



POLITECNICO DI TORINO

DIPARTIMENTO DI ELETTRONICA E TELECOMUNICAZIONI

ACADEMIC YEAR 2023/2024

Operating systems

Laboratory experiences

Authors:

Bonino Andrea - 314709
Fasolis Gabriele - 314903
Mattiauda Mattia - 315096
Mondino Ettore - 317755
Tartaglia Federico - 319857

Professor:

Stefano Di Carlo

October 25, 2023

Simulated Device Driver © 2023 by Andrea Bonino, Gabriele Fasolis, Mattia Mattiauda, Ettore Mondino, Federico Tartaglia is licensed under Attribution-NonCommercial 4.0 International. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/4.0/>



Contents

I	LAB 1 - Setup of the Working Environment	2
1	Introduction	2
2	Lab Goal	3
3	Yocto Project	3
3.1	Install packages	3
3.2	Download and install Poky	3
4	QEMU	4
4.1	Compile QEMU	4
4.2	Run QEMU manually	5
4.3	Run QEMU in automatic with script	6
4.4	Add a device to QEMU	7
4.4.1	Files to modify in order to add a device	7
4.4.2	Files to be modified in order to compile QEMU with the new device	9
5	Layer and Recipes	10
II	LAB 2 - Cryptocore Test and Library Creation	13
1	Attributes test	13
2	Library description	15
3	Library test	16
III	LAB 3 - Application Program with SHA-256	18
1	Program specification	18
2	About SHA-256	19

LAB 1 - Setup of the Working Environment

1 Introduction

In this lab experience you will set up the environment needed for the next laboratories. Specifically, you will get acquainted with the software required to simulate an **ARM machine** running on your PC.

The reference operating system suggested for these lab experiences is **Ubuntu 22.04.2 LTS**. If you do not have access to a PC with an Ubuntu operating system installed (e.g. a PC with dual boot) the best option for you is to create a virtual machine (VM). Since the working environment setup will require a lot of disk space and the installation procedure will take a lot of time, you are suggested to follow figure 1 during the creation of the VM (we used VirtualBox hypervisor to host the guest machines).

Warning: if you do not reserve enough disk space (e.g. 90 GB will suffice) for your VM, the Poky download and installation from section 3.2 will fail and you may have to create a new VM from scratch. Moreover, try to reserve as much memory and processors as your machine can afford, so that the Poky installation will require less time: even if you reserve 5 MB and 5 processors on a standard PC this operation will take some hours to accomplish.

When your VM is up and running, try to execute the following command in your terminal, so that you can double check your operating system version:

```
vboxuser@Ubuntu-progettoDriverOS:/$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.2 LTS
Release:       22.04
Codename:      jammy
```

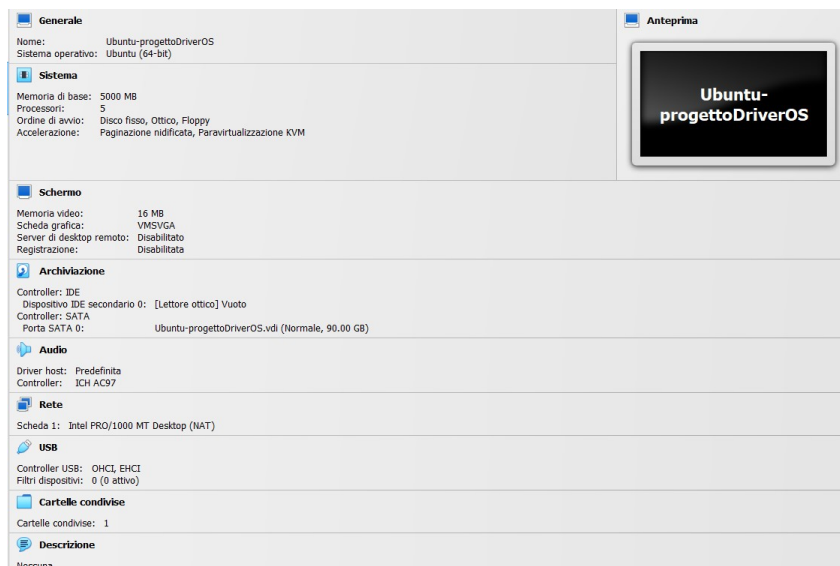


Figure 1: Features of the Virtual Machine

2 Lab Goal

The target of the experience is to run the ARM machine and connect a fully **simulated device**. The simulated ARM machine will run a Linux operating system (without Graphic User Interface) and the device connected will be managed by the OS through a **device driver**.

The simulated machine:

The ARM machine is simulated using **QEMU**, a software which emulates a board and lets you connect different devices. The ARM board chosen is called **QEMU virt board**, a platform which does not correspond to any real hardware and it is designed for use in virtual machines. This is the recommended board type if you simply want to run a guest such as Linux and do not care about reproducing the idiosyncrasies and limitations of a particular bit of real-world hardware.

The simulated device:

The simulated device encrypts (or better *hashes*) a string inputted by the ARM board and then returns it to the board. Such a device will be referred to as a “**cryptocore**”.

QEMU lets you create the device you want to connect to the board through a behavioral description, so the cryptocore description should include how the input string is encrypted, that is with the **SHA-256** cryptographic hash function.

The device driver manages the input and output (hashed) of the cryptocore and it will be tested at the beginning of the next lab experience.

In order to generate the operating system image for the ARM board and set up the QEMU simulation you first have to download **Poky**. Poky is part of the **Yocto Project**, an open source collaboration project that helps developers create custom Linux-based systems regardless of the hardware architecture. The emulator software QEMU is installed on your machine as part of cloning the Poky repository and sourcing the environment script. The following sections explain in a step by step tutorial how to get the Poky git repository, which contains the source code from which the custom Linux distribution (i.e. bootloader, kernel, root file system containing system and user programs) is built.

The step by step tutorial will guide you through the Poky installation and will help you to get acquainted to QEMU by showing you how to emulate a new board and connect the cryptocore device. The tutorial will then explain you how to bind the device to a device driver.

3 Yocto Project

3.1 Install packages

Before the Poky installation you need to install the following packages:

```
sudo apt-get install gawk wget git diffstat unzip texinfo gcc-multilib build-essential
sudo apt-get install chrpath socat cpio libSDL1.2-dev xterm ncurses-dev lzop
sudo apt-get install minicom u-boot-tools curl net-tools}
```

(if needed exec the command `ps` to see python pid and then `kill pid`)

```
sudo apt-get install python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping
➔ python3-git python3-jinja2 libegl-mesa python3-subunit zstd liblz4-tool
```

3.2 Download and install Poky

Download Poky source repository from git. Execute this command inside the folder where you want the Poky repository to be cloned:

```
git clone https://git.yoctoproject.org/poky
cd poky
```

Select Dunfell branch:

```
git checkout -b dunfell origin/dunfell
git pull
```

Initialize the build environment for an ARM machine QEMU target:

```
source oe-init-build-env build_qemuarm
```

Your current working directory should now be `build_qemuarm` which is your build directory and where the images for your target are built. As the output from running the command above indicates, you will now need to select the target hardware `MACHINE` in the `conf/local.conf` file. In order to do so, modify the file as shown in figure 2, you just need to uncomment the line:

```
MACHINE ?= "qemuarm"
```

Build the system. **Warning:** the first time you execute this command it could take more than 3 hours (depending on your internet connection and PC features):

```
bitbake core-image-minimal
```

Now you can start the QEMU simulation:

```
runqemu qemuarm nographic
```

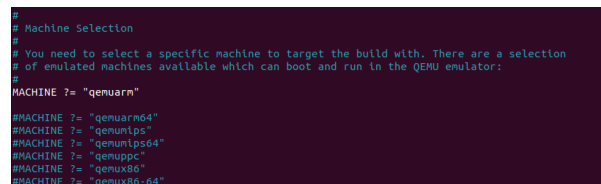
This last command should run the ARM machine and you should see Linux booting on your console.

Enter “root” when presented in the login prompt. Once inside the running QEMU emulation you should see a different prompt symbol right before your cursor (see figure 9, where a “#” symbol is shown instead of the usual “\$” Linux prompt symbol).

Run `uname -a` to check the Linux distribution running and the target hardware architecture. In order to exit the QEMU simulation you have to press “Ctrl + A”, then release and press “X”.

```
root@qemuarm:/# uname -a
Linux qemuarm 5.4.257-yocto-standard #1 SMP PREEMPT Tue Sep 26 14:42:26
↪ UTC 2023 armv7l GNU/Linux
```

NOTE: If internet connection is lost during the installation procedure the executing command (`bitbake`) will fail. If this happens or any other error occurs before the `bitbake` command is completed, try to rerun the command: it will maintain the progress made until the error occurred and try again.



```
#
# Machine Selection
#
# You need to select a specific machine to target the build with. There are a selection
# of emulated machines available which can boot and run in the QEMU emulator:
#
MACHINE ?= "qemuarm"
#MACHINE ?= "qemuarm64"
#MACHINE ?= "qemuntips"
#MACHINE ?= "qemuntips64"
#MACHINE ?= "qemuppc"
#MACHINE ?= "qemu86"
#MACHINE ?= "qemu86-64"
```

Figure 2: `local.conf` modified

4 QEMU

4.1 Compile QEMU

Instead of running `runqemu qemuarm nographic`, which runs the default version of QEMU you can run a specific version of QEMU and choose a specific image to be run. QEMU is installed on your machine as part of cloning the Poky repository and sourcing the environment script, as explained in section 3.2.

Before starting to add any new devices connected to the ARM machine (e.g. in section 4.4 will be explained how to add the SHA-256 cryptcore), you should try to compile QEMU. This operation must be done every time you modify files inside the QEMU directory, otherwise the changes will not be applied.

In order to compile QEMU you need to download and install the following packets:

```
sudo apt-get update -y
sudo apt-get install -y libgcrypt-dev
sudo apt install flex
sudo apt install bison
```

- The “libgcrypt” library is a low-level cryptographic library that provides various cryptographic algorithms and functions for secure data processing and encryption.
- “flex” refers to the GNU Flex (Fast Lexical Analyzer Generator), a tool for generating lexical analyzers.
- “bison” refers to the GNU Bison, a parser generator tool that is used for generating parsers and compilers.

NOTE: If an error such as `unable to disambiguate ...` occurs, it is possible to fix it by commenting the lines 2050 2051 2052 2053 of the `configure` file at the path:

```
/home/mattia/Desktop/progetto/poky/build_qemuarm/tmp/work/x86_64-linux/qemu-system-native
↪ /4.2.0-r0/qemu-4.2.0
```

To launch the compile process you have to execute the `./configure` script and then the `make` command at the path:

```
/home/mattia/Desktop/progetto/poky/build_qemuarm/tmp/work/x86_64-linux/qemu-system-native
↪ /4.2.0-r0/qemu-4.2.0
```

The `configure` script (it takes about 20 minutes to execute) has to be executed right before the `make` command.

When QEMU will have to be recompiled (see section 4.4) it will not be needed to run again the `configure` script. In fact, this script is responsible for getting ready to build the software on your specific system. It makes sure that all the dependencies for the rest of the build and install process are available, and finds out whatever it needs to know to use those dependencies (the `make` command now is able to do what is described inside the Makefile).

4.2 Run QEMU manually

After following the instruction of section 4.1, you should be able to compile QEMU. Now you will see how to boot the whole ARM system.

Execute the command:

```
source oe-init-build-env build_qemuarm
```

This command changes the working directory to the Source Directory and runs the Yocto Project “oe-init-build-env environment” setup script. Running this script defines the OpenEmbedded build environment settings needed to complete the build. The script also creates the Build Directory, which is `build_qemuarm` in this case and is located in the Source Directory. After the script runs, your current working directory is set to the Build Directory. Later, when the build completes, the Build Directory contains all the files created during the build.

Execute this command to run QEMU:

```
sudo <YOUR PATH TO POKY DIRECTORY>/poky/build_qemuarm/tmp/work/x86_64-linux/qemu-system-
↪ native/4.2.0-r0/qemu-4.2.0/arm-softmmu/qemu-system-arm -device virtio-net-device,
↪ netdev=net0,mac=52:54:00:12:34:02 -netdev tap,id=net0,ifname=tap0,script=no,
↪ downscript=no -drive id=disk0,file=<YOUR PATH TO POKY DIRECTORY>/poky/build_qemuarm
↪ /tmp/deploy/images/qemuarm/<IMAGE .ext4>,if=none,format=raw -device virtio-blk-
↪ device,drive=disk0 -show-cursor -device VGA,edid=on -device qemu-xhci -device usb-
↪ tablet -device usb-kbd -object rng-random,filename=/dev/urandom,id=rng0 -device
↪ virtio-rng-pci,rng=rng0 -nographic -machine virt,highmem=off -cpu cortex-a15 -m 256
↪ -serial mon:stdio -serial null -kernel <YOUR PATH TO POKY DIRECTORY>/poky/
↪ build_qemuarm/tmp/deploy/images/qemuarm/zImage -append 'root=/dev/vda rw console=
↪ ttyS0 mem=256M ip=192.168.7.2::192.168.7.1:255.255.255.0 console=ttyAMA0 '
```

Explanation of the long command above:

Inside the folder `poky` there is a QEMU version installed at the path:

```
poky/build_qemuarm/tmp/work/x86_64-linux/qemu-system-native/4.2.0-r0/qemu-4.2.0/arm-
↪ softmmu
```

Here you can find the executable file `qemu-system-arm` that is called in the first part of the command.

If a new image is built (through *bitbake* as explained in the Poky installation section) you have to replace the old `<FILE SYSTEM.ext4>` inside the long command above. You have to substitute the old file system path with the absolute path of the more recent one.

Both the old and new file system are stored in a directory which path is:

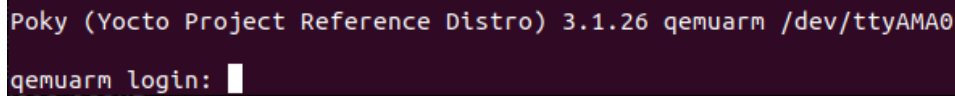
```
/home/mattia/Desktop/progetto/poky/build_qemuarm/tmp/deploy/images/qemuarm
```

A new version of the `<FILE SYSTEM.ext4>` is created when you launch the command `bitbake core-image-minimal` every time (and only if) there are changes to be applied to the file system.

For example, every time you change the file system by adding a layer or by modifying the recipes inside a layer you have to run again the command `bitbake core-image-minimal` to create an updated version of the file system.

NOTE: An operating system image is a complete snapshot of an operating system that includes the core operating system components, while a file system is a method of organizing and storing files and directories on storage media within an OS. The file system is just one component of the larger operating system image.

If everything went ok, at the end of the procedure it will appear the log-in prompt as shown in figure 3. Enter the user name `root` to log into the system as the Linux superuser.



```
Poky (Yocto Project Reference Distro) 3.1.26 qemuarm /dev/ttyAMA0
qemuarm login: 
```

Figure 3: log-in prompt

4.3 Run QEMU in automatic with script

In order to speed-up the procedure we wrote the script `bitbake_run_auto.sh` that does the following operations:

- move to `poky` directory (line 3)
- run the command `source oe-init-build-env build_qemuarm` (line 8)
- run the command `bitbake core-image-minimal` (as a consequence it generates a new file system if any change is made) (line 15)
- find the most recent file system with extension `.ext4` (line 22)
- run QEMU using the last file system created (line 25).

NOTE: inside the script there are different paths; remember to modify them according to your files directories.

```
1 #!/bin/bash
2
3 cd ~/Desktop/progetto/poky/
4 echo ""
5 echo "*****"
6 echo "SOURCE oe-init-build-env"
7 echo "*****"
8 source oe-init-build-env build_qemuarm
9
10
11 echo ""
12 echo "*****"
13 echo "EXEC bitbake"
14 echo "*****"
15 bitbake core-image-minimal
16
17 echo ""
18 echo "*****"
19 echo "EXEC command_long"
20 echo "*****"
21
22 root_fs=$(ls /home/mattia/Desktop/progetto/poky/build_qemuarm/tmp/deploy/images/qemuarm | grep
    ↪ "rootfs.ext4")
```

```

23
24
25 exec sudo /home/mattia/Desktop/progetto/poky/build_qemuarm/tmp/work/x86_64-linux/qemu-system-
    ↪ native/4.2.0-r0/qemu-4.2.0/arm-softmmu/qemu-system-arm -device virtio-net-device,
    ↪ netdev=net0,mac=52:54:00:12:34:02 -netdev tap,id=net0,ifname=tap0,script=no,
    ↪ downscript=no -drive id=disk0,file=/home/mattia/Desktop/progetto/poky/build_qemuarm
    ↪ /tmp/deploy/images/qemuarm/$root_fs,if=none,format=raw -device virtio-blk-device,
    ↪ drive=disk0 -show-cursor -device VGA,edid=on -device qemu-xhci -device usb-tablet -
    ↪ device usb-kbd -object rng-random,filename=/dev/urandom,id=rng0 -device virtio-rng-
    ↪ pci,rng=rng0 -nographic -machine virt,highmem=off -cpu cortex-a15 -m 256 -serial
    ↪ mon:stdio -serial null -kernel /home/mattia/Desktop/progetto/poky/build_qemuarm/tmp
    ↪ /deploy/images/qemuarm/zImage -append 'root=/dev/vda rw console=ttyS0 mem=256M ip
    ↪ =192.168.7.2::192.168.7.1:255.255.255.0 console=ttyAMA0 '

```

Run this script executing the following command (you have to be in the same folder where the script is saved):

```
./bitbake_run_auto.sh
```

At the end of the script you should see the log-in prompt as in figure 3. Enter `root` to log into the system as Linux superuser.

4.4 Add a device to QEMU

4.4.1 Files to modify in order to add a device

There are 3 files you need to modify/create; in the second line of each element of the bulleted list there is the path where you can find the file if it has to be modified or where you can place the new file if it has to be created:

- `virt.c` (to be modified)

```

/home/mattia/Desktop/progetto/poky/build_qemuarm/tmp/work/x86_64-linux/qemu-system-
    ↪ native/4.2.0-r0/qemu-4.2.0/hw/arm

```

- Map the device (cryptocore) into the memory of the 32 bit simulated ARM board giving it the necessary space
- Map the interrupt on an available line
- Describe the function `create_virt_sha_device`

This file helps in the emulation of a virtual board, it passes Linux all the information it needs about which devices are present via the device tree. A device tree in Linux is a data structure that describes the hardware components and their configuration in a platform-independent and machine-readable format. It allows the Linux kernel to adapt to different hardware configurations without recompilation. Device trees are commonly used in embedded systems and are loaded during the boot process to configure hardware and drivers.

By mapping the memory we reserve a portion of memory to our device. There is a specific portion of memory dedicated to devices. As shown in figure 6a:

- 0..128 MB is the space dedicated for a flash device so we can run bootrom code such as UEFI.
- 128 MB..256 MB is used for miscellaneous device I/O.
- 256 MB..1 GB is reserved for possible future PCI support (i.e. where the PCI memory window will go if we add a PCI host controller).
- 1GB and up is RAM (which may happily spill over into the high memory region beyond 4 GB). This represents a compromise between how much RAM can be given to a 32 bit VM and leaving space for expansion and in particular for PCI.

The function `create_virt_sha_device` manages the links between our device, the chip and the driver. As you can see in picture 4 there are 3 different names you have to choose:

- `virt-sha-device`: this name links the device with the board.
- `virt-sha-driver`: this name links the device with its driver.
- `virt-sha-nodename`: this is just the name of our device mounted on the board.


```

static void create_virt_sha_device(const VirtMachineState *vms, qemu_irq *pic)
{
    hwaddr base = vms->memmap[VIRT_SHA].base;
    hwaddr size = vms->memmap[VIRT_SHA].size;
    int irq = vms->irqmap[VIRT_SHA];
    char *nodename;

    /*
     * virt-foo@0b000000 {
     *     compatible = "virt-foo";
     *     reg = <0x0b000000 0x200>;
     *     interrupt-parent = <0>;
     *     interrupts = <176>;
     * }
     */

    sysbus_create_simple("virt-sha-device", base, pic[irq]); //legame con virt_sha.c dove descrivo il device

    nodename = g_strdup_printf("/virt-sha-nodename%" PRIx64, base);
    qemu_fdt_add_subnode(vms->fdt, nodename);
    qemu_fdt_setprop_string(vms->fdt, nodename, "compatible", "virt-sha-driver"); //legame con virt-sha.c in meta-example
    qemu_fdt_setprop_size_cells(vms->fdt, nodename, "reg", 2, base, 2, size);
    qemu_fdt_setprop_cells(vms->fdt, nodename, "interrupt-parent",
                           vms->gic_phandle);
    qemu_fdt_setprop_cells(vms->fdt, nodename, "interrupts",
                           GIC_FDT_IRQ_TYPE_SPI, irq,
                           GIC_FDT_IRQ_FLAGS_LEVEL_HI);

    g_free(nodename);
}

```

Figure 4: virt.c modified

The three strings containing the names described above constitute links between different files which describe the board, the device and the driver. Those names are present in both the files that have to be recognised as bounded (e.g. “virt-sha-device” is present both in “virt_sha.c” and in “virt.c”, which describe the device and the board respectively).

As shown in figure 6b, the virt.c file contains also the mapping of the device interrupts.

- virt.h (to be modified)

```

/home/mattia/Desktop/progetto/poky/build_qemuarm/tmp/work/x86_64-linux/qemu-system-
  ↪ native/4.2.0-r0/qemu-4.2.0/include/hw/arm

```

- Adda the cryptocore device to the list of devices

In this file we add to an enum list the device we want to create as shown in figure 5.

```

#define ARCH_F10K_IRQ_0_IRQ 10

#define VIRTUAL_PMU_IRQ 7

#define PPI(irq) ((irq) + 16)

enum {
    VIRT_FLASH,
    VIRT_MEM,
    VIRT_CPUPERIPHERS,
    VIRT_GIC_DIST,
    VIRT_GIC_CPU,
    VIRT_GIC_V2M,
    VIRT_GIC_HYP,
    VIRT_GIC_VCPU,
    VIRT_GIC ITS,
    VIRT_GIC_REDIST,
    VIRT_SMMU,
    VIRT_UART,
    VIRT_MMIO,
    VIRT_RTC,
    VIRT_FW_CFG,
    VIRT_PCIE,
    VIRT_PCIE_MMIO,
    VIRT_PCIE_PIO,
    VIRT_PCIE_ECAM,
    VIRT_PLATFORM_BUS,
    VIRT_GPIO,
    VIRT_SECURE_UART,
    VIRT_SECURE_MEM,
    VIRT_SHA,
    VIRT_PCDIMM_ACPI,
    VIRT_ACPI_GED,
    VIRT_LOWMEMMAP_LAST,
};

/* indices of IO regions located after
enum {

```

Figure 5: virt.h modified

- virt.sha.c (to be created)

```
/home/mattia/Desktop/progetto/poky/build_qemuarm/tmp/work/x86_64-linux/qemu-system-
  ↳ native/4.2.0-r0/qemu-4.2.0/hw/misc
```

- SHA256 algorithm
- read function virt_sha_read
- write function virt_sha_write

In this file we defined the device we want to add: virt.sha. We implemented the algorithm necessary to execute the SHA256 (or better *hashing*). Inside this file we described also two functions: virt_sha_write and virt_sha_read. In these functions we pass the value of the offset where we will write the data with respect to the base address assigned to the device. In the function virt_sha_write we pass the value we want to write as well (it is a 64 bit value). When dealing with strings we have to recall the virt_sha_write function for each character we want to write and the virt_sha_read function for each time we want to read a character.

After the description of these two new functions there is the initialization of the device.

```
/* to accommodate guests using 64K pages.
 */
static const MemMapEntry base_memmap[] = {
/* Space up to 0x00000000 is reserved for a boot ROM */
[VIRT_FLASH] = { 0, 0x00000000 },
[VIRT_CPU_PERIPHERALS] = { 0x00000000, 0x00020000 },
/* GIC distribution and CPU interface is in the CPU peripheral space */
[VIRT_GIC_DIST] = { 0x00000000, 0x00010000 },
[VIRT_GIC_CPU] = { 0x00010000, 0x00010000 },
[VIRT_GIC_VPM] = { 0x00020000, 0x00001000 },
[VIRT_GIC_HYP] = { 0x00030000, 0x00010000 },
[VIRT_GIC_VPM] = { 0x00040000, 0x00010000 },
/* The space in between here is reserved for GIC CPU/VPM */
[VIRT_GIC_VPM] = { 0x00080000, 0x00020000 },
/* This redistributor space allows up to 256K*128 CPUs */
[VIRT_GIC_REDIST] = { 0x000A0000, 0x00F00000 },
[VIRT_UART] = { 0x00000000, 0x00001000 },
[VIRT_RTC] = { 0x00010000, 0x00001000 },
[VIRT_FW_CFG] = { 0x00020000, 0x00000018 },
[VIRT_GPIO] = { 0x00030000, 0x00001000 },
[VIRT_SECURE_UART] = { 0x00040000, 0x00001000 },
[VIRT_SMMU] = { 0x00050000, 0x00020000 },
[VIRT_PCIE_DMI] = { 0x00070000, MEMORY_HOTPLUG_LEN },
[VIRT_ACPI_GED] = { 0x00080000, ACPI_GED_EVT_SEL_LEN },
[VIRT_MMIO] = { 0x00080000, 0x00000200 },
[VIRT_SHA] = { 0x00040000, 0x00000200 },
/* ...repeating for a total of NUM_VIRTIO_TRANSPORTS, each of that size */
[VIRT_PLATFORM_BUS] = { 0x00000000, 0x02000000 },
[VIRT_SECURE_MCA] = { 0x00000000, 0x01000000 },
[VIRT_PCIE_MMIO] = { 0x10000000, 0x2eff0000 },
[VIRT_SMMU] = { 0x3eff0000, 0x00010000 },
[VIRT_PCIE_ECAM] = { 0x3f000000, 0x01000000 },
/* Actual ROM size depends on host's RAM and device memory settings */
[VIRT_MEM] = { 0, LEGACY_RAMINIT_BYTES },
};
```

(a) Memory mapping modified

```
};
static const int a15irqmap[] = {
[VIRT_UART] = 1,
[VIRT_RTC] = 2,
[VIRT_PCIE] = 3, /* ... to 6 */
[VIRT_GPIO] = 7,
[VIRT_SECURE_UART] = 8,
[VIRT_ACPI_GED] = 9,
[VIRT_MMIO] = 16, /* ...to 16 + NUM_VIRTIO_TRANSPORTS - 1 */
[VIRT_GIC_VPM] = 48, /* ...to 48 + NUM_GICV2M_SPIS - 1 */
[VIRT_SMMU] = 74, /* ...to 74 + NUM_SMMU_IRQS - 1 */
[VIRT_PLATFORM_BUS] = 112, /* ...to 112 + PLATFORM_BUS_NUM_IRQS - 1 */
[VIRT_SHA] = 176,
};
static const char *valid_cpus[] = {
ARM_CPU_TYPE_NAME("cortex-a7"),
ARM_CPU_TYPE_NAME("cortex-a15"),
};
```

(b) Interrupt mapping modified

Figure 6: virt.c modified

4.4.2 Files to be modified in order to compile QEMU with the new device

There are 4 files you need to modify in order to compile QEMU with the new device:

- Kconfig

Modify this file as shown in figure 7a. Kconfig path is:

```
/home/mattia/Desktop/progetto/poky/build_qemuarm/tmp/work/x86_64-linux/qemu-system-
  ↳ native/4.2.0-r0/qemu-4.2.0/hw/misc
```

- Makefile.objs

Modify this file as shown in figure 7b. Makefile.objs path is:

```
/home/mattia/Desktop/progetto/poky/build_qemuarm/tmp/work/x86_64-linux/qemu-system-
  ↳ native/4.2.0-r0/qemu-4.2.0/hw/misc
```

In this file we specify the compiled file virt.sha.o

- config-devices.mak

Modify this file as shown in figure 7c. config-devices.mak path is:

```
/home/mattia/Desktop/progetto/poky/build_qemuarm/tmp/work/x86_64-linux/qemu-system-
↳ native/4.2.0-r0/qemu-4.2.0/arm-softmmu
```

- arm-softmmu.mak

Modify this file as shown in figure 7d. arm-softmmu.mak path is:

```
/home/mattia/Desktop/progetto/poky/build_qemuarm/tmp/work/x86_64-linux/qemu-system-
↳ native/4.2.0-r0/qemu-4.2.0/default-configs
```

```
config VIRT_SHA
bool

config APPLESMC
bool
depends on ISA_BUS

config MAX111X
bool

config TMP105
bool
depends on I2C

config TMP421
bool
depends on I2C

config ISA_DEBUG
bool
depends on ISA_BUS

config SGA
bool
depends on ISA_BUS
```

(a) Kconfig modified

```
common-obj-$(CONFIG_VIRT_SHA) += virt_sha.o
common-obj-$(CONFIG_APPLESMC) += applesmc.o
common-obj-$(CONFIG_MAX111X) += max111x.o
common-obj-$(CONFIG_TMP105) += tmp105.o
common-obj-$(CONFIG_TMP421) += tmp421.o
common-obj-$(CONFIG_ISA_DEBUG) += debugexit.o
common-obj-$(CONFIG_SGA) += sga.o
common-obj-$(CONFIG_ISA_TESTDEV) += pc-testdev.o
common-obj-$(CONFIG_PCI_TESTDEV) += pci-testdev.o
common-obj-$(CONFIG_EDU) += edu.o
common-obj-$(CONFIG_PCA9552) += pca9552.o

common-obj-$(CONFIG_UNIMP) += unimp.o
common-obj-$(CONFIG_FW_CFG_OPA) += vmcoreinfo.o

# ARM devices
common-obj-$(CONFIG_PL310) += arm_l2x0.o
```

(b) Makefile.objs modified

```
CONFIG_VIRT_SHA=y
CONFIG_A15MPCORE=y
CONFIG_A9MPCORE=y
CONFIG_A9SCU=y
CONFIG_A9_GTIMER=y
CONFIG_AC97=y
CONFIG_ACPI=y
CONFIG_ACPI_HW_REDUCED=y
CONFIG_ACPI_MEMORY_HOTPLUG=y
CONFIG_ACPI_PCI=y
CONFIG_ADS7846=y
CONFIG_AHCI=y
CONFIG_AHCI_ICH9=y
CONFIG_ALLWINNER_A10=y
CONFIG_ALLWINNER_A10_PIC=y
CONFIG_ALLWINNER_A10_PIT=y
CONFIG_ALLWINNER_EMAC=y
CONFIG_ARM11MPCORE=y
```

(c) config-devices.mak

```
# Default configuration for arm-softmmu

# TODO: ARM_V7M is currently always required - make this more flexible!
CONFIG_ARM_V7M=y

# CONFIG_PCI_DEVICES=n
# CONFIG_TEST_DEVICES=n

CONFIG_VIRT_SHA=y
CONFIG_ARM_VIRT=y
CONFIG_CUBIEBOARD=y
CONFIG_EXYNOS4=y
CONFIG_HIGHBANK=y
CONFIG_INTEGRATOR=y
CONFIG_FSL_IMX31=y
CONFIG_MUSICAL=y
CONFIG_MUSCA=y
```

(d) arm-softmmu.mak

Figure 7: Files modified to compile QEMU with a new device

5 Layer and Recipes

In section 4.4 we saw how to add a device to the board and compile QEMU with the new features. Now we need a device-driver to interface the device with the system. The driver is described in the file `virt-sha.c`. We will cross compile and load the driver using a dedicated recipe.

A recipe is a set of instructions that describes how to build a particular component, package, or software application for an embedded system. Poky uses these recipes to build custom Linux distributions as they provide metadata, source code, configuration, dependencies, and build instructions, enabling customization and flexibility.

1. Move to poky directory and execute the command

```
source oe-init-build-env build_qemuarm
```

2. Create a layer called meta-example with the command:

```
bitbake-layers create-layer ../meta-example
```

3. Add the layer to the system image:

```
bitbake-layers add-layer ../meta-example/
```

4. add the recipe virt-sha to the new layer:

```
cd ../meta-example/recipes-example
mkdir virt-sha
cd virt-sha
```

5. In virt-sha directory create the folder files and inside it create the file virt-sha.c: this file will describe the management of attributes: id, input, cmd, output, busy. These will be the interface between the user and the device.

In the file virt-sha.c describe the functions:

- vf_show_id
- vf_show_cmd and vf_store_cmd
- vf_show_input and vf_store_input
- vf_show_output
- vf_show_busy

show functions will be called with the command `cat <attribute>` to show the content of the attribute. store functions will be called with the command `echo <value> > <attribute>` to store into the attribute a certain value.

We will interface with the device through attributes. The command `cat <attribute>` displays the value of the attribute and to do so it recalls the function `virt_sha_read` described in the file `virt_sha.c`; an `echo <value> > <attribute>` command lets you modify the attribute content (insert the input, change the value of the command, ...). This operation on an attribute recalls the function `virt_sha_write` described in the file `virt_sha.c`.

Moreover, inside the file `virt_sha.c` we have the function `probe` which contains the function `ioremap`.

6. In the folder files we also add a Makefile.

```
1 obj-m := virt-sha.o
2
3 SRC := $(shell pwd)
4
5 all:
6     $(MAKE) -C $(KERNEL_SRC) M=$(SRC)
7
8 modules_install:
9     $(MAKE) INSTALL_MOD_DIR=kernel/drivers/misc -C $(KERNEL_SRC) M=$(SRC)
10    ↪ modules_install
11
12 clean:
13     rm -f *.o ~* core *.depend *.cmd *.ko *.mod.c
14     rm -f Module.markers Module.symvers modules.order
15     rm -rf .tmp_versions Modules.symvers
```

7. To boot the machine with the new device and device-driver, the file `local.conf` at the path `/home/-mattia/Desktop/progetto/poky/build_qemuarm/conf` has to be modified as shown in figure 8. In the ending part of the file you have to add the lines:

```
IMAGE_INSTALL_append = " virt-sha"
KERNEL_MODULE_AUTOLOAD += "virt-sha"
```

Inside the folder `virt-sha` we place the file `virt-sha.bb` which has this format:

```
1 DESCRIPTION = "Mymod Linux kernel module"
2 LICENSE = "MIT"
3 LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302
  ↳ "
4 inherit module
5 COMPATIBLE_MACHINE = "qemuarm"
6 SRC_URI = "file://Makefile \
  file://virt-sha.c \
  "
7
8
9 S = "${WORKDIR}"
```

Thanks to this file your module will be loaded and installed into the image (you need to compile everything to make it effective).

8. Build the new image with the command:

```
bitbake core-image-minimal
```

```
IMAGE_INSTALL_append = " myhello"
IMAGE_INSTALL_append = " virt-sha"
KERNEL_MODULE_AUTOLOAD += "virt-sha"
```

Figure 8: local.conf modified

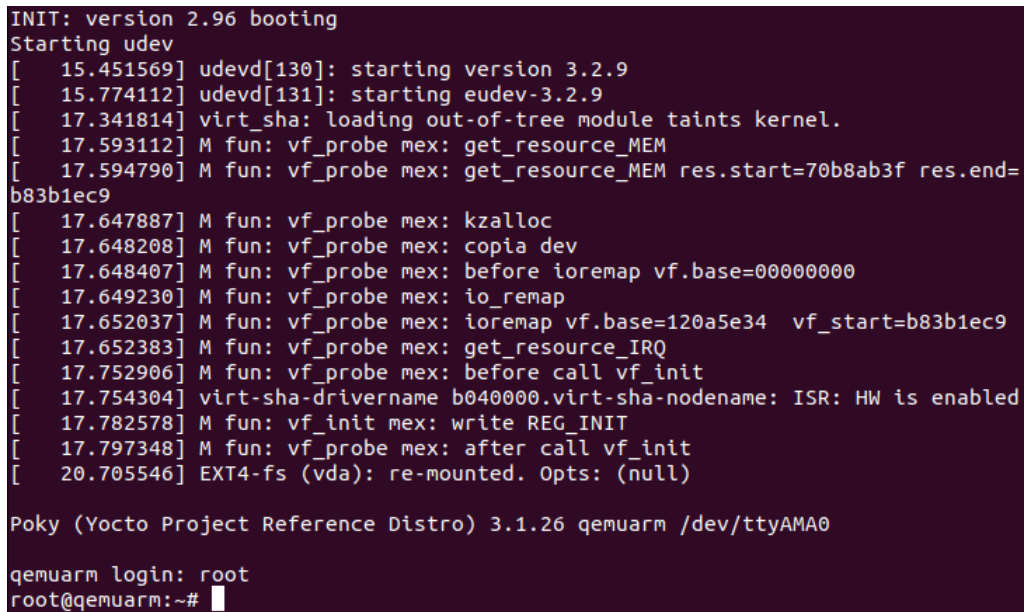
LAB 2 - Cryptocore Test and Library Creation

In this lab experience you will test the cryptocore simulated in LAB 1; then you will be required to code and cross compile a library which aim is to create an higher level interaction with the cryptocore device, allowing you to encrypt an input text string without using the device driver directly (avoiding messy and time consuming code repetition).

1 Attributes test

Up to now we have modified the board adding the device (4.4) and we have written the driver (5). Now we can test if everything is working.

1. Start the machine (lauch the machine manually (4.2) or with the script (4.3)).
Before entering the system you should see the initialization messages as in figure 9.



```
INIT: version 2.96 booting
Starting udev
[ 15.451569] udevd[130]: starting version 3.2.9
[ 15.774112] udevd[131]: starting eudev-3.2.9
[ 17.341814] virt_sha: loading out-of-tree module taints kernel.
[ 17.593112] M fun: vf_probe mex: get_resource_MEM
[ 17.594790] M fun: vf_probe mex: get_resource_MEM res.start=70b8ab3f res.end=
b83b1ec9
[ 17.647887] M fun: vf_probe mex: kzalloc
[ 17.648208] M fun: vf_probe mex: copia dev
[ 17.648407] M fun: vf_probe mex: before ioremap vf.base=00000000
[ 17.649230] M fun: vf_probe mex: io_remap
[ 17.652037] M fun: vf_probe mex: ioremap vf.base=120a5e34 vf_start=b83b1ec9
[ 17.652383] M fun: vf_probe mex: get_resource_IRQ
[ 17.752906] M fun: vf_probe mex: before call vf_init
[ 17.754304] virt-sha-drivername b040000.virt-sha-nodename: ISR: HW is enabled
[ 17.782578] M fun: vf_init mex: write REG_INIT
[ 17.797348] M fun: vf_probe mex: after call vf_init
[ 20.705546] EXT4-fs (vda): re-mounted. Opts: (null)

Poky (Yocto Project Reference Distro) 3.1.26 qemuarm /dev/ttyAMA0

qemuarm login: root
root@qemuarm:~#
```

Figure 9: Initialization messages

2. Enter the system with the user root.
Now you can explore inside the machine looking into different paths:
 - /sys/bus/platform/drivers
Here you should find your device driver with the name assigned in the previous lab (figure 10).
 - sys/devices/platform/
Here you should find your device with the name assigned in the previous lab (figure 11).

```

root@qemuarm:/sys/devices/platform# cd /sys/bus/platform/drivers/
root@qemuarm:/sys/bus/platform/drivers# ls
alarmtimer          of_fixed_factor_clk  virt-sha-drivename
armv7-pmu           pci-host-generic     virtio-mmio
gpio-clk            sbsa-uart
of_fixed_clk        simple-framebuffer
root@qemuarm:/sys/bus/platform/drivers#

```

Figure 10: Device driver inside the simulated ARM machine

```

qemuarm login: root
root@qemuarm:~# cd ../../
root@qemuarm:/# cd sys/devices/platform/
root@qemuarm:/sys/devices/platform# ls
0.flash              a002200.virtio_mmio
3f000000.pcie        a002400.virtio_mmio
90000000.pl011       a002600.virtio_mmio
90100000.pl031       a002800.virtio_mmio
90200000.fw-cfg      a002a00.virtio_mmio
90300000.pl061       a002c00.virtio_mmio
a000000.virtio_mmio  a002e00.virtio_mmio
a000200.virtio_mmio  a003000.virtio_mmio
a000400.virtio_mmio  a003200.virtio_mmio
a000600.virtio_mmio  a003400.virtio_mmio
a000800.virtio_mmio  a003600.virtio_mmio
a000a00.virtio_mmio  a003800.virtio_mmio
a000c00.virtio_mmio  a003a00.virtio_mmio
a000e00.virtio_mmio  a003c00.virtio_mmio
a001000.virtio_mmio  a003e00.virtio_mmio
a001200.virtio_mmio  b040000.virt-sha-nodename
a001400.virtio_mmio  gpio-keys
a001600.virtio_mmio  platform@c000000
a001800.virtio_mmio  power
a001a00.virtio_mmio  psci
a001c00.virtio_mmio  timer
a001e00.virtio_mmio  uevent
a002000.virtio_mmio
root@qemuarm:/sys/devices/platform#

```

Figure 11: Device inside the simulated ARM machine

- `sys/devices/platform/b040000.virt-sha-device`
Here you should find the device attributes of your device (figure 12).

```

root@qemuarm:/sys/bus/platform/drivers# cd /sys/devices/platform/b040000.virt-sh
a-nodename/
root@qemuarm:/sys/devices/platform/b040000.virt-sha-nodename# ls
busy          driver_override  modalias        power
cmd           id               of_node         subsystem
driver        input            output          uevent
root@qemuarm:/sys/devices/platform/b040000.virt-sha-nodename#

```

Figure 12: Device attributes

3. Now you can use the device attributes to test your device; the attributes can be managed exactly like files, so you can write and read them with these commands:

- `echo <VALUE> > <ATTRIBUTE>` to store the string `<VALUE>` inside a writable attribute.
- `cat <ATTRIBUTE>` to show the content of an attribute that can be read.

Anyway, since attributes can be considered as files, you can also open them in a script and write or read text inside them.

First we are going to test the device directly with the basic commands `echo <VALUE> > <ATTRIBUTE>` and `cat <ATTRIBUTE>`, but then we will encapsulate this procedure inside the functions of a library. The sequence of commands and outputs is shown in figure 13 .

Execute these commands in sequence:

(a) `cat id`

This command shows the `id` of the device.

(b) `cat busy`

This command shows the content of the attribute `busy`. This attribute is initialized to 0 and its value is 1 while the device is not available (i.e is busy). The `busy` attribute is automatically set to 1 once the input is written and it is set to 0 again once the output is read (the cryptocore device is ready to receive another input). You can execute this command many times to check the value of `busy` throughout the different steps.

(c) `echo ciao > input`

This command stores the string `ciao` inside the `input` attribute.

(d) `echo 1 > cmd`

This command stores the value 1 inside the `cmd` attribute, which default value is 0. Once the value 1 is stored the input string conversion process starts.

(e) `cat output`

This command shows the content of the attribute `output`. If the `cmd` attribute has been set to 1 the `output` attribute stores the result of the SHA-256 hashing function on the input string.

```
root@qemuarm:/sys/devices/platform/b040000.virt-sha-nodename# cat id
Chip ID: 0xf001
root@qemuarm:/sys/devices/platform/b040000.virt-sha-nodename# cat busy
0
root@qemuarm:/sys/devices/platform/b040000.virt-sha-nodename# echo ciao > input
root@qemuarm:/sys/devices/platform/b040000.virt-sha-nodename# cat busy
1
root@qemuarm:/sys/devices/platform/b040000.virt-sha-nodename# echo 1 > cmd
root@qemuarm:/sys/devices/platform/b040000.virt-sha-nodename# cat output
b133a0c0e9bee3be20163d2ad31d6248db292aa6dcb1ee087a2aa50e0fc75ae2
root@qemuarm:/sys/devices/platform/b040000.virt-sha-nodename# cat busy
0
root@qemuarm:/sys/devices/platform/b040000.virt-sha-nodename#
```

Figure 13: Attributes test

2 Library description

If everything went well up until now it means that you have a working device interfaced with the user through device attributes.

Accessing the device directly through those attributes is not user-friendly, so specific libraries are usually provided with any device. Now you are required to develop a library which implements specific functions to simplify the interaction with the device.

Tip: among the other functions implement `sha256_algorithm(input, output, number of characters)` that will be used by all the user programs that want to hash a string with SHA-256. This function should take care of all the command flow that you used to manually execute in the terminal (e.g. `echo ciao > input`, `echo 1 > cmd`, `cat output`).

Here is an example of the functions that could be contained in the library:

```
int sha256_inizialize();
int sha256_request(const char* input, int n);
int sha256_get_output(char* output);
int sha256_check_busy();
int sha256_algorithm(char* input,int n, char* output);
```

- `sha256_initialize`
This function can be used for any kind of initialization.
- `sha256_request`
This function inserts the input string inside the `input` attribute and starts the conversion process by inserting the value 1 inside the `cmd` attribute.
- `sha256_get_output`
This function retrieves the output of the encryption process by reading the content of the `output` attribute.
- `sha256_check_busy`
This function retrieves the value stored in the `busy` attribute in order to understand if the device is free or if it is currently hashing another input string.
- `sha256_algorithm`
This function recalls in sequence
 1. `sha256_check_busy` (polling)
 2. `sha256_request`
 3. `sha256_get_output`

3 Library test

In this section we will implement a trivial application, just to understand how to cross compile and use the library. To do so, we will write a C program `Test_sha256.c` which will exploit the library described in section 2. In order to cross compile our program and the library we have to create a new Poky recipe following these steps:

1. At the path:

```
/poky/meta-example/recipes-example/
```

create a new directory with the command:

```
mkdir myhello
```

2. Inside the folder `myhello` create:

- the subfolder `files`
- the file `myhello.bb`

3. Inside the folder `files` insert the program source `Test_sha256.c` and the library `Sha256_library.c`, `Sha256_library.h`

4. Edit the file `myhello.bb` in order to cross compile all the files needed into a single executable.
This is how the `myhello.bb` file should look like after you have edited it:

```
1 DESCRIPTION = "Simple helloworld application"
2 LICENSE = "MIT"
3 LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302
  ↪ "
4 SRC_URI = "file://Sha256_library.c file://Sha256_library.h file://Test_sha256.c"
5 S = "${WORKDIR}"
```

```

6 do_compile() {
7     set CFLAGS -g
8     ${CC} ${CFLAGS} Sha256_library.c Sha256_library.h Test_sha256.c ${LDFLAGS} -o
      ↪ TestSha256
9     unset CFLAGS
10 }
11 do_install() {
12     install -d ${D}${bindir}
13     install -m 0755 TestSha256 ${D}${bindir}
14 }

```

(since this is only a user program you do not need to insert a Makefile as you did when you cross compiled the device driver)

5. Add the following line in the the ending part of the file `local.conf`:

```
IMAGE_INSTALL_append = " myhello"
```

(do not insert the line `KERNEL_MODULE_AUTOLOAD` since it is only needed when dealing with a driver)

6. Run the ARM machine simulation and you should find the executable file `TestSha256` at the path `/usr/bin/` (inside the ARM machine).

LAB 3 - Application Program with SHA-256

In this lab experience you will have to program an application that can take full advantage of the encoding device and the library you created (alternatively you can find a working library for the device in the lab solutions). The application has to be written in C language and needs to be cross compiled along with the *Sha256_library* as shown in the previous lab experience with the `Test_sha256.c` file.

1 Program specification

In the material folder you can find the program `UnsafeRegistration.c`. Open it and try to understand how the code works: try to compile it and run it with a series of test inputs.

The generated executable simulates a **login system** which manages multiple users registered with name, password and phone number (which can be used in case a user forgets the password).

As you can see the program is working in all its parts and the databases with usernames, credentials etc. are properly stored. However, **the program is not safe!** Imagine if this same program was used in a company to allow the employees identification and let them access a private area to upload their projects and gather sensitive information about the company; do you notice any security vulnerabilities that could be exploited by someone inside or outside the company?

Tip: The database is currently composed of a bunch of vulnerable text files which contain usernames, passwords and telephone numbers. These information need to be hashed by the SHA-256 algorithm in case a malicious user accesses the files in which these sensitive data are stored.

Try to edit the provided `.c` file and turn it into `SafeRegistration.c` by fixing the security issue. To do this, use the SHA-256 algorithm by interfacing the application with the device through the library *Sha256_library*.

Tip: Some files must be hashed, but at the same time the access by the user who enters username and password (as well as the possibility of changing password only if the second factor corresponds to the correct one) must be guaranteed and verified.

Once you have prepared the `SafeRegistration.c` file, place in the `files` directory, where you have to insert the following files:

- `Sha256_library.h` and `Sha256_library.c`
- `SafeRegistration.c`

You have to modify `myhello.bb` in this way:

```
1 DESCRIPTION = "Simple helloworld application"
2 LICENSE = "MIT"
3 LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
4 SRC_URI = "file://Sha256_library.c file://Sha256_library.h file://Test_sha256.c"
5 S = "${WORKDIR}"
6 do_compile() {
7     set CFLAGS -g
8     ${CC} ${CFLAGS} Sha256_library.c Sha256_library.h Test_sha256.c ${LDFLAGS} -o TestSha256
9     unset CFLAGS
10 }
11 do_install() {
12     install -d ${D}${bindir}
13     install -m 0755 TestSha256 ${D}${bindir}
14 }
```

2 About SHA-256

SHA-256 (Secure Hash Algorithm 256-bit) cannot be reverted because it is a one-way cryptographic **hash function**. When you apply SHA-256 to data, it produces a fixed-length output (256 bits), which appears random and is computationally **infeasible to reverse back** to the original data. This property is crucial for data integrity and security, as it ensures that you cannot determine the original input solely from its hash value.

Notice that, even when SHA-256 is used, the `SafeRegistration.c` login system may be still **vulnerable** to attacks. **Brute forcing SHA-256** means trying all possible input values until you find one that produces the same hash as the target. Given that SHA-256 has a large output space (256 bits), it is computationally infeasible to perform a brute force attack on it in practice, especially for long and complex inputs. In our case, though, the attacker could know part of the input that will be hashed: the telephone number is a private information, but it can easily fall into the attacker's hands!

Knowing part of the key (input of the hashing function) reduces the number of possible combinations that need to be tried during a brute force attack (it improves efficiency also in other more sophisticated kinds of attacks). This can significantly speed up the attack because the attacker doesn't need to test all possible keys; they can focus on a subset of the keys that match the known part.

