, "New security perspectives around BYOD," in *Proc. IEEE Comput. Soc. 7th Int. Conf. Broadband, Wireless Comput., Commun. Appl.*, 2012, pp. 446–451.
[20] M. Bill, "BYOD security challenges: Control and protect your most sensitive data," *Netw. Security*, vol. 2012, no. 12, pp. 5–8, 2012.
[21] N. Singh, "BYOD genie is out of the bottle—'devil or angel'," *J. Bus. Manag. Soc. Sci. Res.*, vol. 1, no. 3, pp. 1–12, 2012.
[22] D. Jaramillo, B. Furht, and A. Agarwal, "Mobile virtualization technologies," in *Virtualization Techniques for Mobile Systems*. New York, NY, USA: Springer, 2014, pp. 5–20.
[23] X. Y. Chen, "Smartphone virtualization: Status and Challenges," in *Proc. IEEE Int. Conf. Electron., Commun. Control*, 2011, pp. 2834–2839.
[24] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 3, pp. 275–287, 2007.
[25] O. Eiferman. (2014). The real challenges of mobile virtualization [Online]. Available: http://www.cellrox.com/blog/the-real-challenges-of-mobile-virtualization/
[26] A. Aguiar and F. Hessel, "Embedded systems' virtualization: The next challenge?" in *Proc. 21st IEEE Int. Symp.Rapid Syst. Prototyping*, 2010, pp. 1–7.
[27] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis, "The VMware mobile virtualization platform: Is that a hypervisor in your pocket?" *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 4, pp. 124–135, 2010.
[28] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, no. 5, pp. 164–177, 2003.
[29] J. Brakensiek, A. Dröge, M. Botteck, H. Härtig, and A. Lackorzynski, "Virtualization as an enabler for security in mobile devices," in *Proc. 1st Workshop Isolation Integr. Embedded Syst.*, 2008, pp. 17–22.
[30] W. Enck, "Defending users against smartphone apps: Techniques and future directions," in *Proc. 7th Conf. Int. Inf. Syst. Security*, 2011, pp. 49–70.
[31] S. Wessel, F. Stumpf, I. Herdt, and C. Eckert, "Improving mobile device security with operating system-level virtualization," in *Proc. Security Privacy Protection Inf. Process. Syst.*, 2013, pp. 148–161.
[32] LXC-Linux Containers. (2014) [Online]. Available: https://linuxcontainers.org/
[33] C. Dall and J. Nieh, "KVM/ARM: The design and implementation of the linux ARM hypervisor," in *Proc. ACM 19th Int. Conf. Architectural Support Programm. Languages Operating Syst.*, 2014, pp. 333–348.
[34] D. Rossier, "EmbeddedXEN: A revisited architecture of the XEN hypervisor to support ARM-based embedded virtualization," *White Paper*, Switzerland, 2012.
[35] G. Heiser and B. Leslie, "The OKL4 microvisor: Convergence point of microkernels and hypervisors," in *Proc. 1st ACM Asia-Pacific Workshop Syst.*, 2010, pp. 19–24.
[36] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library OS from the top down," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 291–304, 2011.
[37] S. L. Scott, G. Vallée, T. Naughton, A. Tikotekar, C. Engelmann, and H. Ong, "System-level virtualization research at oak ridge national laboratory," *Future Gen. Comput. Syst.*, vol. 26, no. 3, pp. 304–307, 2010.
[38] J. Andrus, C. Dall, A. Van, T. Hof, O. Laadan, and J. Nieh, "Cells: A virtual mobile smartphone architecture," in *Proc. ACM 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 173–187.
[39] M. Kerrisk. (2013) Namepaces overview [Online]. Available: http://lwn.net/Articles/531114/
[40] P. Menage. (2014). Cgroups [Online]. Available: https://www.kernel.org/doc/Documen-tation/cgroups/cgroups.txt
[41] LMBench. (2014). [Online]. Available: http://lmbench.sourceforge.net/
[42] Qtopia. (2014) [Online]. Available: http://qpe.sourceforge.net/
[43] Antutu Benchmark for Android. (2014) [Online] Available: http://play.google.com/store/apps/details?id=com.antutu.ABenchMark
[44] Quandrant Standrad Edition for Android. (2014) [Online]. Available: http://play.google.com/store/apps/details?id=com.aurorasoftworks.quadrant.ui.standard
[45] Sunspider for Android. (2014) [Online]. Available: http://www.webkit.org/perf/sunspider/sunspider.html
[46] PassMark Benchmark for Android. (2014) [Online]. Available: http://play.google.com/store/apps/details?id=com.passmark.pt mobile
[47] Vellamo Mobile Benchmark for Android. (2014) [Online]. Available: http://play.google.com/store/apps/details?id=com.quicinc.vellamo

Wenzhi Chen received the BS, MS, and PhD degrees in computer science and technology from Zhejiang University, Hangzhou, China. He is currently a professor at the School of Computer Science and Technology at Zhejiang University. His areas of research include computer architecture, system software, embedded system, and network security. He is a member of the IEEE and the ACM.



Lei Xu received the bachelor's degree in computer science and technology from the North West Agriculture and Forestry University, Xi'an, China. He is currently working toward the PhD degree at the School of Computer Science and Technology, Zhejiang University. His current research interests include operating systems, virtualization, distributed systems, and cloud infrastructure. He is a student member of the IEEE and the ACM.

Guoxi Li received the bachelor's degree in computer science from Zhejiang University. He is currently working toward the PhD degree at Zhejiang University. His research interests include operating system architecture, OS kernel analysis and system virtualization, especially mobile system virtualization. He is a student member of the IEEE and the ACM.

Yang Xiang received the PhD degree in computer science from Deakin University, Australia. He is currently a full professor at School of Information Technology, Deakin University. He is the director of the Network Security and Computing Lab (NSCLab). His research interests include network and system security, distributed systems, and networking. In particular, he is currently leading his team developing active defense systems against large-scale distributed network attacks. He is the chief investigator of several projects in network and system security, funded by the Australian Research Council (ARC). He has published more than 170 research papers in many international journals and conferences, such as *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Information Security and Forensics*, and *IEEE Journal on Selected Areas in Communications*. He has served as the program/general chair for many international conferences such as ICA3PP 12/11, IEEE/IFIP EUC 11, IEEE TrustCom 13/11, IEEE HPCC 10/09, IEEE ICPADS 08, NSS 11/10/09/08/07. He has been the PC member for more than 60 international conferences in distributed systems, networking, and security. He serves as the associate editor of the *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *Security and Communication Networks* (Wiley), and the editor of *Journal of Network and Computer Applications*. He is the coordinator, Asia for IEEE Computer Society Technical Committee on Distributed Processing (TCDP). He is a senior member of the IEEE.