Para-Virtualization of Android Operating System on IMX6Q-SABRESD

August 2018

Contents

1 Introduction

Virtualization is technology that allows you to create multiple simulated environments or dedicated resources from a single, physical hardware system. Software called a hypervisor connects directly to that hardware and allows you to split 1 system into separate, distinct, and secure environments known as virtual machines (VMs). These VMs rely on the hypervisor's ability to separate the machine's resources from the hardware and distribute them appropriately. Virtualization helps you get the most value from previous investments.

Virtualization introduces a plethora of benefits to the working environment.

- When VMs and applications are properly isolated, only one application on one OS is affected by an attack.
- If a VM is infected, it can be rolled back to a prior "secure" state that existed before the attack.
- Hardware reductions that occur due to virtualization improve physical security since there are fewer devices and ultimately fewer data centers.

1.1 Objective

The aim is to run a virtualized flavour of Android Operating System on IMX6Q-SABRESD board. However, the board does not support hardware virtualization. To achieve the result, a para-virtualization approach was followed, wherein a software approach for hardware virtualization is taken. Paravirtualization seeks to bolster virtualization performance by allowing an OS to actually recognize the presence of a hypervisor and communicate directly with that hypervisor to share activity that would otherwise be complex and time-consuming for the hypervisor's VM manager to handle. Commands sent from the OS to the hypervisor are dubbed hypercalls. In order for paravirtualization to work, the guest VM OSes must be modified or adapted to implement an API capable of exchanging hypercalls with the paravirtualization hypervisor.

1.2 Design Prototype

The following image parades the rudimentary notion of a Unix-like Operating System. Once the bootloader verifies and powers up the hardware components on a motherboard, the kernel initializes them and loads a root file system, which starts the important processes, thereby loading the complete operating system for use.

```
 -------------------
| Operating System  |
 -------------------
| Root File System  |
 -------------------
|      Kernel       |
 -------------------
|    Bootloader     |
 -------------------
```

Generally, the kernel underlying any Android Operating System, is a linux kernel with some tweaks, giving the linux kernel the ability to handle Android applications. However, the basic unaltered linux kernel can mount a minimal Android root file system.

We will use U-Boot bootloader. Considering our virtualization requirements, we will use L4/Fiasco microkernel to act as the hypervisor. Atop the microkernel, L4Linux kernel is booted - which is a flavor of the linux kernel that can handle para-virtualization. L4Linux exchanges hypercalls with L4/Fiasco to behave as a Guest.

```
 -------------------
|   Android Root    |
|   File System     |
 -------------------
|   L4Linux Kernel  |
 -------------------
| L4/Fiasco Kernel  |
 -------------------
|    Bootloader     |
 -------------------
```

L4Linux kernel does not have the tweaks necessary to run a fully functional Android Operating System. We will use a minimal Android root file system that can be mounted on the L4Linux Kernel. To experience the complete Android OS, the para-virtualized linux kernel needs to undergo the necessary tweaks.

2 Tools Required

2.1 Hardware

- **NXP's IMX6Q-SABRESD** Board: Documentation about the board can be obtained from NXP's website
- USB Serial Cable
- Machine running Ubuntu 16.04 LTS
- EXT4 formatted SD-Card

2.2 Software

- U-Boot Bootloader
- L4 Runtime Environment source code
- L4/Fiasco Microkernel Source code
- L4Linux source code
- Android 7 Boot Image for IMX6Q-SABRESD board
- "gtkterm" or similar serial terminal
- Packages to be installed on Ubuntu:

  make, gawk, g++, binutils, pkg-config, g++-multilib, subversion, flex, bison

3 Procedure

The following steps need to be followed to get a para-virtualized Android Root File system.

- Setting up the bootloader
- Setting up L4 Runtime Environment
- Compiling L4/Fiasco Microkernel
- Compiling L4Linux Kernel
- Setting up Android Ramdisk
- Putting it all together

3.1 Setting up the bootloader

We will be using U-Boot. Das U-Boot (subtitled "the Universal Boot Loader" and often shortened to U-Boot) is an open source, primary boot loader used in embedded devices to package the instructions to boot the device's operating system kernel.

The source can be obtained from `https://github.com/u-boot/u-boot`. Since IMX6Q-SABRESD uses ARM Instruction Set, we need a cross-compiler as we are working on an Intel machine. It can be downloaded from ubuntu's repository. The cross-compiler used in here is `arm-linux-gnueabihf-gcc`. The following steps need to be followed to bring up u-boot on the device.

Now a u-boot directory exists with all the latest sources. Type the following commands in the Ubuntu terminal.

```
$ cd u-boot
$ export CROSS_COMPILE=arm-linux-gnueabihf-
$ make mx6qsabresd_config
```

U-Boot can be used to boot kernel images with have a maximum of 8MB. In our case, our virtualization requirements call for kernel images larger than 8MB in size. For u-boot to be able to handle such large image sizes, in the file at `u-boot-2015.10-rc1/common/bootm.c`, set the value of CONFIG_SYS_BOOTM_LEN as required. We will set it as 15MB.

```
#define CONFIG_SYS_BOOTM_LEN 0xFF0000
```

Now u-boot can be built.

```
$ make
```

This should create a number of files, including `u-boot.imx`. Now copy the bootloader to your SD Card. It should reside at offset 1024B on the SD Card. Use the following command for the same.

```
$ dd if=u-boot.imx of=/dev/<your-sd-card> bs=1k seek=1;sync
```

3.2 Setting up L4 Runtime Environment

For the next three sections, the source code can be obtained in one go using the following command. It grabs the required files from a subversion repository. Let the files be stored in the home directory.

```
$ svn cat http://svn.tudos.org/repos/oc/tudos/trunk/repomgr |
perl - init http://svn.tudos.org/repos/oc/tudos fiasco l4re
l4linux_requirements
```

This creates a directory called `/src`, with two subdirectories `/l4` and `/kernel`. The former containing the source for L4 runtime environment and the latter having the source for L4/Fiasco kernel.

Once the image is built, to be able to view messages in a serial terminal like gtkterm, some changes need to be done.

In the file at `/src/l4/pkg/bootstrap/server/src/platform/imx6.cc`, in line 78, part of the switch statement, note the value of 'case' where `kuart.base_address = 0x02020000`. In our case, it was found to be 1. So in file at `/src/l4/mk/platforms/imx6.conf`, set the following value

```
PLATFORM_UART_NR = 1
```

Create a build directory for the L4 runtime environment. In our case it is at /src/l4/B3. Navigate to this directory and setup initial configuration using,

```
$ make O=~/src/l4/B3 config
```

A configuration file can be found at ~/src/l4/B3/.KConfig

In the menu that appears, select target architecture as ARM, CPU variant as ARMv7A and platform as IMX6. Now the L4 runtime environment can be compiled using

```
$ make O=~/src/l4/B3
```

3.3 Compiling L4/Fiasco Microkernel

Once the L4 runtime environment has been setup, the L4/Fiasco kernel can be configured and compiled using the following steps.

```
$ make BUILDDIR=~/src/kernel/fiasco/FB3

$ cd ~/src/kernel/fiasco/FB3

$ make menuconfig
```

In the menu that appears, select target architecture as ARM, platform as Freescale IMX6 and CPU as ARM Cortex-A9. This generates a configuration file at ~/src/kernel/fiasco/FB3/globalconfig.out. Now the configuration can be compiled using

```
$ make
```

3.4 Compiling L4Linux Kernel

Obtain the source for L4linux using the following command and store in home directory.

```
$ svn co http://svn.tudos.org/repos/oc/l4linux/trunk l4linux
```

This creates a directory `~/l4linux`. Next we create a build directory, setup the configurations and compile L4Linux for IMX6Q-SABRESD using the following commands. Cross-compilation needs to be done.

```
$ mkdir  /l4linux/B3tux

$ make L4ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
O=~/l4linux/B3tux arm_defconfig

$ make L4ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
O=~/l4linux/B3tux menuconfig
```

In the menu that appears, set L4 build directory to `~/src/l4/B3`. Set the system type as ARM and build type as ARMv7. This generates a configuration file at `~/l4linux/B3tux/.config`

Now perform the compilation using,

```
$ make
```

This generates a file called `vmlinuz` in the L4Linux build directory. This is the kernel for paravirtualized linux.

Copy this file to `~/src/l4/B3/bin/arm_armv7a/l4f`

3.5 Setting up Android Ramdisk

RamDisk is a program that takes a portion of your system memory and uses it as a disk drive. We load a minimal Android root file system into this ramdisk. It can be set up as follows.

Get the prebuilt Android 7 boot image for IMX6-SABRESD from NXP's website. The boot image contains the kernel and the ramdisk. We need to extract the ramdisk from the boot image - `boot-imx6q.img`. A script is used for the same.

```
./split_bootimg.pl boot-imx6q.img
```

The script generates three files.

```
boot-imx6q.img-ramdisk.gz
boot-imx6q.img-kernel
boot-imx6q.img-second.gz
```

Among these files, `boot-imx6q.img-ramdisk.gz` is the ramdisk in compressed form. We create a new directory and uncompress the contents of this file into it as shown below.

```
gzip -dc ../boot-imx6q.img-ramdisk.gz | cpio -i
```

We will use some of the contents of the existing ramdisk to create our own ramdisk as follows.

Our ramdisk, ramdisk1-arm.rd, is a 4MB file formatted as ext4.

```
$ dd if=/dev/zero of=ramdisk1-arm.rd bs=1MiB count=4

$ mkfs.ext4 ./ramdisk1-arm.rd
```

Mount this ramdisk into a newly created directory to add files to it.

```
$ mkdir ./rdmnt

$ sudo mount -o loop ./ramdisk1-arm.rd ./rdmnt/
```

The newly created ramdisk would contain the following structure.

```
/bin -> /sbin
/sbin - busybox, ueventd->init, scripts that are linked to busybox
init
init.rc
default.prop
adbd
```

```
linuxrc->init
```

To statically compile busybox for Android on ARM, follow the steps as shown below.

```
$ export ARCH=arm

$ export CROSS_COMPILE=arm-linux-gnueabi-

$ make menuconfig
```

In the menu that appears, In 'Build Options' category, choose to build busybox as a static binary. In 'Module Utilities' category, unset the options - Default directory containing modules and Default name of modules.dep. It generates a configuration file as `/.config`. Now busybox is ready to be compiled.

```
$ make -j32
```

Once the compilation is successful, we will be using the generated binary `busybox`. Copy the files from the original ramdisk to our new ramdisk as per the above specified structure. Create links wherever necessary. We create a link from `linuxrc` to `init`, because the L4Linux kernel `vmlinuz` looks for an executable with the name `linuxrc` in the ramdisk.

```
$ sudo ln -s ./init linuxrc
```

The configuration of the `init.rc` file looks like this

```
    on early-init
start ueventd
on init
sysclktz 0
export PATH /bin:/sbin:
on boot
start adbd
service ueventd /sbin/ueventd
```

Modify `default.prop` to enable only adb and debugging. Once the necessary files have been copied into the new ramdisk, unmount it and copy it to ∼`/src/l4/B3/bin/arm_armv7a/l4f`

3.6 Putting it all together

Now we have all the required binaries to generate an image for a minimal Android root file system, running on a para-virtualized kernel. In the file `l4lx.cfg` at ∼`/src/l4/B3/bin/arm_armv7a/l4f`, modify the value of `l4x_rd` to match the name of our ramdisk and the `ramdisk_size` to 9000.

In file at ∼`/src/l4/conf/modules.list`, create an entry for our final image as follows.

```
   entry l4and
roottask moe rom/l4lx.cfg
module l4re
module ned
module l4lx.cfg
module io
module arm-rv.io
module vmlinuz
module ramdisk1-arm.rd
```

It specifies the binaries to be used to build the image. Now navigate to ∼`/src/l4/B3` and run the following command

```
make uimage E=l4and MODULE_SEARCH_PATH= /src/kernel/fiasco/FB3
```

The output file is generated at

∼`/src/l4/B3/bin/arm_armv7a/bootstrap_l4and.uimage`

This file can be loaded onto a SD-Card and can be used for booting the IMX6-SABRESD board.

3.7 Running Para-Virtualized Android Root File System on IMX6-SABRESD

Section 3.1 discussed about setting up the bootloader on the SD-Card. Now copy the `bootstrap_l4and.uimage` generated in the previous step to the SD-Card that has been formatted using EXT4 file system.

Connect the device to the Ubuntu machine using serial cables and open the desired serial terminal - gtkterm has been used here. In gtkterm, make sure the appropriate serial device has been selected, here `/dev/ttyUSB0` and the baud rate needs to be set to 1152000. Once this is done, insert the SD-Card into the device and boot it.

U-boot begins to load. Press any key to terminate auto-boot. The u-boot prompt should appear. Type the following command in the u-boot prompt, to boot the device from the SD-Card.

```
=> mmc dev 1
=> ext4load mmc 1:2 0x12000000 bootstrap_l4and.uimage
=> bootm 0x12000000
```

`0x12000000` is the load address of the kernel for this device. It is obtained from the processor manual. The `bootstrap_l4and.uimage` is loaded from the EXT4 formatted SD-Card to the device's memory. The command `bootm` boots the image that has been loaded into memory.

The L4 Bootstrapper initializes the L4/Fiasco microkernel, which brings up the L4Linux kernel, wherein the minimal Android root file system is mounted.

Here the kernel image has been booted from memory. Once the final required image has been obtained, the image can be flashed to the `emmc` chip of the device, to boot from it.

The following images, demonstrate the output obtained.

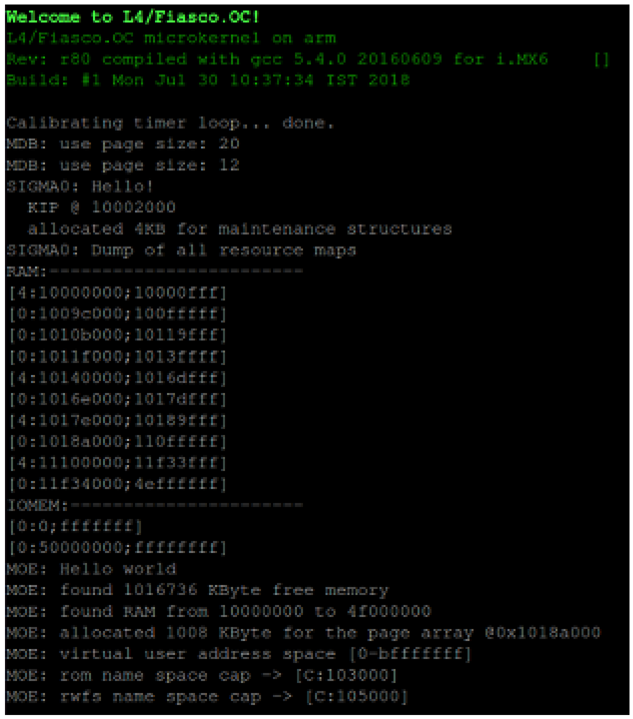
Let us analyze the first few bytes of `init`. As seen in Figure 6, we can see that this file is indeed one built for Android.

```
=> bootm 0x12000000
## Booting kernel from Legacy Image at 12000000 ...
   Image Name:   L4 Image #26
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    15758764 Bytes = 15 MiB
   Load Address: 11000000
   Entry Point:  11000000
   Verifying Checksum ... OK
   Loading Kernel Image ... OK

Starting kernel ...


L4 Bootstrapper
  Build: #26 Tue Aug 14 12:16:14 IST 2018, 5.4.0 20160609
  Scanning up to 1024 MB RAM, starting at offset 32MB
  Memory size is 1024MB (10000000 - 4fffffff)
  RAM: 0000000010000000 - 000000004fffffff: 1048576kB
  Total RAM: 1024MB
  Scanning fiasco
  Scanning sigma0
  Scanning moe
  Moving up to 10 modules behind 11100000
  moving module 02 { 11ed9000-11f075ab } -> { 11fca000-11ff85ab } [189868]
  moving module 01 { 11ece000-11ed854b } -> { 11fbf000-11fc954b } [42316]
  moving module 00 { 11e43000-11ecda77 } -> { 11f34000-11fbea77 } [567928]
  moving module 09 { 11643000-11e42fff } -> { 11734000-11f33fff } [8388608]
  moving module 08 { 11226000-1164270f } -> { 11317000-1173370f } [4310800]
  moving module 07 { 11225000-112250e7 } -> { 11316000-113160e7 } [232]
  moving module 06 { 11083000-11224dc3 } -> { 11174000-11315dc3 } [1711556]
  moving module 05 { 11082000-1108212d } -> { 11173000-1117312d } [302]
  moving module 04 { 11026000-11081e27 } -> { 11117000-11172e27 } [376360]
  moving module 03 { 1100f000-1102551f } -> { 11100000-1111651f } [91424]
  Loading fiasco
  Loading sigma0
  Loading moe
  find kernel info page...
```

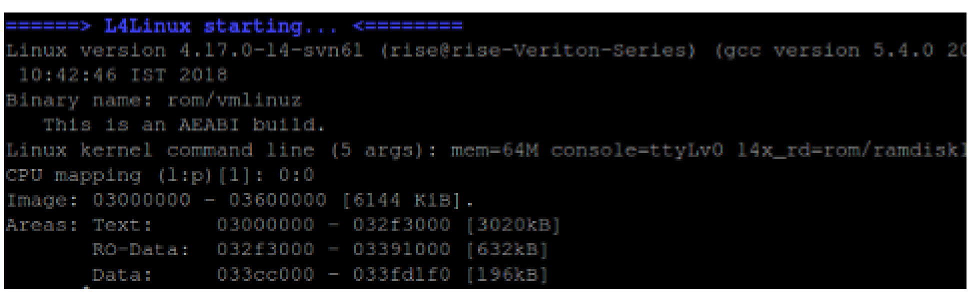
Figure 1: L4 Bootstrapping



```
Welcome to L4/Fiasco.OC!
L4/Fiasco.OC microkernel on arm
Rev: r80 compiled with gcc 5.4.0 20160609 for i.MX6    []
Build: #1 Mon Jul 30 10:37:34 IST 2018

Calibrating timer loop... done.
MDB: use page size: 20
MDB: use page size: 12
SIGMA0: Hello!
  KIP @ 10002000
  allocated 4KB for maintenance structures
SIGMA0: Dump of all resource maps
RAM:------------------------
[4:10000000;10000fff]
[0:1009c000;100fffff]
[0:1010b000;10119fff]
[0:1011f000;1013ffff]
[4:10140000;1016dfff]
[0:1016e000;1017dfff]
[4:1017e000;10189fff]
[0:1018a000;110fffff]
[4:11100000;11f33fff]
[0:11f34000;4effffff]
IOMEM:----------------------
[0:0;fffffff]
[0:50000000;ffffffff]
MOE: Hello world
MOE: found 1016736 KByte free memory
MOE: found RAM from 10000000 to 4f000000
MOE: allocated 1008 KByte for the page array @0x1018a000
MOE: virtual user address space [0-bfffffff]
MOE: rom name space cap -> [C:103000]
MOE: rwfs name space cap -> [C:105000]
```

Figure 2: L4/Fiasco Kernel started



```
======> L4Linux starting... <========
Linux version 4.17.0-l4-svn61 (rise@rise-Veriton-Series) (gcc version 5.4.0 20
 10:42:46 IST 2018
Binary name: rom/vmlinuz
   This is an AEABI build.
Linux kernel command line (5 args): mem=64M console=ttyLv0 l4x_rd=rom/ramdiskl
CPU mapping (l:p)[1]: 0:0
Image: 03000000 - 03600000 [6144 KiB].
Areas: Text:     03000000 - 032f3000 [3020kB]
       RO-Data:  032f3000 - 03391000 [632kB]
       Data:     033cc000 - 033fd1f0 [196kB]
```

Figure 3: L4Linux started

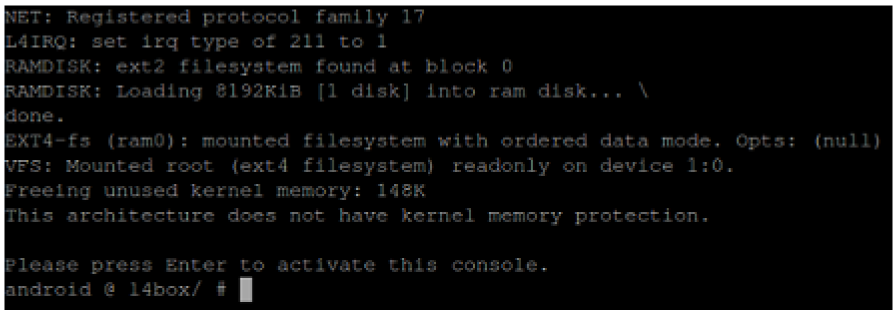


Figure 4: Mounted minimal root file system

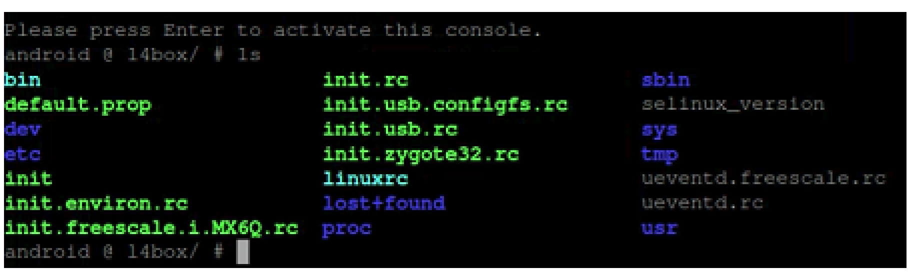


Figure 5: Viewing files in root file system

Figure 6: HEX Bytes of Android init

4 Future Work

The L4Linux kernel can be modified further to be able to boot Android Operating System completely.