Digital Investigation 27 (2018) 38–46

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Dynamic analysis with Android container: Challenges and opportunities

Ngoc-Tu Chau, Souhwan Jung*

School of Electronic Engineering, Soongsil University, Seoul, South Korea

ARTICLE INFO

Article history:
Received 16 March 2018
Received in revised form
19 August 2018
Accepted 28 September 2018
Available online 5 October 2018

Keywords:
Android container
Android dynamic analysis
C-Android

ABSTRACT

Until now, researchers have been analyzing Android applications dynamically with the use of either emulators or real devices. Emulators are an option that testers currently have to achieve scalability. Besides, these approaches can also take snapshots which help to revert back to a known state in a matter of seconds. However, emulators are often slow in performance and contain heuristic emulation traces. As for the case of real devices, restoring mechanism and hardware utilization are limitations for increasing analysis productivity. In this paper, we developed C-Android with the motivation to seek an alternative solution to solve the existing problems. C-Android leverages Linux container technology to provide restore mechanism on the bare-metal device and to utilize the most from existing hardware. To study on the opportunities of applying Android container into the analysis, we have performed several comparative implementations between C-Android and other existing approaches. The purpose of our first comparison is to assess the compatibility between container-based (C-Android) and existing solutions. The following test is related to performance between our approach and other platforms. Lastly, the utilization test is established in order to assess the possibility to optimize C-Android hardware without causing too much reduction in the performance. The challenges of applying container technology to Android analysis are also discussed in the paper as a result of our study.

Introduction

When dealing with a great number of applications (or apps, in short), Android analysts have to build a deployment solution that provides an on-ready environment to support their work. To address this challenge, researchers have come up with various approaches which can be categorized into two basic types: 1) Emulators and 2) Real devices.

Emulators approaches give more convenience in managing analysis environments (or sandboxes) within multiple hosts. It is also possible to have more than one Android guest machines running inside a single host. In order to boost up the deployment process, analysts can use snapshot function to quickly boot into the desired state (G.Inc). Studies that used emulator-based sandboxes for their analysis included DroidScope, Andrubis, CopperDroid, VetDroid, and more (Yan and Yin, 2012; Lukas and Matthias; Tam and Khan, 2015; Zhang et al., 2013; Bläsing et al., 2010; Afonso et al., 2016). However, as security issues on Android applications become more critical, the number of Android applications that contain anti-emulator techniques also increase accordingly. As for emulator-based sandboxes, virtualization technology becomes an obstacle on the way to deal with anti-emulator techniques.

Real devices are one of the good choices for analysts because of its authentic environment towards Android apps. Using a real mobile device as an analysis environment could accommodate both requirements on compatibility and performance. Also, the real phone yields more accurate results with features like accelerometers, geo-location as well as push notifications compare to emulators. In practice, Joe Sandbox provided options for analyzing apps on the real device (Motorola Moto G model). However, this approach shows limitations as a large-scale analysis platform. A restore mechanism needs to be applied to the real device since the analysis process requires a fresh environment for each analyzed app. Besides, considering there is only one Android OS existing on a single device, the number of apps that can be analyzed at one time is depending on the number of devices available. Lastly, knowing that the analysis tools and analyzed app often share the same domain, target apps could recognize the sandbox by seeking for the

* Corresponding author.
E-mail addresses: chaungoctu@soongsil.ac.kr (N.-T. Chau), souhwanj@ssu.ac.kr (S. Jung).

https://doi.org/10.1016/j.diin.2018.09.007

analysis fingerprints.

In this paper, we have developed C-Android that was built based on container technology. The objective is to motivate the adoption of a new analysis platform on the Android research. This approach provides a better solution in comparison with emulators by producing real hardware information to the Android environment. By optimizing hardware usage, the model could increase productivity in comparison with real devices. Besides, by migrating all analysis traces into a separate layer, we could expect a close-to-native environment. Through a series of comparative experiments between C-Android and other platforms, we also raise discussions about the opportunities and challenges of applying container technology to Android analysis.

Our paper is organized as follows. Section 2 and 3 provide backgrounds about recent researches. Our developed model is introduced in section 4 following with evaluation in section 5. Section 6 summaries our research content.

Related works

Emulators

Until now, there are two emulator architectures provided for the analysts: 1) Intel-based and 2) ARM-based emulator.

X86 Emulators An Intel-based Android operating system (Android OS) is supported directly by Intel with Intel Hardware Accelerated Execution Manager (HAXM) to improve the performance of Android virtualization environment (Intel). Despite the fact that Intel-based emulator provides a fast and scalable solution, there are still many apps that are incompatible with this environment. We have verified the compatibility problem in ×86 emulator by analyzing the mixed Operating System (OS) malware dataset that provided on 27th *December, 2017* from VirusShare. We got 571 APK files after filter out the unrelated samples that belong to other OSs. Of all those Android APKs, there are 3076. *so* files that are compiled for ARM architecture and only 544. *so* files that are belonged to ×86. Based on the results, it is certain that the number of malware that executable on ×86 architecture is far less than those in the ARM architecture.

ARM Emulators ARM-based emulators are alternative options to overcome the compatibility problem. However, these solutions are slow performance, especially with the 64-bit version that requires decoding from hypervisors. Because of that, ARM-based emulators produce less productivity in an analysis process. Furthermore, since both ARM and Intel emulators are built based on virtualization, they leave potential footprints from executing instructions for Android apps to detect.

Others Android container projects

A container is originally a process running on the host OS. However, running multiple Android containers on the single host cause confliction problems since multiple Android OS may attempt to access the same hardware. In 2011, Cells has proposed a device namespace to solve the sharing device problem (Andrus et al., 2011). Device namespace is, in fact, a hardware broker which responsible to control the access of various containers to different hardware. To solve the similar problem with Cells, Condroid proposed virtual binders for containers, each will act as a broker for switching between foreground and background containers (Xu et al., 2015). Ubuntu touch project, backed by Canonical, research for an Ubuntu-based environment on Android by taking advantage of container technology. In this project, Android container is responsible to run apps while Linux layer deals with graphics and hardware connections. Ubport is known as the descendant project of Ubuntu Touch (Ubports). Recently, Anbox is another project related to Android container. The main purpose of the Anbox is to migrate the Android application onto the Linux platform in a lightweight manner. Anbox forwards hardware connection from Android container to QEMU pipe through GLES emulation. Although all of the mentioned works are related to Android container, their designs are not suitable for becoming Android sandboxes. Cells, Condroid, and Ubuntu Touch focused more on building an enhancing environment for a mobile user. Anbox aimed at building an Android app supported-environment, with support by some of the QEMU components.

Backgrounds

In this section, we deliver fundamental knowledge about Android regarding the boot process. We further explain basis designs for both regular mobile device and emulator. With the essential background provided, it will be easier to comprehend with the design of our platform.

Android booting process

In the Android process, as soon as the power button is pushed, the ROM will be booted following with bootloader stages. The boot stage is known for low-level system initialization and the kernel-loading stage is responsible for hardware, driver and file system initialization. The kernel stage helps set the system into the on-ready state before starting up the user-space programs. Unlike Linux system where initialization program is put into */sbin/init*, the Android OS keeps a *init* binary file under the root directory. The *init* program is started after kernel stage, following with others core processes that specified in the **init. rc** file, as shown in Fig. 1. The others Android services and apps are originated after the finish of core processes.

Emulator design for android OS

As depicted in Fig. 2, the Android Emulators are deployed on another OS based on virtualization technologies. In Linux, the

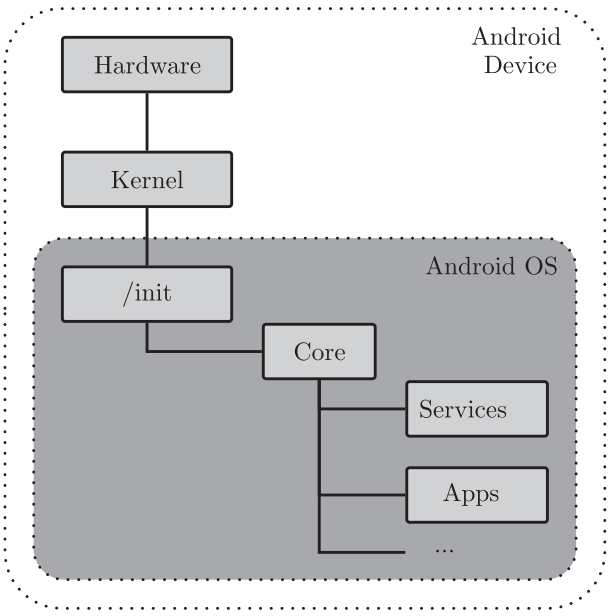

Fig. 1. Common booting process in android OS.

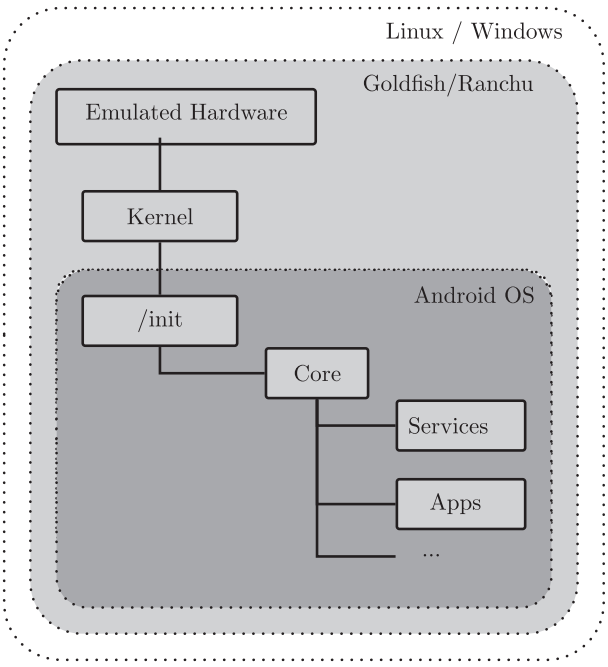

Fig. 2. Common design for android emulator.

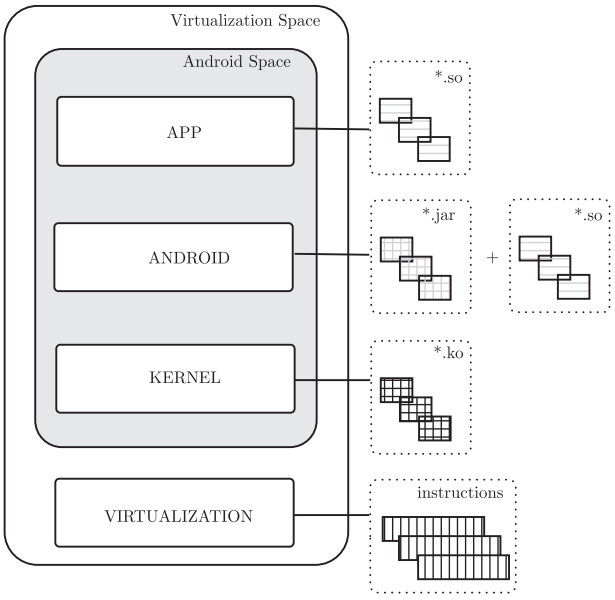

Fig. 3. Layers of android analysis.

virtualization technology is supported by Quick Emulation (QEMU) as a machine emulator and Kernel-based Virtual Machine (KVM) as a backed module (Server; Golden, 2011). For X86 Emulators, app performance can be sped up with support from Intel through the use of HAXM. Android OS is operated on QEMU-based virtualization boards under the code name of Goldfish (QEMU version 1) and Ranchu (with support from QEMU version 2). With the support from QEMU, hardware features like Bluetooth, WIFI, and GPS are available to the emulator. As for the kernel layer, the dedicated kernel module options are enabled to support drivers and file systems according to the virtual hardware. The other booting sequences that go after kernel stage are alike to the real device.

Android analysis approaches

App analysis on the real device is usually targeted at three layers inside Android space: 1) App layer, 2) Android Framework, and 3) Kernel layer. Throughout those layers, there are several ways to intercept and manipulate behaviors of target apps. The Android space in Fig. 3 illustrated three analysis layers of real devices.

Kernel-level Analysis This is a well-known approach that aims at tracing system calls that invoked by apps. As for Android OS, A. Schmidt and the others have introduced an enhanced security solution as well as malware detection at kernel-level in the year of 2008 (Schmidt et al., 2008). In 2011, I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani has proposed CrowDroid which also take advantages of **strace** for detecting malware behaviors (Burguera et al., 2011). Zheng, Min and Sun et al. have introduced DroidTrace, a **ptrace**-based Android analysis system to collect behaviors from Android app (Zheng et al., 2014). For practice, Frida is an instrumentation tool that supports memory-based syscall hooking using its interceptor function (Ravnås). At Kernel-level, type of data for collection can be various: not only system calls but also network, memory information, process information and more (Tam et al., 2017).

Framework-level Analysis In this layer, the Android framework and Dalvik virtual machine are altered for enhancing either security or analysis. In 2011, the authors of MockDroid provide users with an ability to manipulate app permission to prevent a malicious app from doing harmful activities (Beresford et al., 2011). I-ARM-Droid was introduced in 2012 by B. Davis et al. as a solution to monitoring apps by reworking on the Dalvik bytecode (Davis et al., 2012). In the same year, M. Grace et al. has introduced RiskRanker, a zero-day Android malware detection (Grace et al., 2012). The model has been developed based on information collected from both static and dynamic analysis. In their paper, the secure API and Dexloader behaviors can be collected from a modified Android framework. This solution helps to improve the performance of their virtualization-based protection methods.

App-level Analysis This level is usually conducted through either of the following methods: 1) Instrumentation, and 2) App Repackaging. Instrumentation is a technique that allows the injection of dynamic code into a target process in order to manipulate app behaviors. In 2012, C. Mulliner has introduced his research on Binary Instrumentation on Android (ADBI) (Mulliner, 2012). The proposed tool injects a dynamic hooking library into Android app to extract and handle behaviors. Although the binary instrumentation is not a new technique, Mulliner has proven the feasibility of this method on Android. Frida and Xposed Framework are well-known instrumentation tools available online to support analysis of Android apps (Ravnås; Rovo89). App repackaging is another method for tracing and modifying app behaviors that have been used by both analysts and malicious developers. In 2012, W. Zhou, Y. Zhou, X. Jiang et al. proposed DroidMOSS. Their model took advance of fuzzy hashing technique to detect the changes from app repackaging behaviors (Chow et al., 2008). In the following year, W. Zhou, Y. Zhou, M. C. Grace et al. have introduced PiggyApp as a scalable detection solution to detect malicious repackaged apps (Zhou et al., 2013). App repackaging can also be used for analysis app behaviors. In 2016, Rasthofer et al. has proposed HARVERSTER that take advantages of repackaging technique to remove triggering requirements and to improve analysis coverage (Rasthofer et al., 2016).

Emulator-level Analysis As for the case of the Android emulator, there is another layer of analysis known as the virtualization layer. This layer focuses on obtaining traces while translating instructions

from guest OS to host OS. For the records, researchers have come up with this idea a long time ago. In 2008, Jim Chow et al. has introduced their proposal about performing dynamic analysis by taking advantage of binary translation layer (Chow et al., 2008). A. Dinaburg and P. Royal et al. have introduced Ether, a solution that collects malware behaviors inside chip equipped with Intel VT technology (Dinaburg et al., 2008). Panorama is another research that proposes dynamic taint tracking based on QEMU-layer (Yin et al., 2007). Last but not least, CopperDroid is also a QEMU-based model aiming at reconstructing the behaviors of malware (Tam et al., 2015). Although these solutions proved effective against malware, they are still vulnerable from hypervisor-based detections (Petsas et al., 2014; Paleari et al., 2009; Maier et al., 2014; Lindorfer et al., 2011).

Proposed model

We consider the adoption of operating-system-level virtualization method (which has the other name as Linux container) into Android analysis to overcome the problems that exist on both real devices and Android emulators. So far, to the best knowledge of the author of this article, the application of the container technology in Android analysis has not been mentioned and put into consideration. For that, we have decided to develop a specific analysis platform for Android based on container technology. Ideally, by migrating the Android sandbox into a process, we could reduce deployment time and improve transparency to Android apps in comparison with hardware virtualization technology. Besides, the second layer of OS provided in our design help isolate the domain for analysis and application domain. The isolation approach helps reducing effort to hide analysis traces in comparison with real devices, as illustrated in Fig. 4. By forwarding all required hardware into the Application domain, our approach provided a more native environment in comparison to other virtualization-based approaches.

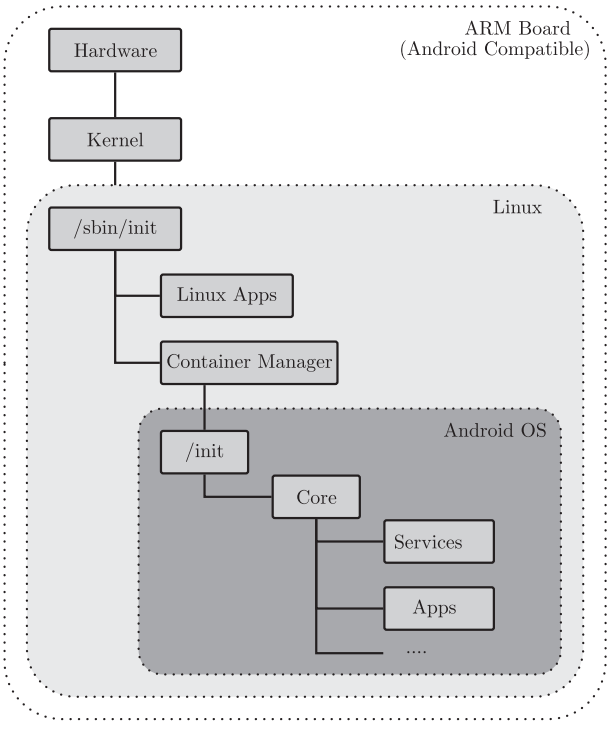

Fig. 4. Design of C-Android.

Design for Android container

As we explained in the previous Android Booting Process, the first user-space process will be started as soon as all kernel threads are started and the root file system is mounted. In order to boot up the Linux system instead of a normal Android OS, we must to prepare a Linux root file-system and let it be mounted during the booting process. Since Android kernel shares basic features with Linux kernel, it is possible to boot into Linux system with only some minor tricks.

In order to deploy Android environment, we chose LXC as container management tool. After examining the source code of alternative application-based approaches like Docker, rkt or LXD, we found out that they have limited user configuration from the core libraries. Because of that reason, those container managers can only access to a restricted number of devices. Since LXC is a system-layer container tool, it leaves the security and resources configuration to the user. With that, we can freely share necessary hardware information for our environment. Since Android OS is highly depending on the hardware information, sharing of the device nodes from host to container space is needed. By applying control group rules into the LXC configuration file, we are able to share most of the available hardware devices into Android environment.

A normal Android system is partitioned into several components including system, data, cache, and root. Those components are usually flashed into corresponding disk partitions and are mounted during the boot process. In our approach, we extracted Android components from image files and mounted those directories during container boot process. To provide Linux-based privileged users with appropriate permissions, we converted those permissions defined in **uevent** files into **udev** rules.

Uevent to Udev permission conversion

For Linux-based operating systems, devices that are recognized by the kernel will be stored in the directories each labeled with a device name and contain files that stored attributes of the device. By that way, the kernel can send the uevent signal to the userspace for notifying whenever there is a connection or disconnection from the device. A daemon is executed at startup for management purpose. In Linux-based OS, this daemon is named as **udev** and in Android OS, the daemon is known as **ueventd**. At OS bootup time, the udev daemon read device rules that usually stored at*/etc/udev/rules.d/*.rules* to determine how to handle the devices. For ueventd, the device rules are stored in*/uevent.rc*. In order for Android container to recognize the hardware, we must share hardware information from Linux space to the Android space. Before that, we must ensure that the Linux space could recognize device information. In order to do that, it is convenience to convert from uevent rules to udev rules.

Each line in *uevent. rc* file contains one configuration for a device. The format for one line is as follows: **device_name mode user_id groud_id** (Google, 2018). For udev, the usual format for a device information is as follows: **action kernel owner group mode**. To convert from uevent to udev rules, we first set the device action as ADD so that the device permissions can be managed at booting time. The other values can be matched as follows: **{kernel, device_name}, {owner, user_id}, {group, group_id}, {mode, mode}**.

Discussions on the design

Clustering Android Container Nodes Since each of the Android Container hosts contains a Linux layer, analysts can take advantage of various available programming languages and tools to develop

clustering agents. Android container is actually deployed from folders and files, analysts could create a template in any specific moment and reuse as a new container. It is also possible to share templates between hosts.

App Analysis on Android Container As illustrated in Fig. 5, the Android container could support the same number of analysis layers that a real device could, with a better efficiency. At the kernel level, analysts could build a kernel module that affects only the Android container environment. We could also put more kernel access restrictions on Android space without affecting the Linux environment. At the framework layer, framework and Dalvik modification approaches could be applied to our platform with the use of simple copy and paste commands. For application layer analysis, since the Linux layer holds full privileges and could act as Recovery mode, it is possible to install instrumentation tools into our Android environment with only small adjustments in the setup scripts.

Our model inherits all properties of a container. First of all, the access to Android files and folders is viable through **proc** filesystem. Furthermore, users from Linux space can attach themselves to Android environment as root by using console command that provided by Linux container. Through our implementation, we have used **lxc-attach** command as root privilege to connect to the Android environment. So far, performing auto analysis on a real device requires a lot of customization on Android environment including the setup of root privilege or enabling of Android Debug Bridge. Those customizations could be exposed by anti-analysis techniques. Because of that reason, analysts need a lot of effort to hide those traces. With container attachment bridge, analysts could act as root user inside of Android environment without the need to install superuser binary files and to enable ADB daemon. Although LXC console might be detected by the Android application, it is easier to focus on hiding a single trace than multiple traces.

Hardware Utilization in C-Android. One advantage of using container approach instead of real devices is the fact that we can limit the use of hardware features to the need of an analyzed app. In short, it is a waste for a target app that only requires small resources to run on a full hardware supported Android environment. By analysis meta-data provided by apps, we could limit the number of resources needed to deploy an analysis environment.

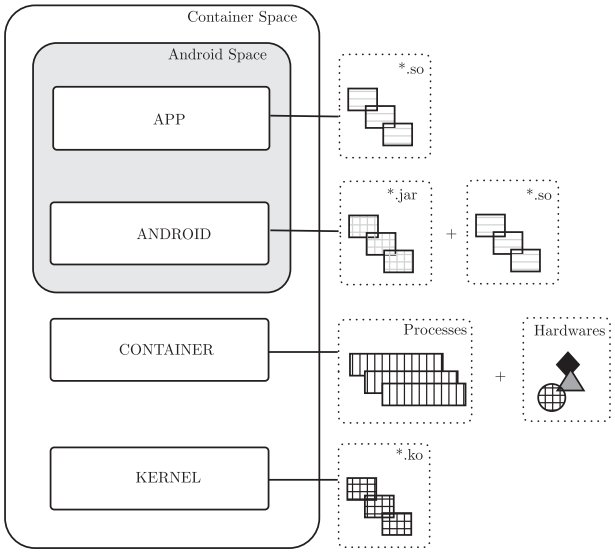

Fig. 5. Analysis layers in Android Container.

Evaluation & discussions

Environment setup

In order to evaluate C-Android, we have installed the C-Android cluster which contains more than one C-Android nodes. Each of C-Android host contains one agent to provide connection to our private cloud. In fact, C-Android can be ported into real devices. However, in our experiment, we chose to apply our proposed model into ODROID-XU4 provided by Hardkernel. The reason for using ARM-board instead of real devices is because we want to take advantages of available USB ports to extend the number of mobile features. With that, we could deploy more than one Android containers with the same support on a specific mobile feature. At the time of writing, we have successfully developed C-Android host with Bluetooth, GPS, WIFI, and data support. We are planning to provide Telephony related features in the near future. For platform under test, we have prepared 4 representatives for each analysis platform including C-Android (Container-based), Rooted Nexus 5× (Real device), AVD (ARM emulator), and Nox (X86 emulator). Real device and all Emulators are kept as-it state, with ADB enabled for support experiment scripts. C-Android is deployed without ADB and su traces since they could use LXC channel for accessing the Android environment and acquiring root privilege.

Evaluations

Compatibility test

Samples Preparation Most Android apps can be executed in the Android environment since app developers expect their product to be downloaded and widely used. However, there are apps that decline to operate on analysis environments that contain superuser or analysis traces. Because of that reason, we collected and sorted out into 5 types: financial, gaming, antivirus, malware, and banking apps. Notes that since Calling and SMS features are not yet supported in C-Android, the test samples are also non-telephony apps. Our original dataset contains apps that have been manually collected from the year 2017. However, since the dataset is outdated and does not bring the fresh result to our experiments, we decided to crawl recent samples on Google Store with the help of open source crawling tools. Since there are a limited number of apps that can be crawled at the same time, we extended our crawling to more than one country. By applying the search term accordingly to the type of apps (financial, antivirus, banking, and gaming), we have collected a total number of 60 banking, 47 financial, 68 gaming, and 53 antivirus apps. The number of crawling locales are including American/New_York, Asia/Shanghai, Asia/Seoul, Africa/Johannesburg, Asia/Kolkata, and Europe/Berlin. The samples are collected in May 2018. In summary, we have 72 banking, 57 financial, 77 gaming, and 68 antivirus apps in our dataset.

Investigation On Benign Apps The implementation processes are as follows: For each of the application, we extract the content inside to get main activity and package name. The applications that are unable to extract is considered as a failure. Also, applications that do not contain the main activity are also marked as failed. Throughout our implementation, we only executed the main activity and filtered error reports from the *logcat* channel. The applications that are unable to exit properly and if the *logcat* contains one of the following messages: “Crashed, Force closed, Application error”, the apps are also considered as failed.

Normally, UI/Application Exerciser Monkey or other UI automation tools are used in application analysis for improving report quality. However, we decided to not use the automation tools since they may generate a random event in every run, which may be resulted in inconsistency output.

Results Fig. 6 illustrates our implementation results. The detail of our results are as follows:

- **Banking** For 72 banking samples, there are 59 apps that successful run on the C-Android environment, which is higher than both rooted device and emulator. The results indicate that banking apps are concerned about whether their applications are running on a rooted or emulated environment.
- **Antivirus** In a total of 68 apps, C-Android have surpassed both rooted device and emulator environment with 54 success apps. Among the applications that have been put into the tests, there are several applications that detected emulated or rooted environments and still allow for continued use.
- **Gaming** Gaming apps also concern in the rooted environment as only 58 games are running on the device. However, running on the emulator is allowed by many game developers. C-Android environment shows more compatible with running 68 games without fail.
- **Financial** Financial applications are quite sensitive to running applications on abnormal environments. In this category, C-Android is exceptional with 52 applications that can run on the platform.

Discussion on The Experiment Although C-Android could provide better compatibility comparing to other platforms, there are some apps that are marked as failed. After an investigation, we found the following reasons: 1) Failed to extract app due to protection, 2) Incompatible SDK version, 3) Does not contain main activity, and 4) Platform does not contain dependency app. The 4^{th} error occurs on some banking apps that require the installation of additional security app.

It is acknowledged by the authors that since all devices did not have any analysis supported (or blue-pill) apps installed, rooted device and virtualization ×86 can be bypassed easily by secure apps. Possibly, if the rooted device is installed with instrumentation tools, it can be protected against anti-root and anti-analysis. Still, it might take more efforts to cover rooted and debugging fingerprints. Moreover, analysts might have to deal with binary translation detection techniques and compatibility problems in ×86 emulators.

Compatibility Tests on Malware Samples Although the results of anti-analysis and anti-emulator benign apps have shown the advantages of container-based in comparison with other platforms, the effectiveness of this design compared to current malware samples remains a question. Since malware apps are provided by VirusShare that carried out the inspection through the use of emulator and real device with analysis supported apps, there is no meaning to perform the platform comparison. Instead, we decide to perform the stress tests on large malware samples on C-Android.

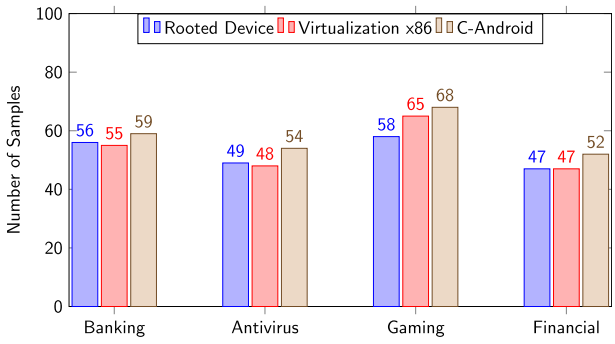

Fig. 6. Apps compatibility test.

The test is conducted to clarify that C-Android could provide the same compatibility as other platforms. For the dataset, we decided to test the effectiveness of the platform on malware samples collected from VirusShare in December 2017 (Virus share, 2018). The total number of malware that put into the test is 571. We used the same script on the previous compatible test. The result is 552 worked and 19 failed apps.

Discussion on The Experiment After investigating, we found out that there are 8 apps that are unable to install, 6 apps were unable to extract, and 5 apps that contain do not contain any activity. After further investigation, we found that 6 apps encountered an "unable to extract" error due to lack of space in the */data/app/* folder. Fixing this can be done by deleting the */data/app/vmdl***xxx***tmp* files. 5 apps do not contain activity, in fact, they can be installed and executed since the apps are provided as services.

The reason for installation problems are:

1. **INSTALL_FAILED_DUPLICATE_PERMISSION** This error occurs when the Package Manager detects that the target app declared to have a custom permission with the same name as the other existing permission. In our case, the app has declared "android.permission.ACCESS_DOWNLOAD_MANAGER". which already exists in an installed package: **com. android.providers.downloads.**
2. **INSTALL_PARSE_FAILED_UNEXPECTED_EXCEPTION** This error is reported specifically by PackageParser as follows: "Failed reading classes. dex in java. util.jar.StrictJarFile" and "META-INF/ MANIFEST. MF has invalid digest for classes. dex". This error usually occurs when there is a change in the classes. dex file for instrumentation purposes, such as using a soot framework (Soot, 2018).
3. **INSTALL_PARSE_FAILED_MANIFEST_MALFORMED** This error is reported by PackageParser in Package Manager. The content of the specific error is: "*/data/app/vmdl***xxx***.tmp/base.apk* (at Binary XML file line #**yyy**): ⟨*provider*⟩ does not include authorities attributes". Specifically, the provider attribute in the manifest file built inside the target app does not contain the authority attribute. According to information provided by Google developer website, at least one authority must be specified (Android developers: provider). Because of that reason, this is not the problem caused by our platform.
4. **INSTALL_FAILED_INVALID_APK** This error is reported by the Android Package Manager system: "Zip: **xxx** extraneous bytes at the end of the central directory." Due to the encapsulation of the APK in the Zip format, the addition of extra bytes in the Central Directory resulted in the Package Manager refusing to install the application.
5. **INSTALL_PARSE_FAILED_NO_CERTIFICATES** This error is reported by the Android system when a user attempts to install an application that has not been properly signed.

Based on the review of the installation errors, we are convinced that those errors are not caused by C-Android platform. In order to recheck our argument, we have tested the installation of those error apps on the Nexus 5, which is a real Android device. Results confirmed that all 8 apps were unable to install on a real device.

Performance test

For this experiment, we have estimated the average time for C-Android to restore from a template, and compare with other approaches. At first, we have tried to compare our model with Baredroid (Mutti et al., 2015), which so far is the only solution for backup and restore the real device from a specific state using Team Win Recovery Project (TWRP) tool. Unfortunately, the source code that shared on Github contains a lot of bugs, and even after

spending time fixing all the bugs, the Android still cannot properly boot. Because of that reason, the benchmark test only compares C-Android and Nexus 5 phone (No baredroid supported) with the same specification of memory and CPU. The Android version is Lollipop since it is still one of the most popular Android versions for analysis. We also tried to compare with ARM emulator, however, the deployment time is really slow with Lollipop version. It has no meaning to test on emulator ARM version 4.4 because of the difference in runtime (From Dalvik to ART). In order to perform the test, we divided the test into 3 stages: Instance creation, deployment, and clean-up with maximum estimation time up to 200 s. Fig. 7 shows the benchmark results between 4 approaches.

We could see that in the case of virtualization, the first stage (Instance creation) is faster compared to both real devices. In this stage, C-Android requires the delete and re-copy of Android template. By looking at the figure we could see that container-based provides a competitive time with Virtualization approach in the creation stage. Since the kernel provided for ODROID-XU4 does not support modern container technology, we cannot apply overlay file-system and checkpoint features into our platform. With those modules, it is possible to improve the output of the creation stage in C-Android. The deployment time for ARM virtualization is actually too slow for the other approaches. The figure also points out that real devices and $\times 86$ emulator are slower than our approach. By looking further, we figure out that since the container-based approach does not need to go through kernel booting process on deployment, it requires a shorter time for deploying an Android container than a real device. The termination time for real devices and virtualization are faster than our C-Android due to the difference in implementation. For C-Android, we calculated the time regarding the status of LXC container using the *lxc-info* command. The state of the container will change into STOPPED after the Android process is fully terminated. For other platforms, we need to calculate the availability of Android Debug Bridge (ADB). Since the ADB daemon is stopped sooner than Android OS, the real OS termination time will be later than our estimation result. Consider that fact, the container-based still provide the comparative in stopping time with other approaches.

Utilization experiments

In this section, we have performed 2 experiments. In the first experiment, we adjust various resources on the Android container in order to measure the reduction of performance in case of resource sharing. For the second experiment, the main objective is to confirm whether multiple containers could provider better result or not.

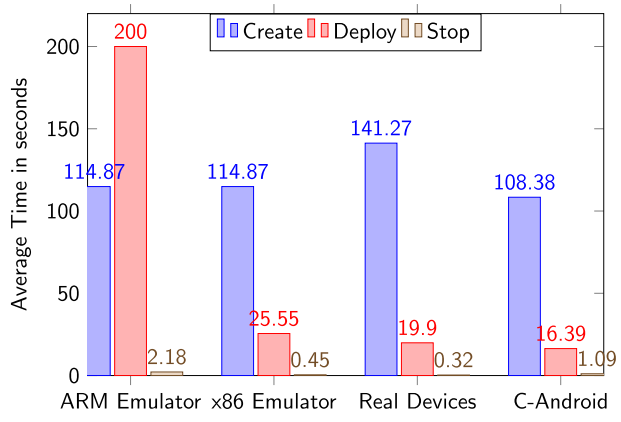

Fig. 7. Benchmark test results.

Hardware Adjustment Experiment The experiments have been done through the change of 3 metrics: 1) Set of CPU cores, 2) CPU Share between containers, and 3) Memory share. We choose the booting time as a comparison unit since it constants for every test. Table 1 illustrates the profiles that we have used to test and the booting time for each profile. In Table 1, the "Minimum Specs" refers to the lowest specification that can be used to deploy the Android container. By looking at the result in Fig. 8, we could see that the changes from CPU share and memory do not affect much of the performance. On the other hand, the number of CPU set clearly reduces the performance of an Android container. Since our ODROID board has 8 cores, it could deploy up to 8 containers at the cost of $\left(\frac{cpu-1}{cpu-8}*100\right) \approx 291\%$ slower in booting times than full Android container.

Hardware Sharing Experiment For this experiment, we have updated the patch for multiple containers based on device namespace (Andrus et al., 2011). We chose succeeded 42 new financial sample to perform the experiment. The samples are divided into 2 sets with each contain 21 apps.

For the first test, we assigned each set into one Android container and estimated the time required to finish all samples. For the second test, we assigned 42 samples to a unique container that running on the host. Noted that since we share 8 CPU cores for 2 containers, each container will have $\left(\frac{cpu-4}{cpu-8}*100\right) \approx 133\%$ slower in speed in comparison to the speed of running a single container on the host.

After the test, the results are as follows: 1) Two containers finished their task with approximate 624 s and 519 s, and all apps were analyzed at 624 s 2) Unique container finished it tasks with approximate 1114 s. The result has shown that hardware sharing helps reduce analysis time and provide better productivity.

Discussions and lessons learned

With the implementation results, we come up with some pros and cons in applying the container-based model to the Android analysis.

Advantages Since the analysis tools can be moved to the Linux layer, it is possible to build a normal Android environment without the need of installing analysis or rooting tools inside. With *lxc-attach*, we still can achieve root privilege in Android container without **su** binary. By that, we can bypass anti-emulator, anti-analysis, and anti-root check provided in some sensitive apps. Hardware sharing can help to provide both full Android and reduced environment. By taking advantage of the ARM board, we can provide better compatibility in comparison with the X86 emulator.

Table 1
References tables for hardware tests.

	CPU set	CPU share	Memory
CPU-8	8	1024	2048
CPU-4	4	1024	2048
CPU-2	2	1024	2048
CPU-1	1	1024	2048
Share-2	8	512	2048
Share-4	8	256	2048
Share-8	8	128	2048
Mem-2	8	1024	1024
Mem-4	8	1024	512
Mem-8	8	1024	256
Minimum Specs	1	128	256

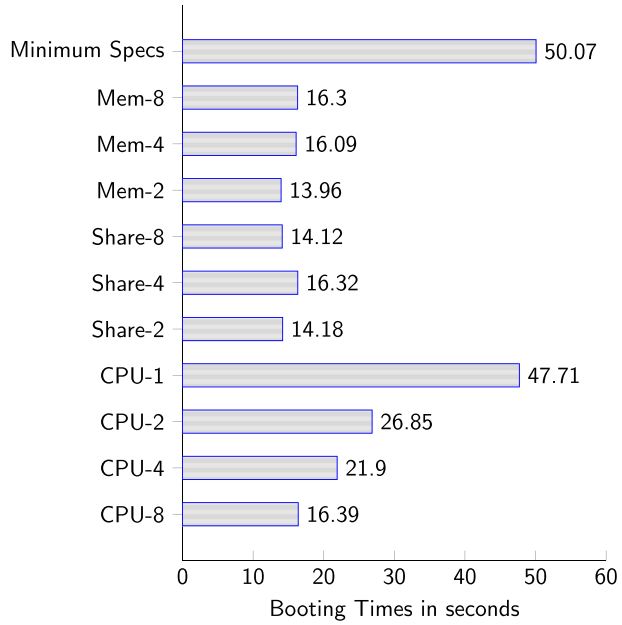

Fig. 8. Utilization experiments.

Disadvantages Container model is still vulnerable to kernel-based and container-based attacks.

Our recommendations Android container should have the same security design as other container-based approaches. Security of the Android container can be enhanced by applying a proper Linux Security Module (LSM) and alternative security models like seccomp, capabilities. For multiple containers, a solution should be applied for managing hardware resource in a proper manner.

Conclusions

Today, as the technology of detection and prevention of analysis grows, the discovery of new analysis platform becomes a matter of concern. For that purpose, this paper proposes a new approach for analyzing Android applications based on container technology. Based on the results of the article, it is possible to see that the combination of Android and container technology creates a more transparent platform than virtualization. Compared with mobile phones, this approach provides better hardware optimization and scalability solution. Although the application of container technology to Android is not new, adopting the Android container presents new opportunities and challenges in the Android analysis industry.

Acknowledgement

This work was supported by Institute for Information & communications Technology Promotion grant funded by the Korea government(MSIT) [No.2016-0-00078, Cloud based Security Intelligence Technology Development for the Customized Security Service Provisioning].

References

Afonso, V.M., de Geus, P.L., Bianchi, A., Fratantonio, Y., Kruegel, C., Vigna, G., Doupé, A., Polino, M., 2016. Going native: using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In: NDSS.

Android developers: provider. URL https://developer.android.com/guide/topics/manifest/provider-element.

Andrus, J., Dall, C., Hof, A.V., Laadan, O., Nieh, J., 2011. Cells: a virtual mobile smartphone architecture. In: Proceedings of the Twenty-third ACM Symposium on Operating Systems Principles. ACM, pp. 173–187.

Beresford, A.R., Rice, A., Skehin, N., Sohan, R., 2011. Mockdroid: trading privacy for application functionality on smartphones. In: Proceedings of the 12th Workshop on Mobile Computing Systems and Applications. ACM, pp. 49–54.

Bläsing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S.A., Albayrak, S., 2010. An android application sandbox system for suspicious software detection. In: Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on. IEEE, pp. 55–62.

Burguera, I., Zurutuza, U., Nadjm-Tehrani, S., 2011. Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. ACM, pp. 15–26.

Chow, J., Garfinkel, T., Chen, P.M., 2008. Decoupling dynamic program analysis from execution in virtual environments. In: USENIX 2008 Annual Technical Conference on Annual Technical Conference, pp. 1–14.

Davis, B., Sanders, B., Khodaverdian, A., Chen, H., 2012. I-arm-droid: a rewriting framework for in-app reference monitors for android applications. Mobile Security Technologies (2), 17, 2012.

Dinaburg, A., Royal, P., Sharif, M., Lee, W., 2008. Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. ACM, pp. 51–62.

Golden, B., 2011. Virtualization for Dummies. John Wiley & Sons.

Google. Ueventd [online, Cited 2018]..

Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X., 2012. Riskranker: scalable and accurate zero-day android malware detection. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services. ACM, pp. 281–294.

G. Inc, https://developer.android.com/studio/run/emulator.html#quickboot.

Intel, https://software.intel.com/en-us/articles/intel-hardware-accelerated-execution-manager-intel-haxm.

Lindorfer, M., Kolbitsch, C., Comparetti, P.M., 2011. Detecting environment-sensitive malware. In: International Workshop on Recent Advances in Intrusion Detection. Springer, pp. 338–357.

L. M. e. a. W. Lukas, N. Matthias, Andrubis: a Tool for Analyzing Unknown Android Applications.

Maier, D., Müller, T., Protsenko, M., 2014. Divide-and-conquer: why android malware cannot be stopped. In: Availability, Reliability and Security (ARES), 2014 Ninth International Conference on. IEEE, pp. 30–39.

Mulliner, C., 2012. Binary Instrumentation on Android. summercon.

Mutti, S., Fratantonio, Y., Bianchi, A., Invernizzi, L., Corbetta, J., Kirat, D., Kruegel, C., Vigna, G., 2015. Baredroid: large-scale analysis of android apps on real devices. In: Proceedings of the 31st Annual Computer Security Applications Conference. ACM, pp. 71–80.

Paleari, R., Martignoni, L., Roglia, G.F., Bruschi, D., 2009. A fistful of red-pills: how to automatically generate procedures to detect cpu emulators. In: Proceedings of the USENIX Workshop on Offensive Technologies (WOOT), vol. 41, p. 86.

Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S., 2014. Rage against the virtual machine: hindering dynamic analysis of android malware. In: Proceedings of the Seventh European Workshop on System Security. ACM, p. 5.

Rasthofer, S., Arzt, S., Miltenberger, M., Bodden, E., 2016. Harvesting runtime values in android applications that feature anti-analysis techniques. In: NDSS.

O. A. V. Ravnås, https://www.frida.re.

T. Rovo89, Xposed framework. retrieved from http://repo.xposed.info.

Schmidt, A.-D., Schmidt, H.-G., Clausen, J., Yuksel, K.A., Kiraz, O., Camtepe, A., Albayrak, S., 2008. Enhancing security of linux-based android devices. In: Proceedings of 15th International Linux Kongress. Lehmann.

E. Server, Enabling Intel® Virtualization Technology Features and Benefits.

Soot, Aug. 2018. A Java Optimization Framework. https://github.com/Sable/soot.

Tam, A. F. e. a. K., Khan, S., 2015. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In: Network and Distributed System Security Symposium. NDSS).

Tam, K., Khan, S.J., Fattori, A., Cavallaro, L., 2015. Copperdroid: automatic reconstruction of android malware behaviors. In: NDSS.

Tam, K., Feizollah, A., Anuar, N.B., Salleh, R., Cavallaro, L., 2017. The evolution of android malware and android analysis techniques. ACM Comput. Surv. 49 (4), 76.

Ubports, https://ubports.com.

Virus share [online, Cited 2018].

Xu, L., Li, G., Li, C., Sun, W., Chen, W., Wang, Z., 2015. Condroid: a container-based virtualization solution adapted for android devices. In: Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2015 3rd IEEE International Conference on. IEEE, pp. 81–88.

Yan, L.K., Yin, H., 2012. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In: Proceedings of the 21st USENIX Conference on Security Symposium, Security'12. USENIX Association, Berkeley, CA, USA, 29–29. http://dl.acm.org/citation.cfm?id=2362793.2362822.

Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E., 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. ACM, pp. 116–127.

Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X.S., Zang, B., 2013. Vetting undesirable behaviors in android apps with permission use analysis. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications

Security. ACM, pp. 611–622.
Zheng, M., Sun, M., Lui, J.C., Droidtrace, 2014. A ptrace based android dynamic analysis system with forward execution capability. In: Wireless Communications and Mobile Computing Conference (IWCMC), 2014. International, IEEE, pp. 128–133.
Zhou, W., Zhou, Y., Grace, M., Jiang, X., Zou, S., 2013. Fast, scalable detection of piggybacked mobile applications. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy. ACM, pp. 185–196.