2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering

# Condroid: A Container-Based Virtualization Solution Adapted for Android Devices

Lei Xu, Guoxi Li, Chuan Li, Weijie Sun, Wenzhi Chen, Zonghui Wang
College of Computer Science and Technology
Zhejiang University, Hangzhou, China
Email: {leixu, guoxili, lichuan, luckyweijie, chenwz, zjuzhwang }@zju.edu.cn

***Abstract*— Mobile virtualization, emerging fairly recently, is considered to be a valuable way to mitigate security risks on Android devices. In this paper, we propose a lightweight Android virtualization solution based on container technology, named *Condroid*. *Condroid* includes resource isolation based on namespace feature and resource control based on cgroups feature. By leveraging them, *Condroid* can host several independent Android virtual machines. Our approach requires only a single kernel to support several Android containers. Therefore, we can get a higher performance compared to other virtualization solutions. Furthermore, our implementation presents system service sharing mechanism to reduce memory utilization and 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.**

***Keywords- Container, Virtualization, Android, Security.***

## I. INTRODUCTION

Smart mobile devices have already been an omnipresent part of our daily lives. Among them, Android claimed 61.9% of all smart device market share and nearly 79% of smartphone market share by the end of 2013 [1]. At the same time, people find Android has become an attractive target for malware because of its openness. The report of F-Secure says the number of malicious software on Android platform is accounted for 97% of the overall number of mobile malware. In 2013, malicious deduction virus rank the first place with the proportion of 23%, the frauds and process control take the second and third place respectively with 21% and 16% [2].

One promising approach to limit the effectiveness of malware is to use virtualization to help confine malware. The most important requirements of virtualization on Android devices include the following topics:

### A. Security Threats

Viruses, Trojan horses and malwares from all kinds of external attackers have attracted people's attention. However, deploying a security environment (such as encryption, digital signature, safety audit, access control, and digital certification.) on mobile device is very complicated for common 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 access sensitive data (SMS, contacts) [3]. Virtualization can offer a secure zone to store sensitive Apps, data and private info. Besides, it also can prevent malware from infiltrating the secure zone from other insecure environments.

### B. BYOD

The concept of BYOD (Bring Your Own Device) refers to the policy of permitting employees to bring personally owned mobile devices to their workplace, and to access privileged company information and applications. BYOD increases employee morale and convenience by using their own devices and makes the company look like a flexible and attractive employer [4]. It seems that the answer to BYOD is mobile virtualization [5]. Mobile virtualization enables a single device offer two or more personas with different system settings and user profiles and totally different operating environments. There are some other scenarios: (1) company needs to monitor and remote manage employees' device, but employees do not want to be monitored or controlled when they use their devices for personal purpose (telephony, gaming, web browsing); (2) company needs to backup employees' workspaces, then destroy them after work and restore them on the next workday. Mobile virtualization can make all these possible and easy to do.

This paper describes a mobile virtualization approach adapted for Android, called *Condroid*, which enables a single device to run several Android containers simultaneously and independently. We leverages namespace feature to isolated different Android containers. The purpose of each namespace (currently, 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 cgroups (control groups) feature to limit, account and isolate resource usage of process groups. Cgroups provide a mechanism for partitioning sets of tasks, and all of their future children, into hierarchical groups with specialized behavior. We also transplant the LXC (Linux Container) toolkit to Android platform. To do this, we make several modifications: replace several functions that Bionic library does not support, and replace some syscalls because of the difference in yaffs2, cross-compile by Android NDK toolchain, recompile kernel to enable cgroups and namespace support in kernel configuration.

*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.

This research is funded by 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

978-1-4799-8977-5/15 $31.00 © 2015 IEEE
DOI 10.1109/MobileCloud.2015.9

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 `/proc` filesystem; (2) a read-only filesystem sharing mechanism is proposed to reduce storage usage. Normally, people are concerned about the storage usage while multiple whole Android instances exist on single device. This mechanism make the `/system` partition of Android system to be shared among all the containers.

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 our 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 described in Section 6.

## II. RELATED WORK

Isolation mechanisms to enhance security for Android can be classified in three types: user-level isolation, OS-level virtualization and system virtualization.

Isolation based on user-level, known as *sandbox*, is a traditional way to confine malware. This solution uses different user identifiers for per application group to implement a sandboxing. The communication between applications and core Android components is restricted based on permissions, which are requested during the installation of applications. Drawbridge [6] 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: (1) RDP (Remote Desktop Protocol) is not supported to virtualized applications to render graphics in Android; (2) It does not support shared state between applications in different sandboxes; (3) It also doesn't support multiple processes bound to a single OS library.

Isolation based on OS-level virtualization, as used in our solution, is a common concept of container today. *Cells* introduced in [7], is also an Android container solution. In contrast to *Cells*, our approach spends many efforts 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 most of modifications in Linux kernel layer, and these are unlikely to be merged into the mainline because this feature is not the emphasis of standard kernel. So it may not cause kernel maintainer's attention. Our modifications mostly are in Android framework layer，these are very likely to be collected in AOSP (Android Open Source Project), if Google wishes stock Android to support virtualization.

Another approach to isolate runtime environments is system virtualization. This technology was originally developed for server and desktop. While, in recent, we find some researches have transplanted typical x86 system virtualization platforms to ARM platform:

KVM/ARM [8] 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 is not originally designed for ARM architecture. This solution is now not mature and not stable. There is a long way to modify KVM to be adaptable to ARM hardware. EmbeddedXEN [9] is a para-virtualization hypervisor specifically for ARM embedded systems. In particular, EmbeddedXEN supports heterogeneous ARM cores and keep execution overhead as low as possible. However, Solutions based on para-virtualization is not fit for mobile device. It has a complex configuration which is not easy for common user and it need to modify the guest OS code which means it can't support the latest OS and commercial closed-source operating systems. OKL4 microvisor [10] is designed to serve as a hypervisor as well as replacing the microkernel. OKL4 is a third generation microkernel of L4 heritage for large-scale commercial deployment of mobile virtualization platform. However, Microvisor has to work with device support and emulation, an onerous requirement for mobile devices which contains increasingly diverse hardware devices.

## III. SYSTEM ARCHITECTURE

As mentioned above, several related technologies have been developed over the last few years 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 [3] we designed a container-based architecture. Figure 1 shows the basic *Condroid* architecture and the modifications we make are shown as the grey parts in figure. This design can offer a better user experience: it can enable user to create, start, shutdown, and manage the containers all in device and user can switch containers more conveniently. Because most modifications are located in Android framework layer and all of these will be packaged as a ROM firmware in later.

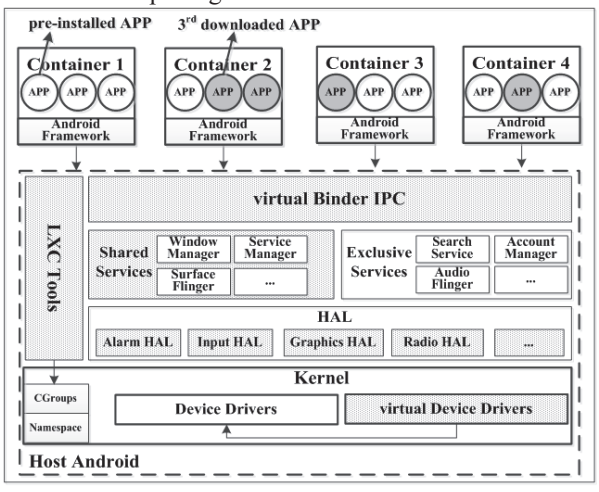



Figure 1: Overview of *Condroid* architecture

Firstly, we think it is necessary to briefly describe the purpose of each component in Figure 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 that we should never execute any untrusted application in this domain. Each complete Android OS consists of 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 cgroups and namespace feature and we also need to make some other modifications in kernel, like creating some virtual devices, writing several virtual device drivers, *etc*.
- Android Framework – It is the 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 container's Android framework.
- LXC Tools – LXC is a userspace interface for the Linux kernel containment features. Through a powerful API and simple tools, it lets Linux users easily create and manage containers. However, LXC is originally designed for Linux and what we have to do is transplanting LXC to 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 also can run the downloaded APPs.

As introduced above, *Condroid* uses a single OS kernel across all containers that virtualizes identifiers and hardware resources which means *Condroid* does not require running multiple complete Android instances, instead, it provides virtual environments in which multiple containers can be run on a single Linux kernel. *Condroid* ensures that the containers are individual, completely independent, and secure from one another in order to prevent buggy or malicious applications running in one container to adversely impact the operation of the other containers. This is done by leveraging namespace and cgroups. Each container has its own private namespace so that containers can run concurrently and use the same OS resource names inside their respective namespaces. Containers can't conflict with each other or even feel each other. This is the function of namespace feature. Meanwhile, each container has its own constrained resource because of the allocated software and hardware quotas for RAM, CPU, Disk and devices from cgroups feature.

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 in commodity OS platforms such as Linux. Android applications, however, are expected to interact with a variety of hardware devices, many of which are not designed originally to be multiplexed, and 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 such a kind of solution that is adapted for Android devices. It does so by integrating kernel-level and user-level device virtualization methods to present a complete virtual Android OS environment. Figure 1 shows the relationship between host and containers. The host is a control center and never installs any downloaded apps despite it is a complete Android OS. This design can ensure the safety of host at the most extent. 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 this container only includes pre-installed apps and trustful apps. *Condroid* can offer several secure containers and insecure containers efficiently.

In Figure 1, we can see LXC toolkit is the console of *Condroid* which is a user interface to manage containers. LXC combines cgroups and namespace support to provide an isolated environment for applications. To make it possible to use LXC in Android system, we have to make lots of modifications: (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 by Android NDK toolchain; (4) reconfiguring Linux kernel with cgroups and namespace features enabling.

In Figure 1, we also can see there are some other modifications shown as grey in both user space and kernel space. 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 Linux *proc* filesystem interface. We also designed device virtualization mechanism by creating several virtual device drivers and modifying HAL (Hardware Abstract Layer).

## IV. 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 couldn't cover all of them in this paper. We'll describe the implementation of main components in *Condroid*.

### *A. Binder System Virtualization*

Binder is a system for IPC (Inter-Process Communication) now used in the Android operating system. Binder is a

primary subsystem used ubiquitously by all Android processes. That's the reason why we should virtualize Binder first of all.

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

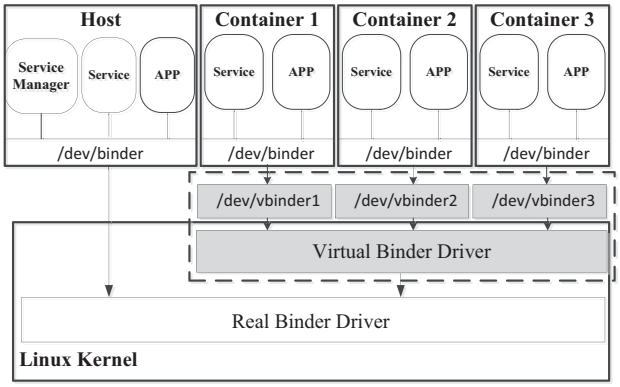



Figure 2: Binder virtualization architecture

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

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

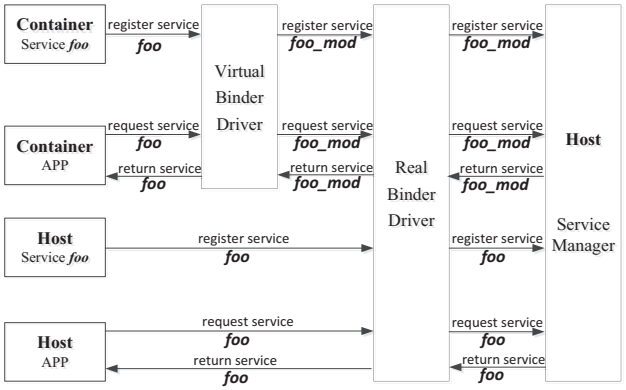



Figure 3: The workflow of Condroid with binder virtualization

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

### B. 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 Android framework rather than virtualizing the framebuffer device in Linux kernel. It can extremely reduce the memory usage without maintaining virtual hardware state and renders any output to a virtual screen memory buffer in RAM and it's also very hard to debug when you make some modifications in kernel. By the way, our solution has more flexibility and portability than *Cells* without creating any new virtual devices (*Cells* need to create `mux_fb` device).

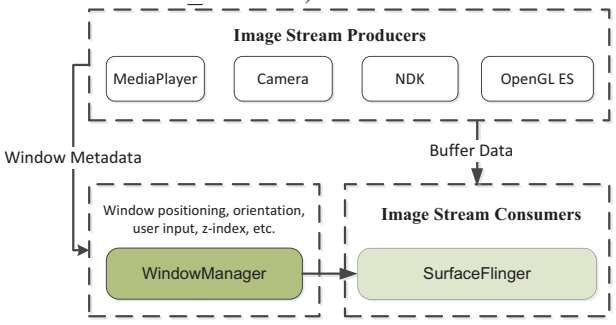



Figure 4: How surfaces are rendered

All modifications we made are in the WindowManager, 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 of the window metadata to SurfaceFlinger, Android's system service that composites visible surface onto the display. Figure 4 shows how these components work together.

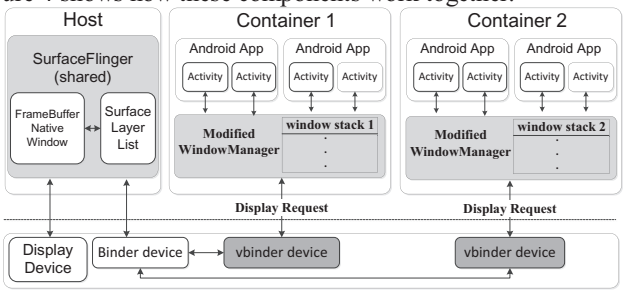
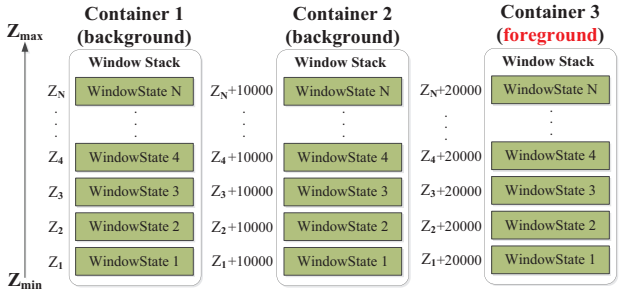
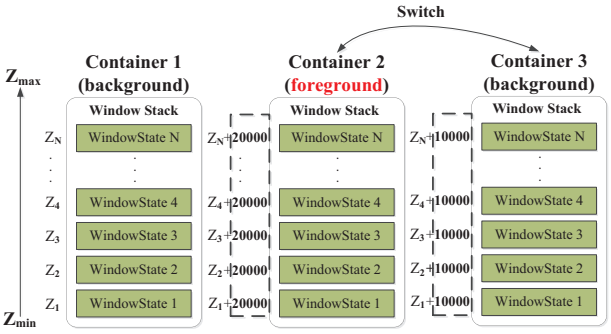



Figure 5: Architecture of display system virtualization

In Android, WindowManager maintains a window stack which is very important for SurfaceFlinger to decide which windows to be drew on the screen. Each item in window stack is a WindowState object, and WindowManager will calculate a Z-index for each item through a mapping function. SurfaceFlinger chooses the max Z-index value of WindowState to draw on the screen. Here, we should modify the mapping function of WindowManger in each container. The Z-index value of the $N_{th}$ container should add (N-1)*10000 to avoid repetition of Z-index value as shown in Figure 6(a). Therefore, the container which starts at last 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.




(a) The mapping relation when 3 containers run.




(b) The mapping relation after a switching.

Figure 6: The mapping function of modified WindowManager in Condroid.

However, while we want to switch the container 2 to the foreground we need to put all of its windows to the top of 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, e.g., in Figure 6 (b), swapping container 2 with container 3 when we press the switching shortcut key (in our prototype, press volume-up and volume-down simultaneously).

### C. Input System Virtualization

In Android, input events are all handled by InputManager. Input system virtualization is done by modifying the InputManager to let current foreground container respond the input events, and background containers will ignore. Originally, in InputManager there are two member variables: *mInputDispatcher* and *mInputReader*, pointed to an InputDispatcher object 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 the input events. They run in separate threads. Acquiescently, the InputDispatcher thread will continually call its member function *dispatchOnce* to check whether InputReader dispatches input events. If not, InputDispatcher thread will go to sleep until roused by InputReader. The InputReader thread will continually call its member function *loopOnce* to check whether user issues an input order. If yes, it will call the function *notifyKey* of InputDispatcher to wake up InputDispatcher so as to dispatch the input events to the activated windows.

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

### D. Service Sharing Mechanism

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

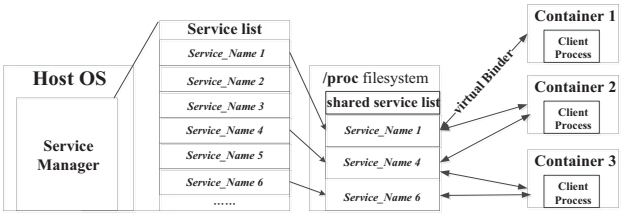



Figure 7: The service sharing mechanism in *Condroid*.

In stock Android, each service should register itself in ServiceManager after it starts. So the ServiceManager maintains a global service list. In *Condroid*, we implement an interface to allow user to custom shared services through `/proc` filesystem. Because containers can get the reference object of shared services in host's `/proc` temporary filesystem. As shown in Figure 7, user can share some not security-aware services among host and containers. Our virtual binder driver will direct the access request from the client process in container to the corresponding shared service in host's `/proc` filesystem.

### E. Filesystem Sharing Mechanism

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

In *Condroid*, the *nonvolatilefs* contains two subdirectories: `/data` and `/system`. Particularly, `/system` has many read-only subdirectories, like: `/app`, `/fonts`, `/framework`, `/lib`, *etc*. We offer a way that all of the read-only subdirectories in containers are linked to the host. It reduces lots of storage usage and will not introduce any security issues. The size of these subdirectories is so relatively large that we think this mechanism is necessary when user need to run many containers in a single device.

## V. Experiment Results

We have implemented a prototype of Android virtualization named *Condroid* using container technology, and transplanted it to the latest Google devices: **Nexus 5** smartphone and **Nexus 7** tablet PC. In order to evaluate the usability, scalability, robustness, efficiency and stability of our prototype, we carried out many experiments regarding many aspects such as performance impact, power consumption, booting up time, memory utilization and so on. In this paper, all experiment results are from Nexus 5 because we have not enough time to experiment on Nexus 7.


Figure 9: Experiments on Nexus 5 and Nexus 7.

### A. Methodology

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

*Condroid* has been proved works successfully with many versions of Android while in this paper all of our experimental results presented are collected from Nexus 5 running with Android 4.4.2 and Linux kernel 3.4.0. However, *Cells* can only support Nexus S with Android 4.1.2 currently. For fair comparison, all the results are normalized to the result of manufactory's unmodified Android OS.

### B. Evaluation Results

#### 1) Booting up Time

Usually, booting up time is an important factor of user experience. We measure the time one container spends from getting the starting command until it is ready to get user inputs. Figure 10 shows the result with 1 container, 2 containers, 3 containers, and 4 containers running in the background versus that of *Cells* in the same configuration respectively.

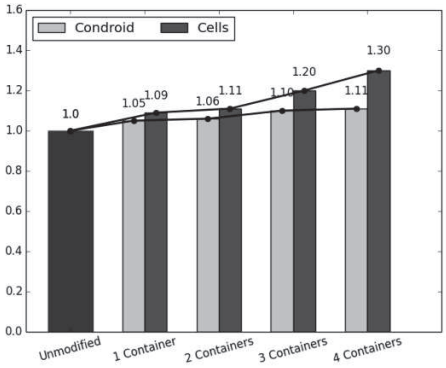



Figure 10: The results of booting up time.

We measure from the time when `init` process starts to the time when Android framework application `Launcher2` starts. In this way, the unmodified Android OS takes about 15.31 seconds. Though with the increasing number of containers booting up time is getting larger, our prototype incurs no more 11% overhead in all cases while *Cells* incurs 30% overhead at most.

#### 2) Memory Usage

In *Condroid*, some system services are shared among containers in order to reduce usage of memory. This experiment measures the memory utilization of *Condroid* and *Cells* when 1 container, 2 containers, 3 containers and 4 containers run respectively. In our tests, the unmodified Android OS occupies about 297MB memory in average.

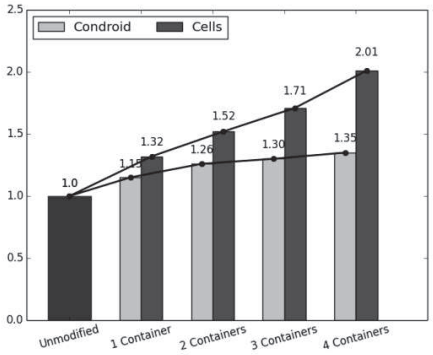



Figure 11: The results of memory usage.

Figure 11 shows our prototype incur no more 35% overhead in all cases while *Cells* gets memory utilization doubled at worst. Because *Condroid* does not virtualize framebuffer by a multiplexing framebuffer device driver which needs to render any output to a virtual screen memory buffer in system RAM. In addition, *Cells* does not solve the services sharing problem yet.

*3) Storage Usage*

In our prototype many containers share several read-only directories, which will much reduce the usage of storage. The unmodified Android occupies about 619MB storage in average.

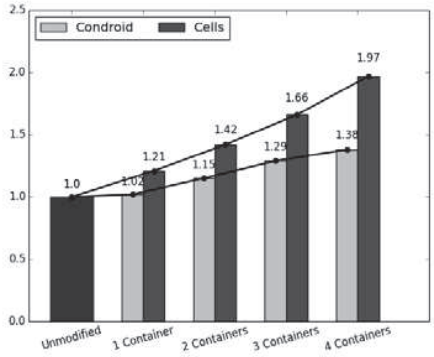



Figure 12: The results of storage usage.

Figure 12 shows that our prototype incurs no more 38% overhead while *Cells* gets almost double at worst. This may because *Cells* also offer a kind of filesystem sharing mechanism but it shares less files each other. It may only share some configure files and some apk files.

*4) Container Switch Overhead*

As is known, containers will be switched frequently in daily using, which makes switching time to be a critical part of runtime overhead. And switching time is an important factor of user experience which determines whether users are willing to use this product. We measure the time elapsed during one container in the foreground switching to the background resulting another container back to the foreground.

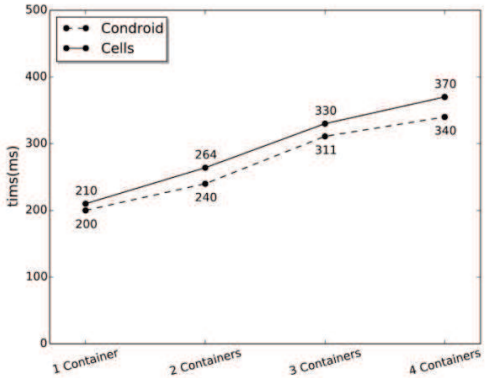



Figure 13: The results of container switching overhead.

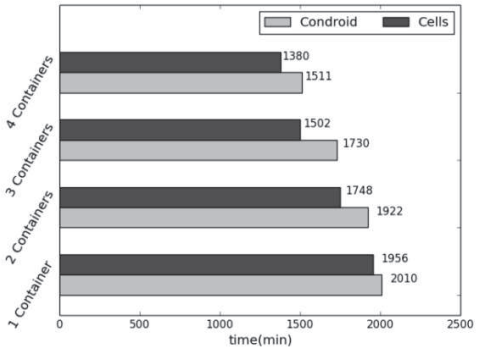
Figure 13 shows the switching time of one container from the foreground to the background. From the figure, we can see our prototype incurs no more 140ms extra overhead. And the result also shows that our prototype is better than *Cells* by 20ms averagely in all cases.

*5) Power Measurements*

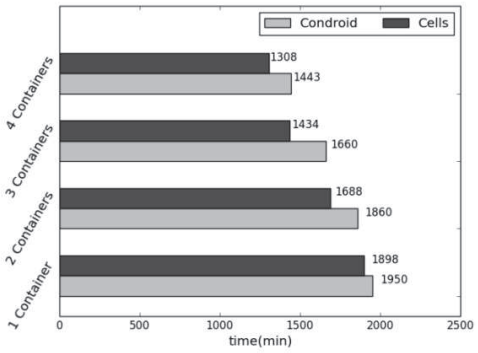
There are 2 frequently-used usage scenarios to measure power consumption in mobile device benchmarks:

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

2) A music player running in the foreground with display turned off.




(a) Running time of idle state.




(b) Running time with music playing.

Figure 14: The results of power consumption.

Figure 14 (a) shows power consumption in the 1st scenario, letting the phone sit idle in a low power state , while figure 14 (b) shows playing music with the standard Android music player continuously. We measure how long when the battery dies in this two usage scenarios. Though with increasing number of containers running enduring time is getting smaller, our prototype incurs no more than 26% loss of enduring time. This is better than *Cells* with the same configuration respectively.

*6) Macro Benchmarks*

To measure performance in a macro way, we select five benchmarks designed for measuring different aspects of Android OS: Antutu v4.5.2; Quadrant Standard Edition v2.1.1; SunSpider v0.9.1 JavaScript benchmark; Passmark PerformanceTest Mobile v1.0.4000 and Vellamo v2.0.3.

Figure 15 shows the scores given by the five macro benchmarks with *Condroid* when 1 container, 2 containers, 3 containers, and 4 containers are already running. All the results are normalized to the result of manufactory's unmodified Android OS. Though with increasing of number of Containers we get worse scores, our prototype incurs no more than 46% difference of scores between unmodified Android OS.

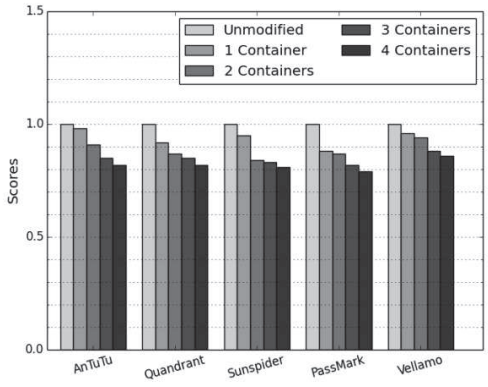



Figure 15: The scores of 5 acknowledged benchmarks.

## VI. CONCLUSIONS

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

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

## ACKNOWLEDGMENT

Mr. Guoxi Li helped with running benchmarks to obtain many of measurements in this paper. Mr. Chuan Li helped a lot to promote this project to open source on Github (http://condroid.github.io/).

## REFERENCES

[1]. http://androidcommunity.com/tag/market-share/.

[2]. http://www.f-secure.com/en/web/labs_global/ whitepapers/ reports

[3]. L. Xu, W.Z. Chen, and Z.H. Wang. "Research about Virtualization of ARM-Based Mobile Smart Devices," in *Proceedings of the International Conference on Multimedia and Ubiquitous Engineering*, MUE '14, 2014, pp.259-266.

[4]. Shim, J.P., Mittleman. "Bring Your Own Device (BYOD): Current status, issues, and future directions," in *Proceedings of the 19th Americas Conference on Information Systems*, AMCIS '13, 2013, pp.595-596.

[5]. Scarfo, and Antonio, "New security perspectives around BYOD," in *Proceedings of the 7th International Conference on Broadband, Wireless Computing, Communication and Applications*, BWCCA '12, 2012, pp.446-451.

[6]. http://research.microsoft.com/en-us/projects/drawbridge/

[7]. Andrus, J., Dall, C., Hof, A. V., et al. "Cells: a virtual mobile smartphone architecture," *in Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 173-187.

[8]. Dall, C., Jason, N., "KVM for ARM," in Proceedings of the Ottawa Linux, 2010, pp.1-12.

[9]. D. Rossier. "EmbeddedXEN: A Revisited Architecture of the XEN hypervisor to support ARM-based embedded virtualization". in *Journal of White paper*, *Switzerland*.

[10]. Heiser, G. and Leslie, B., "The OKL4 Microvisor: Convergence point of microkernels and hypervisors," *in Proceedings of 1st ACM Asia-Pacific Workshop on Systems*, 2010, pp. 19-24.

[11]. Aguiar, A. and Fabiano H. "Embedded systems' virtualization: The next challenge?" *in Proceedings of the 21st IEEE International Symposium on Rapid System Prototyping*, 2010, pp. 1-7.

[12]. Heiser, Gernot, "The role of virtualization in Embedded Systems," *in Proceedings of the 1st ACM Workshop on Isolation and Integration in Embedded Systems*, 2008, pp. 11-16.

[13]. Chen, X. Y., "Smartphone virtualization: Status and Challenges," *in Proceedings of IEEE International Conference on Electronics, Communications and Control*, 2011, pp. 2834-2839.

[14]. Soltesz, S., Herbert, P., et al., "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," *ACM SIGOPS Operating Systems Review*, Vol 41(3), 2007, pp. 275-287.