Memory overhead analysis of container-based Android sandboxes

Aditi Goyal - BT/CSE/190057

Yatharth Goswami - BT/CSE/191178

Supervisor: Dr. Debadatta Mishra

April 25, 2022

Acknowledgments

We are deeply indebted to Prof Debadatta Mishra for allowing us to work with him. We are thankful to him for having regular meetings despite his busy schedule and helping us acquire the necessary perseverance for solving a research problem.

Abstract

Virtualization, as a technology, allows one to more efficiently use physical resources at hand. The increasing usage of smartphones has led to a fast-growing demand for mobile virtualization. This would allow a smartphone user to run multiple instances of the same app in a device. This also has security benefits in the sense that usually a naive user may give extra permissions to an unknown app. Use of multiple "virtual smartphone environments" would prevent harm to the complete system. In this way, virtualization also acts as a sandbox for testing of applications.

In this report, we discuss one of the recently-proposed solutions : VPBox which employs a container based android sandboxing mechanism. VPBox claims to provide a novel solution having properties like transparency and stealthiness which make it difficult to detect presence of virtual environments. The paper provides a framework for running virtual phones directly on smartphone devices. We study their approach and try to adapt it for running on android emulators. The approach suffers from scaling of memory usage with an increasing number of virtual phone instances. We perform experiments to analyze the cause of this memory overhead as well.

Contents

Chapter 1

Introduction

Mobile phones are used by almost all people and generally the end users are not aware of potential security threats they might introduce while installing/ running untrusted applications. Sandboxing is a mechanism which provides an isolated environment so that one can execute or test untrusted or untested programs. It achieves this by providing a tightly controlled set of resources for guest programs to run in. As mobile phones are used for critical tasks these days, people are trying to employ sandboxing techniques for mobile phones. Sandboxing techniques introduce their own overheads. We look at one such sandboxing solution : VPBox [4] and we analyse the memory overheads for the same. We first discuss some basic terminologies to understand virtualization mechanism.

1.1 Theory Review

Transparency: For a system to be transparent, there are 2 requirements - a transparent environment should show no appreciable difference with its underlying device, it should exhibit the same hardware effects as if it were run without a virtual environment and virtualization should be supported for all devices.

Stealthiness: An attacker should not be able to fingerprint the presence of virtual machines, including any of its virtualized components.

Namespace: It is feature of the Linux kernel that partitions kernel resources

such that one set of processes sees one set of resources and another set of processes sees a different set of resources. It helps in multiplexing a physical resource across groups of processes. VPBox has introduced a new namespace - device namespace. This namespace was introduced to differentiate (group) processes belonging to different (same) virtual phones. It is basically a label attached to each process which marks it with the virtual phone it belongs to.

1.2 Virtualization Technique

In VPBox, virtualization takes place at 2 levels : kernel level and user level.

Kernel level

Binder is an inter-process communication mechanism specific to Android. To provide isolation between phones, it is necessary that processes belonging to different device namespaces do not communicate through the binder driver. Binder is made aware of the device namespace and allows communication only between processes of same device namespace. GPS virtualization is also done at kernel level. The GPS service is provided by an on-device chip, which only allows one connection at a time. Virtualization is done by changing the driver code so that location information received from the chip is forwarded to multiple clients simultaneously.

User level

User-level device virtualization is necessary because some hardware vendors provide proprietary software stacks that are completely closed source. Binder driver virtualization is the basis of user level virtualization technique which allows a service process in the VP to share the corresponding service in the host system. For the services registered in binder's ServiceManager (a special binder service which stores various services registered inside binder for IPC). VP's Binder data structures are modified to store a new specific handler which points to a host binder's ServiceManager. Therefore, this mechanism allows a VP's service process to share the corresponding service in the host system.

1.3 VPBox model

VPBox adapts the foreground-background model proposed by Cells [3]. Out of the multiple VPs running on the mobile, one of the VP is the foreground VP which is displayed at any time, and has direct access to hardware while background VPs have shared access. An app is provided to switch between foreground and background VPs.

As discussed in the VPBox paper, memory usage is a bottleneck for scalability of number of virtual phones. Memory optimization techniques like AUFS filesystem, Linux KSM (Kernel same page merging) have been implemented and we have done experiments by enabling these features. AUFS is used to mount the read only partition of the VP system. KSM enabling is done to merge processes across virtualizaed guests/processes. The low memory killer kills background and inactive processes which consume large amounts of RAM. Virtualization of low memory killer is done for it to become aware of the namespace and changed the scheduling si that background VP's processes are killed more often than foreground VP's processes.

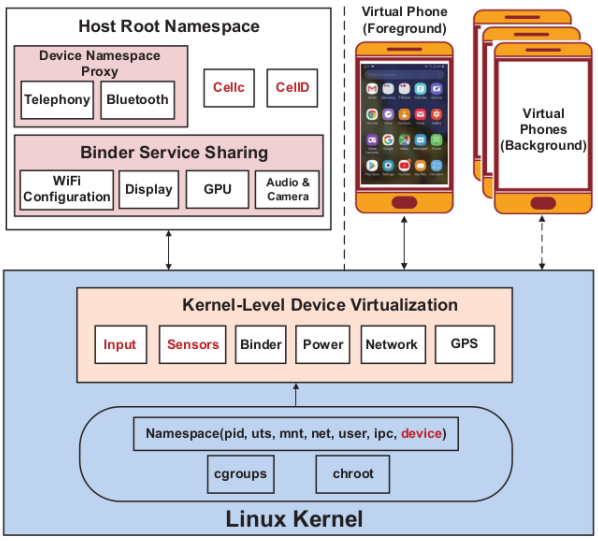

Figure 1.1: VPBox Architecture Overview

Chapter 2

Building VPBox for Emulators

The physical device that we have emulated is Pixel 3a XL.

2.1 System Requirements

To build the source code[1], we have the following requirements:

- Operating system : Ubuntu 20.04 LTS
- JDK version : OpenJDK version 9
- Android source code : android-10.0.0_r33
- Android Emulator API version : API version 29
- Linux kernel version : goldfish-kernel-4.14.112

Also, we configured kvm and QEMU for achieving native phone like runtime performance. It allowed us to check if there are runtime overheads introduced by VPBox.

2.2 Kernel modifications

The bonito kernel used by the VPBox source code [2] is for ARM based devices. For emulators, we had to manually inspect the changes they had applied to bonito kernel and adapt them for goldfish kernel. The kernel configuration file (`.config`) was also modified to include device namespace,

AUFS and a few others. Configurations irrelevant for emulators like touch screen related configurations which were enabled by VPBox were simply ignored.

2.3 Framework and System modifications

The frameworks and system folder given in the VPBox source code are a subset of the vanilla android source code. The files in vanilla source code can be classified into 3 categories:

- Present in VPBox source code as it is.
- Present in VPBox source code and modified.
- Not present in VPBox source code.

For the files of first kind, nothing needs to be done. For the files of second kind, after a manual verification, their contents can be copied/adapted. For the files of the last kind, the files should be kept as in vanilla. There are files which are only in VPBox source code, they were copied to correct locations, like switching app to `packages` directory, cells services to `vendor` directory.

Some of the configuration files (`.rc` files) have been modified to include services needed by VPBox. Also, certain virtualization components like GPU virtualization were disabled for running VPBox on emulators. We also modified the app used for switching between virtual phones to allow 5 virtual phones at a time. The demo for VPBox running on emulator can be found here.

2.4 Building VPBox

Here we list the steps we followed to build the VPBox project and run it on emulator on Ubuntu 20.04 for Pixel 3a XL phone:

First we installed the required version control tools and other packages required for aosp project.

2.4.1 Setting up a linux build environment

Installing required packages

Required packages were installed with the following command:

```
>> sudo apt-get install git-core gnupg flex bison build-
   essential zip curl zlib1g-dev gcc-multilib g++-multilib
   libc6-dev-i386 libncurses5 lib32ncurses5-dev x11proto-core
   -dev libx11-dev lib32z1-dev libgl1-mesa-dev libxml2-utils
   xsltproc unzip fontconfig
```

Installing kvm and qemu

These were needed for hardware virtualization for accelerated emulator performance. For setting up these, following steps were followed:

First we checked that the processor we are using supports hardware virtualization using the command:

```
>> cat /proc/cpuinfo | grep 'name'| uniq
```

For intel processor, we checked the VT-X technology flag. The flag was set meaning our system was ready for virtualization.

Next, we checked whether virtualization was enabled or not using the command:

```
>> egrep -c '(vmx|svm)' /proc/cpuinfo
```

In our case, it was not enabled. We enabled Intel Virtualization Technology (Intel VT) from the BIOS.
To install KVM, we ran the following command:

```
>> sudo apt install -y qemu qemu-kvm libvirt-daemon libvirt-
   clients bridge-utils virt-manager
```

To confirm that the virtualization deamon - libvritd-daemon was running, we executed the command

```
>> sudo systemctl status libvirtd
```

The output showed that the deamon was active(running), and we proceeded by enabling the deamon by

```
>> sudo systemctl enable --now libvirtd
```

To check if the KVM modules were loaded, we ran the command:

```
>> lsmod | grep -i kvm
```

The output of the above command contained kvm_intel.

2.4.2 Source Control Tools

Installing Repo

Repo unifies Git repositories when necessary, performs uploads to the Gerrit revision control system, and automates parts of the Android development workflow. We ran the following commands to install repo.

```
>> export REPO=$(mktemp /tmp/repo.XXXXXXXXX)
>> curl -o ${REPO} https://storage.googleapis.com/git-repo-
   downloads/repo
>> gpg --recv-key 8BB9AD793E8E6153AF0F9A4416530D5E920F5C65
>> curl -s https://storage.googleapis.com/git-repo-downloads/
   repo.asc | gpg --verify - ${REPO} && install -m 755 ${REPO}
    ~/bin/repo
```

Note: In case the bin directory is not present in the home directory, one needs to first create it and then follow then above mentioned steps.
To verify that repo installation is successful, we ran

```
>> repo version
```

The output for it was

```
<repo not installed>
repo launcher version 2.17
(from /usr/bin/repo)
```

One should expect a similar output, the repo version might be different.

2.4.3 Downloading the Android Source

We create an empty repository to hold the files. For our project, we stored it in `aosp/` folder. The commands that follow are written accordingly.

```
>> mkdir aosp
>> cd aosp
```

Configured git with our name and email address.

```
>> git config --global user.name Your Name
>> git config --global user.email you@example.com
```

Ran repo init to get the latest version of Repo.

```
>> repo init -u https://android.googlesource.com/platform/
   manifest -b android-10.0.0_r33
```

The branch android-10.0.0_r33 is for a particular set of mobile devices (the branch used by VPBox). A successful initialization ends with a message stating that Repo is initialized in the working directory. The client directory contained a .repo directory where files such as the manifest are kept.

To download the Android source tree to working directory from the repositories as specified in the default manifest, we ran:

```
>> repo sync -c -j$threadcount
```

2.4.4 Downloading VPBox source code

We cloned VPBox source code in the parent directory of `aosp` project using

```
>> git clone git@github.com:VPBox/Dev.git
```

After this, we manually merged `system`, `frameworks` and kernel as described in the above sections. Other than this, there are 2 more modifications:

Correcting files copied from VPBox

In `aosp/frameworks/native/include/powermanager/Ipowermanager.h`, we changed the declaration of Wakeup function to include one more parameter : int reason. Parameters can be seen from the file:
`aosp/frameworks/native/services/powermanager/IPowerManager.cpp`

Modifications from AOSP build

The file `aosp/device/generic/goldfish/tools/mk_combined_img.py` is in python2. The following changes were made to make it python3 compatible:

1. Changing the function encode('hex') to .hex()
2. Add parenthesis for print statements
3. Wherever dd_comm shell command is called in the file, typecasting the seek parameter to int before converting to string

The VPBox source code does not have proper symbolic links. We checked the correct symbolic links from the vanilla aosp and corrected them.

Selinux policy in cells has a file 'cell1.te' which gives errors on building. This file was deleted as it was not required.

The .rc files which have been taken from cells are (these were different in vanilla source code and VPBox source code) :

1. system/statsd.rc
2. system/cppreopts.rc
3. vendor/android.hardware.camera.provider@2.4-service_64.rc
4. vendor/hw/init.bonito.rc
5. vendor/hw/init.sargo.rc

Note: These paths are relative to `system/core/rootdir/cells` and these need to put at appropriate locations as per vanilla aosp code.

In the file `system/libhidl/transport/ServiceManagement.cpp`, modified the variable `static const char* inithidlservice` to include android.hardware.nfc, android.hardware.power, android.hardware.health, android.hardware.gnss, android.hardware.radio and android.hardware.graphics.allocator.

2.4.5 Building Android

First, we separately built the modified goldfish kernel using:

```
>> make -j$threadcount
```

The kernel image was at `arch/x86_64/boot/bzImage` relative to kernel directory.
Next, we built Android source code.The following commands were executed with pwd as `aosp` directory.

Setting up Environment

The following command sets up the environment:

```
>> source build/envsetup.sh
```

Choosing a target

For running android on emulator, the target was set to sdk_phone_x86_64.

```
>> lunch sdk_phone_x86_64
```

Built the code with the following command

```
>> make -j$threadcount
```

Note: Building gave errors related to unupdated ABI references which were resolved by running the command given in the error log.

Once the source was successfully built, we got the `out` directory which contained images. We copied the custom kernel image that we built to `out/target/product/generic_x86_64/kernel-ranchu`. After this, we ran the emulator with the command

```
>> emulator -no-snapshot-save -no-snapshot-load -memory 8192
```

Chapter 3

Collecting Memory Usage Data

3.1 Algorithm

In the kernel file `kernel/fork.c`, we have implemented a breadth first search algorithm. It takes as input a process id, and walks over VM areas of all the processes in the process subtree of the given process id. Also, only those processes which belong the device namespace of the given process id are considered.

3.2 Data collected

As mentioned above, we iterate over all the VM areas of processes. For each page in the VM area, we first find out whether it is physically mapped. We ignore pages which have not been mapped physically. For all the remaining pages, we keep track of the physical mapping, VM area permissions, PTE permissions and the file path which was mapped to the VM area in which the page belonged. Anonymous mappings were also maintained separately.

3.3 Implementation

The information mentioned above is collected by using the pseudo filesystem sysfs. We have introduced a VPBox attribute memusage which gives us a kernel callback at the time when we give some pid to sys/kernel/vpbox/memusage using adb shell. To collect the data, we found the process

id's of host init, and the init processes of virtual phones and gave them as input to our algorithm. The data was taken without running any applications. Also, mappings created by `goldfish_pipe` were not taken into consideration because these mappings are created because of using qemu and these will not be observed in the physical phones.

3.4 Experiment 1 : Only Host

Only the host is switched on and all the virtual phones (VPs) are switched off. We look at the space of all unique physical addresses and maintain a count of how many virtual addresses map to each physical address. Then, we count how may such physical addresses were mapped by unique virtual addresses, how many were mapped by 2 virtual addresses and so on. The bar graph is given below:

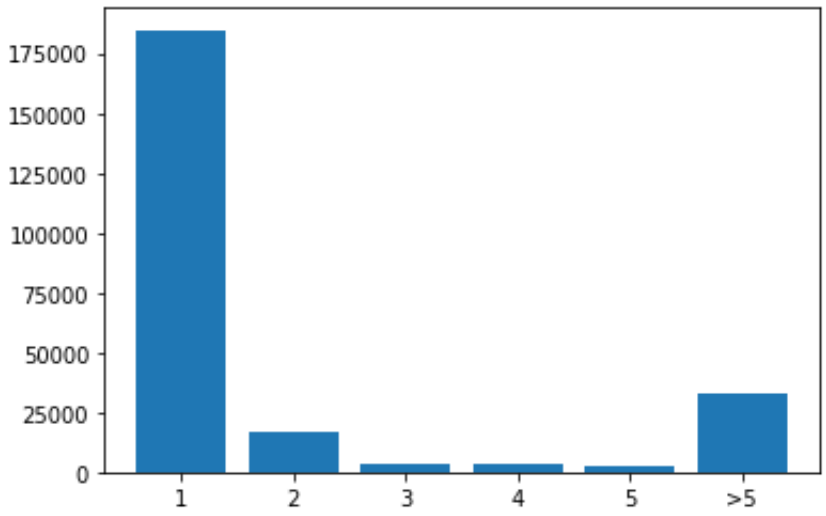

Figure 3.1: Only host bar graph

The corresponding pie chart :

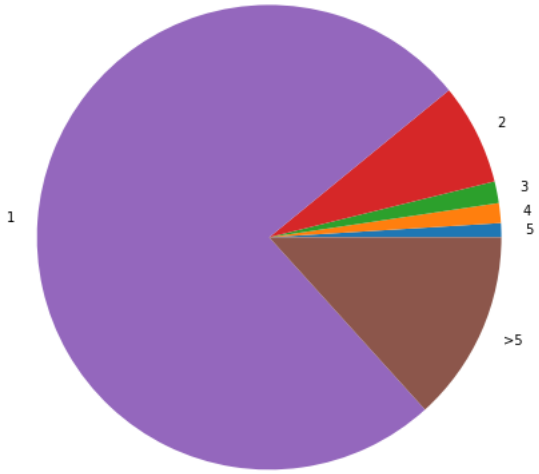

Figure 3.2: Only host pie chart

The total memory consumption given by unique physical pages is 952 MB in this case. The sharing that is there across virtual addresses is sharing across processes which would mainly be due to COW (Copy on Write).

3.5 Experiment 2 : Host + VP1

The host is switched on, and the first VP is switched on. The data for both is measured after this. Plots are made in the same way as described in Experiment-1. The total memory consumption given by unique physical pages is 1388 MB in this case. Host uses 956 MB memory and VP1 uses 795 MB. Out of this, the memory usage that is common in host and VP1 is around 363 MB. In the pie chart, it can be seen that the proportion of pages mapped by 2 virtual addresses has increased. There is sharing across phones. Anonymous mappings have been merged by KSM.

The bar chart is as follows

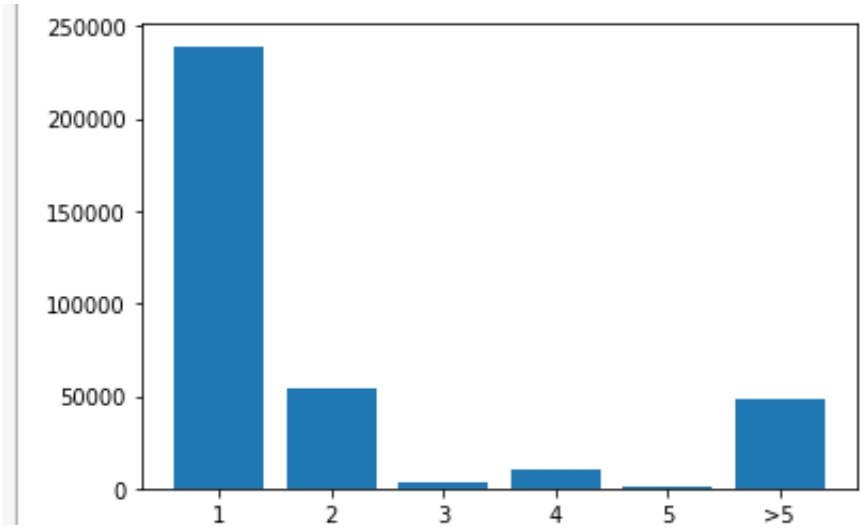

Figure 3.3: Host + VP1 bar graph

The corresponding pie chart :

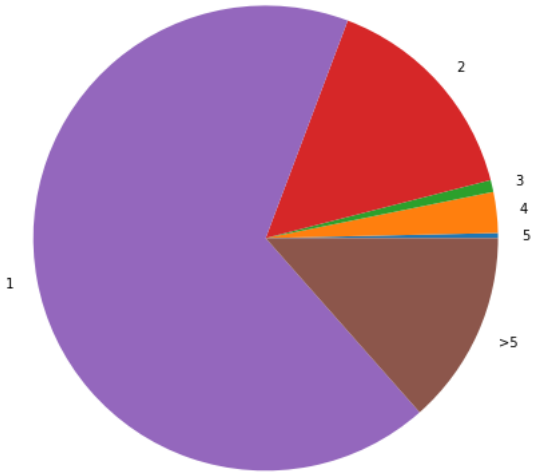

Figure 3.4: Host+VP1 pie chart

3.6 Experiment 3 : Host + VP1 + VP2

Host, VP1 and VP2 are switched on. The data for all the three is measured after this. Plots are made in the same way as described in Experiment-1. The total memory consumption given by unique physical pages is 1697 MB in this case. Host uses 896MB, VP1 uses 744MB and VP2 uses 763MB. In the pie chart, it can be seen that the proportion of pages mapped by 3 virtual addresses has increased. There is sharing across all the phones. All the VPs share 334MB of memory.

The bar graph is given below:

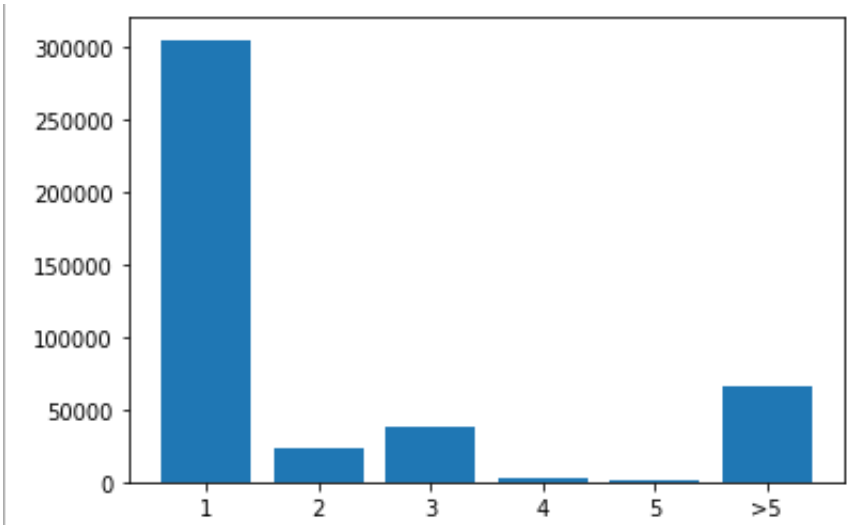

Figure 3.5: Host+VP1+VP2 bar graph

The corresponding pie chart :

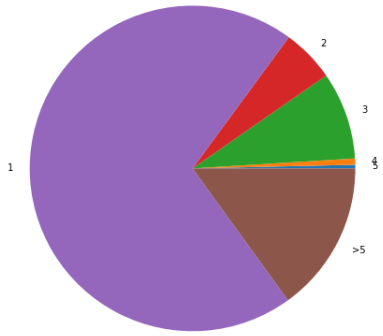

Figure 3.6: Host+VP1+VP2 pie chart

The summarized plot for the 3 experiments done is as follows:

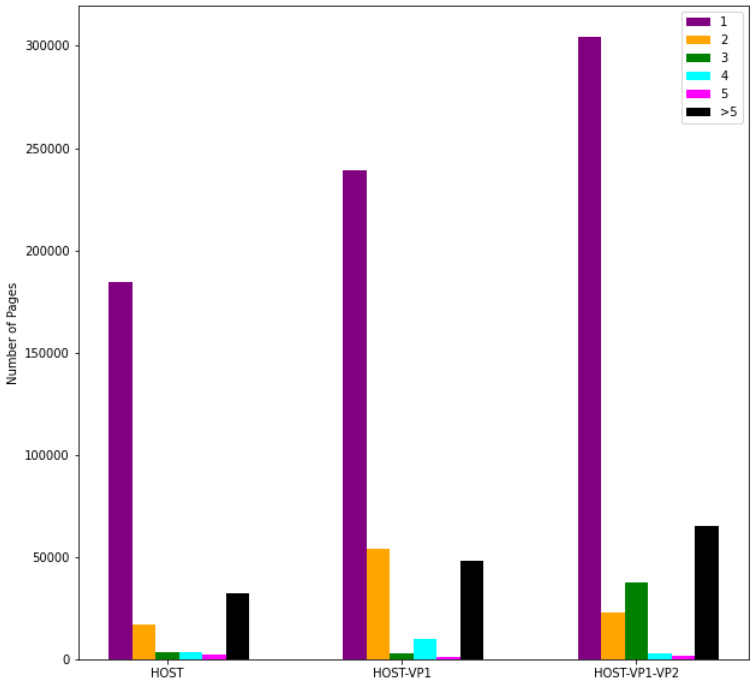

Figure 3.7: Summarized graph

3.7 Observations from Experiments

From the experiments conducted above, following points are noteworthy:

- Introducing a virtual phone needs around 350-400 MB of memory. From host to host+VP1, memory increased from 952MB to 1388MB and and from host+VP1 to host+VP1+VP2 memory increased from 1388MB to 1697MB. This is consistent with the results in the paper.
- Also, the host memory usage and VP's memory usage do not sum up to the total memory usage, indicating that there is sharing across phones.
- In the summarized graph, we look at physical pages which are uniquely mapped. The memory used by them increases by around 250-300 MB, again consistent with the paper.
- We found the top VMAs which have maximum number of unshared physical pages and their corresponding files. We saw some shared object files in this list. On looking at the namespaces these files belonged to, we found that these are almost equally distributed in host, VP1 and VP2. Example: '/system/lib64/libcrypto.so' had 399 physical pages unique to host and 304 pages unique to both VP1 and VP2.
- For KSM analysis, we looked at the PTE flags while translating a virtual address. We found the addresses which had VMA permissions as read-write and PTE flags as read. These might be the ones being handled by KSM. On looking at the file names corresponding to these addresses, we found that most of these mappings are anonymous (handled by KSM) or they lie in the same namespace (Copy on Write!).

Bibliography

[1] Android Open Source Code. https://source.android.com/.

[2] VPBox Source Code. https://github.com/VPBox/Dev, 2021.

[3] Jeremy Andrus, Christoffer Dall, Alexander Hof, Oren Laadan, and Jason Nieh. Cells: A virtual mobile smartphone architecture. pages 173–187, 10 2011.

[4] Wenna Song, Jiang Ming, Lin Jiang, Yi Xiang, Xuanchen Pan, Jianming Fu, and Guojun Peng. Towards transparent and stealthy android os sandboxing via customizable container-based virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2858–2874, New York, NY, USA, 2021. Association for Computing Machinery.