



Politecnico di Torino
Collegio di Elettronica, Telecomunicazioni e Fisica

Master degree in Electronics Engineering

Laboratory Experiment
Device Driver Development for Linux

Professor: Stephano di Carlo

Authors:
Berkay Demir (s308879)
Firat Kagitci (s314487)
Malaz Abdalla Osman Mohamed (s315390)
Yang Zeyu (s315346)
Shanchong Shang (s314324)

Contents

1	Introduction	1
2	Experiment 1 - Environment Setup	2
2.1	installing QEMU	2
2.2	Installing Buildroot	2
3	Experiment 2 - Your First Kernel Module	5
3.1	Cross-Compiling Kernel Module For Buildroot Linux running on RISC-V:	6
4	Experiment 3-SHA256 Cryptocore device driver Development	9
4.1	SHA-256 cryptocore	9
4.2	QEMU File Configuration	9
4.2.1	congratulation files	9
5	Experiment 4 - SHA-256 Cryptocore Device test	12
6	Experiment 5 - The SHA-256 Device Driver	14
6.1	User Program for Testing	14
7	Experiment 6 - Device Initialization and Final Test	15
7.1	Device Registration and test	15
7.2	Your turn to explore	16
8	Potential Attacks to Linux driver	17
8.1	Denial of Service (DoS)	17
8.2	Buffer Overflow	18
8.3	Memory Corruption	18
8.4	Conclusion	18

CHAPTER 1

Introduction

This Lab experiment is prepared to learn how to create device drivers in Linux and how to use them to establish communication with the device. in this specific laboratory; the Builtroot-Linux is the target environment. it will be running with QEMU emulating RISC-V architecture.

A simple kernel module will be used as a test at first and built on that, the student will investigate the SHA-256 crypto core device driver that will be provided with a basic program to establish a higher level of communication with the driver.

Students attempting to do this lab experiment will be better equipped if they have the knowledge of the basics of operating systems and kernel modules.

CHAPTER 2

Experiment 1 - Environment Setup

The development environment setup is going to be completed in this experiment. The Linux version used is Ubuntu Version 23.10. it will be the host OS, and it will be emulated in Virtualbox.

2.1 installing QEMU

The first step is installing Qemu from the git repository and beforehand make sure you have the necessary toolchains and dependencies: Installing dependencies:

```
sudo apt-get update
sudo apt-get install -y build-essential git ncurses-dev bison flex libssl-dev make gcc
sudo apt install ninja-build
sudo apt-get install pkg-config
sudo apt-get install libglib2.0-dev
sudo apt-get install libpixman-1-dev
```

Note: In case you still encounter any dependency errors which is highly possible, read the error messages and install the required dependent tools indicated in the errors.

```
git clone https://github.com/buildroot/buildroot.git
cd buildroot/
make qemu_riscv64_virt_defconfig
make
```

Configure the system for risc-v architecture by the following command:

```
./configure --target-list=riscv64-softmmu
make
```

2.2 Installing Buildroot

Then you have to install Buildroot.

```
./configure --target
list=riscv64-sofmmu
make
```

The last make command might take a long time, possibly more than one hour since it compiles the entire kernel of the target system.

After the kernel compilation command make you will see the image files inside `/buildroot/output/images` directory path. Go to the `/images` directory and you will see the script named `start-qemu.sh`. You need to modify the directory inside this script to your qemu/build directory path.

`buildroot/output/images/start_qemu.sh` file content:

```
#!/bin/sh

BINARIES_DIR="${0%/*}/"
# shellcheck disable=SC2164
cd "${BINARIES_DIR}"

mode_serial=false
mode_sys_qemu=false
while [ "$1" ]; do
    case "$1" in
        --serial-only|serial-only) mode_serial=true; shift;;
        --use-system-qemu) mode_sys_qemu=true; shift;;
        --) shift; break;;
        *) echo "unknown option: $1" >&2; exit 1;;
    esac
done

if ${mode_serial}; then
    EXTRA_ARGS='-nographic'
else
    EXTRA_ARGS=''
fi

if ! ${mode_sys_qemu}; then
    export PATH="/home/asdf/Desktop/qemu/build:${PATH}" # MODIFY THIS LINE TO QEMU/BUILD
fi

exec qemu-system-riscv64 -M virt -bios fw_jump.elf -kernel Image -append "rootwait root=/dev/vda ro" -d
```

By adding `./` to the beginning: `./start-qemu.sh`, you can run the buildroot.

Keep in mind that every time you compile the buildroot, you need to change this file. But if you don't want to deal with this you can use the following command, remember to adjust the directory path to your system.

```
exec /Desktop/qemu/build/qemu-system-riscv64 -M virt -bios fw_jump.elf -kernel Image -append "rootw
```

This will boot the system with OpenSbi, and your Buildroot system will start. The password is 'root' and keep in mind that this system does not have a GUI(Graphical User Interface), meaning you will have to use the command line to interact with the new system we just created. The password: root.

```
asdf@asdf: ~/Desktop/qemu/buildroot/output/images
asdf@asdf:~/Desktop/qemu/buildroot/output/images$ ./start-qemu.sh

OpenSBI v1.2

Platform Name      : riscv-virtio,qemu
Platform Features  : medeleg
Platform HART Count : 1
Platform IPI Device : aclint-mswi
Platform Timer Device : aclint-mtimer @ 100000000Hz
Platform Console Device : uart8250
Platform HSM Device : ---
Platform PMU Device : ---
Platform Reboot Device : sifive_test
Platform Shutdown Device : sifive_test
Firmware Base      : 0x80000000
Firmware Size      : 212 KB
```

```
asdf@asdf: ~/Desktop/qemu/buildroot/output/images
Starting network: udhcpd: started, v1.36.1
udhcpd: broadcasting discover
udhcpd: broadcasting select for 10.0.2.15, server 10.0.2.2
udhcpd: lease of 10.0.2.15 obtained from 10.0.2.2, lease time 86400
deleting routers
adding dns 10.0.2.3
OK

Welcome to Buildroot
buildroot login: root
# ls
# cd ..
# cd ..
# ls
bin            lib            lost+found    opt            run            tmp
dev            lib64          media         proc           sbin           usr
etc            linuxrc        mnt           root           sys            var
# cd lib
# cd modules/
# ls
6.1.44
# pwd
/lib/modules
#
```

CHAPTER 3

Experiment 2 - Your First Kernel Module

It is highly suggested that you start with a simple kernel module to test your system, in our case we will create a kernel module that gives a message as 'Hello World'. First, you test it on your host machine Ubuntu. Create a separate directory and name it as you wish (e.g. kmodules) and create the `hello.c` file (`touch hello.c`).

Simple kernel module 'hello.c':

```
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your-Name");
MODULE_DESCRIPTION("A simple hello-world kernel module");

static int __init hello_init(void) {
    printk(KERN_INFO "Hello , -world!\n");
    return 0; // Non-zero return means that the module couldn't be loaded.
}

static void __exit hello_cleanup(void) {
    printk(KERN_INFO "Cleaning-up-module.\n");
}

module_init(hello_init);
module_exit(hello_cleanup);
```

To compile the `hello.c` code we need to use a Makefile file that includes the following script:

```
obj-m += hello.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Here `uname -r` command gives the kernel header name of the system.

Keep in mind that just before make command (inside Makefile) there should be a tab-sized space. Save the Makefile and use make command to compile this C code to create a kernel module. After

that, you will see multiple files created including `hello.ko` which is the kernel module created for the Ubuntu host. Now that we have the kernel module, we can add it to our main kernel by using the command `sudo insmod hello.ko`, now you need to run `dmesg` command to see its effect on the kernel messages prompt as "Hello, World!" message.

Your task: Try to change the message to something else and check it.

3.1 Cross-Compiling Kernel Module For Buildroot Linux running on RISC-V:

We have created a simple kernel module and executed it, this is a good practice to understand the kernel module development. We need to install cross-compilation tools to compile a kernel module for our target architecture RISC-V. The following command is used to install the required cross-compilation toolchain:

```
sudo apt-get update
sudo apt-get install gcc-riscv64-linux-gnu
```

Create another directory for kernel modules of the Buildroot RISC-V based system. name it as you wish(e.g. `kmodules_target`)

Inside, create the same `hello.c`, and create a new Makefile file. Inside the new Makefile write the following script, that uses cross-compilation tool:

```
obj-m += hello.o
all:
make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- -C <path-to-buildroot-kernel> M=$(PWD) modules

clean:
make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- -C <path-to-buildroot-kernel> M=$(PWD) clean
```

path-to-buildroot-kernel: As an example, this path should be like the following:

```
/home/user/Desktop/buildroot/output/build/linux-6.6.36
```

the user and the Linux version(on Buildroot) may be different on your project, so check it and modify accordingly.

Save the Makefile file, and use again the make command on the shell to compile. Then you will see multiple files created including `hello.ko` specifically cross-compiled for our new Buildroot-Linux development environment running on RISC-V architecture. You might question how the `hello.ko` kernel module is loaded to our development environment. Here are the steps:

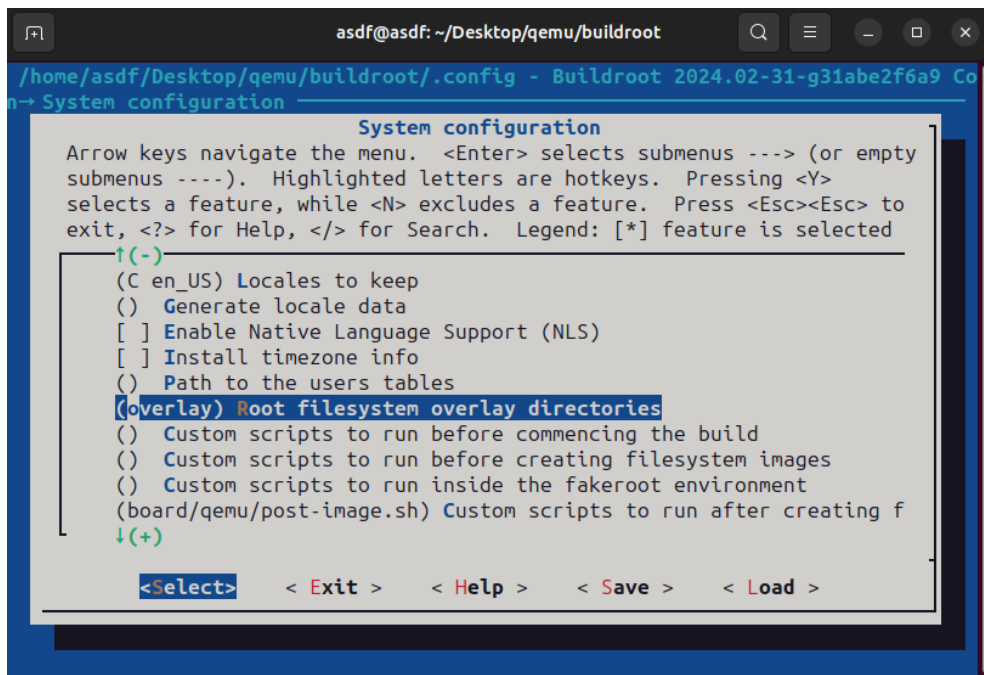
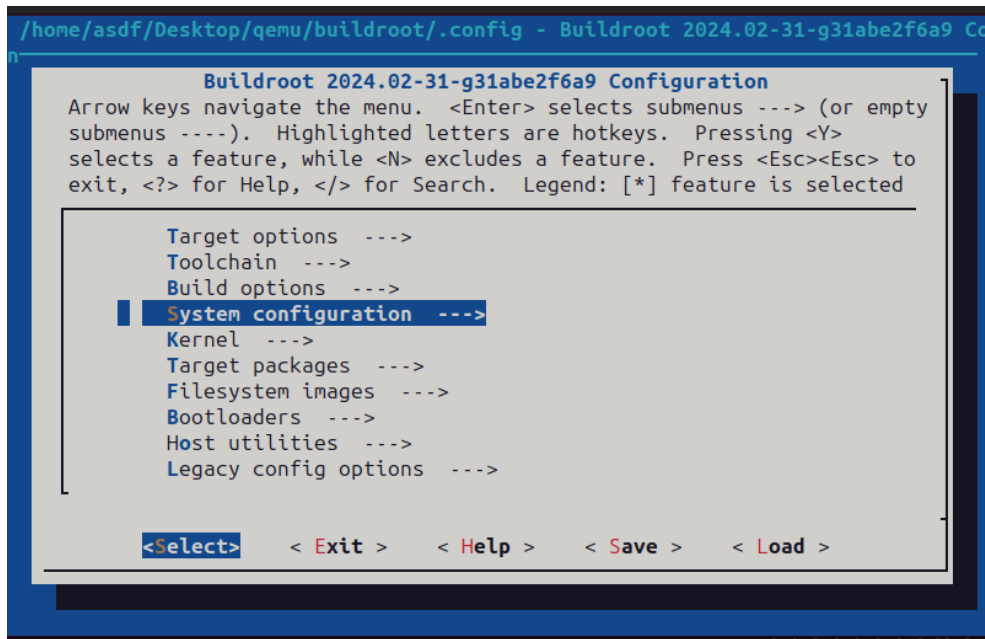
- Go to `/buildroot`, inside create a directory named as you wish (or name it as `overlay` for compatibility), inside create another path (e.g. `/home/driver`)
- use this: `mkdir -p overlay/home/driver`. The `-p` flag ensures that `mkdir` creates all necessary parent directories that do not exist.
- copy the kernel module '`hello.ko`' to `overlay/home/driver` directory.

This path is the path to your kernel header on your target environment.

- After doing this, inside `/buildroot` directory you need to run `make menuconfig` which opens a text-based GUI. Keep in mind that this interface works with a dependency, so run the following command to resolve it if you receive an error

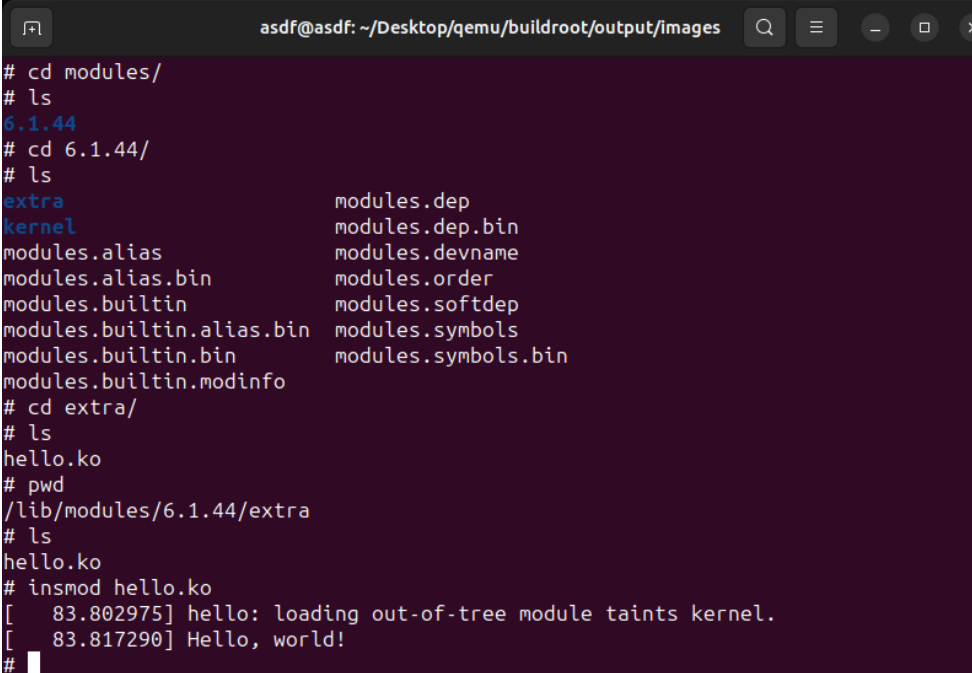

```
sudo apt-get install libncurses5-dev libncursesw5-dev
```

- Configure Buildroot to Use the Overlay: You need to tell Buildroot to use this overlay directory when building the root filesystem image. This can be done in the Buildroot configuration:
- After running `make menuconfig` within your Buildroot directory; Navigate to System configuration → Root filesystem overlay directories. Write 'overlay' to the blank space.
- Save and exit the configuration menu.



Then run make inside /buildroot. Every time when you add a new file to the overlay directory, you need to recompile the buildroot so that you will see the changes in the buildroot.

To see if the kernel module is loaded to Buildroot, start the Buildroot by executing the command `./start-qemu.sh` inside `/buildroot/output/images/`. then the Buildroot system will start, you need to check the directory `/home/driver`, there you should see the loaded kernel module. You can insert this kernel module (`hello.ko`) to the running kernel by using `insmod hello.ko` command, then if you use `dmesg` command you will see as an output the "Hello, World!" message. This means you have successfully cross-compiled a kernel module, added it to your target system, and then inserted it into the running kernel.

A terminal window with a dark purple background and white text. The window title is 'asdf@asdf: ~/Desktop/qemu/buildroot/output/images'. The terminal shows the following commands and output:

```
# cd modules/  
# ls  
6.1.44  
# cd 6.1.44/  
# ls  
extra          modules.dep  
kernel         modules.dep.bin  
modules.alias  modules.devname  
modules.alias.bin  modules.order  
modules.builtin  modules.softdep  
modules.builtin.alias.bin  modules.symbols  
modules.builtin.bin  modules.symbols.bin  
modules.builtin.modinfo  
# cd extra/  
# ls  
hello.ko  
# pwd  
/lib/modules/6.1.44/extra  
# ls  
hello.ko  
# insmod hello.ko  
[ 83.802975] hello: loading out-of-tree module taints kernel.  
[ 83.817290] Hello, world!  
#
```

Until now you have practiced and learned how to:

- set the development environment Buildroot-Linux running on RISC-V architecture
- create a simple kernel module
- cross-compile a kernel module
- add the kernel module to Buildroot-Linux that is running on RISC-V architecture

These steps are also helpful in creating the cryptographic kernel module which is task in the following chapter.

CHAPTER 4

Experiment 3-SHA256 Cryptocore device driver Development

4.1 SHA-256 cryptocore

The Cryptocore is developed for implementing the SHA-256 algorithm along with the necessary functions required for it to be seen as a device inside qemu.

the file of the cryptocore (`crypto.c` and `crypto.h`) can be found in the material folder.
take them and place it in the folder `qemu/hw/misc/`

4.2 QEMU File Configuration

The device that is located at the address `0x4000000` in the virtual address space, and with additional registers for various operations.

4.2.1 congratulation files

implement the following configurations:

File: `qemu/hw/misc/Kconfig`

```
config ARMSSE_MHU
bool
```

```
config ARMSSE_CPU_PWRCTRL
bool
```

```
+ config SHA256_DEVICE
bool
```

File: `qemu/hw/misc/meson.build`

```
softmmu_ss.add(when: 'CONFIG_APPLESMC', if_true: files('applesmc.c'))
+ softmmu_ss.add(when: 'CONFIG_SHA256_DEVICE', if_true: files('crypto.c'))
softmmu_ss.add(when: 'CONFIG_EDU', if_true: files('edu.c'))
softmmu_ss.add(when: 'CONFIG_FW_CFG_DMA', if_true: files('vmcoreinfo.c'))
```

File: qemu/hw/riscv/Kconfig

```
select SERIAL
select SIFIVE_CLINT
select SIFIVE_PLIC
select SIFIVE_TEST
select VIRTIO_MMIO
select FW_CFG_DMA
+ select SHA256_DEVICE
```

File: qemu/hw/riscv/virt.c (which is the Risc-V Virtual Machine Configuration file, in which you should add the cryptocore header file:

```
#include "chardev/char.h"
#include "sysemu/arch_init.h"
#include "sysemu/device_tree.h"
#include "sysemu/sysemu.h"
#include "hw/pci/pci.h"
#include "hw/pci-host/gpex.h"
#include "hw/display/ramfb.h"
+ #include "hw/misc/crypto.h"
```

Then allocate memory space for the device. File: qemu/include/hw/riscv/virt.h

```
static const MemMapEntry virt_memmap[] = {
[VIRT_DEBUG] = { 0x0, 0x100 },
[VIRT_MROM] = { 0x1000, 0xf000 },
[VIRT_TEST] = { 0x100000, 0x1000 },
[VIRT_RTC] = { 0x101000, 0x1000 },
[VIRT_CLINT] = { 0x2000000, 0x10000 },
[VIRT_PCIE_PIO] = { 0x3000000, 0x10000 },
+ [VIRT_SHA256_DEVICE] = { 0x4000000, 0x10000 },
[VIRT_PLIC] = { 0xc000000, VIRT_PLIC_SIZE(VIRT_CPUS_MAX * 2) },
[VIRT_UART0] = { 0x10000000, 0x100 },
[VIRT_VIRTIO] = { 0x10001000, 0x1000 },
[VIRT_FW_CFG] = { 0x10100000, 0x18 },
[VIRT_FLASH] = { 0x20000000, 0x4000000 },
[VIRT_PCIE_ECAM] = { 0x30000000, 0x10000000 },
[VIRT_PCIE_MMIO] = { 0x40000000, 0x40000000 },
[VIRT_DRAM] = { 0x80000000, 0x0 },
};
```

The function `sha_device_create(memmap[VIRT_SHA256_DEVICE].base)` initializes the SHA-256 cryptographic device at the specified base memory address within the virtual machine.

```
/* SiFive Test MMIO device */
sifive_test_create(memmap[VIRT_TEST].base);

/* SHA256_DEVICE */
+ sha_device_create(memmap[VIRT_SHA256_DEVICE].base);

/* VirtIO MMIO devices */
```

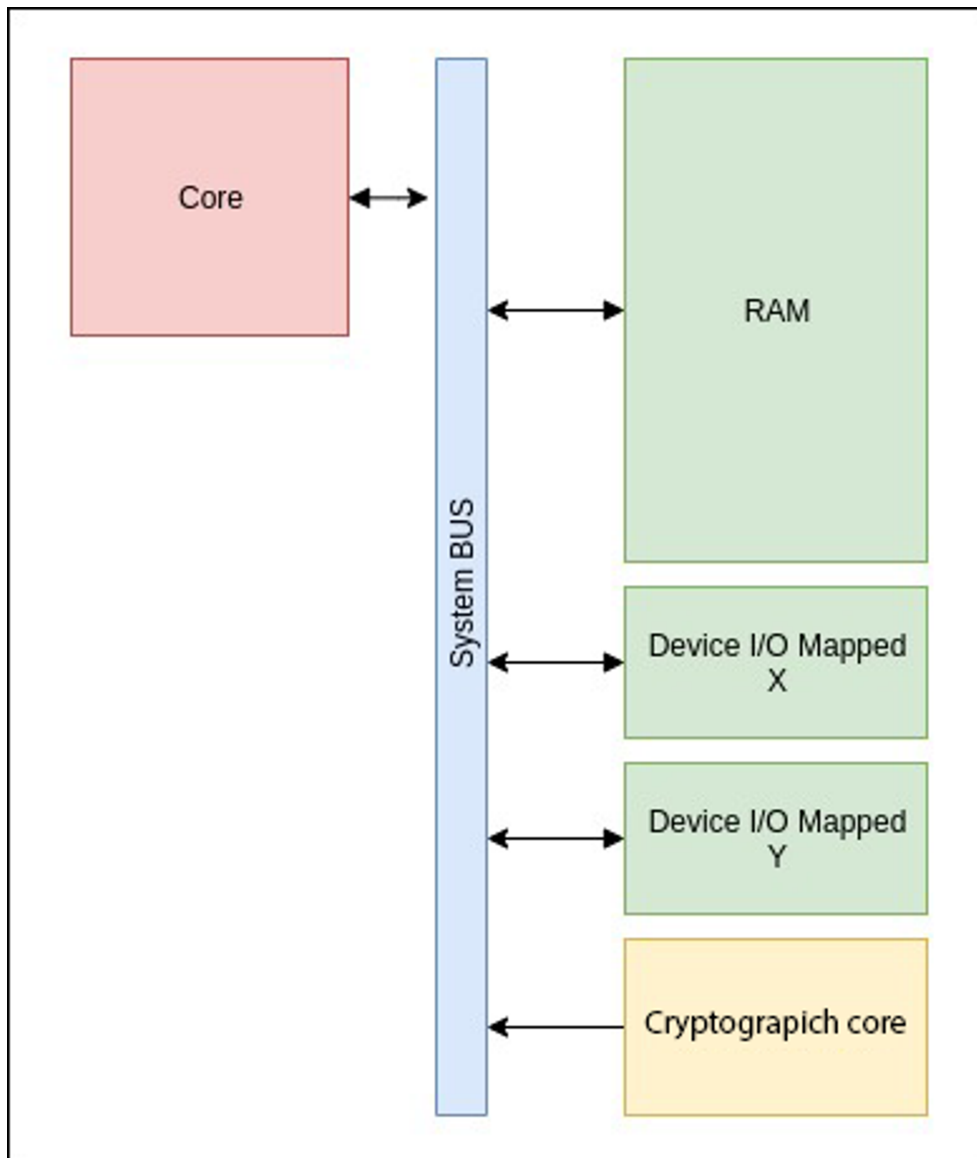
```
for (i = 0; i < VIRTIO_COUNT; i++) {  
    sysbus_create_simple("virtio-mmio",
```

After doing all these steps we need to configure and compile the qemu.

```
./configure --target-list=riscv64-softmmu  
make
```

Note: You should configure and compile the qemu every time you modify your cryptocore, so that the modifications are reflected to the buildroot environment.

Your task: Understand the functions within the driver - and most importantly the read and write functions - within the template that has been availed in the material folder.



CHAPTER 5

Experiment 4 - SHA-256 Cryptocore Device test

Now that you have developed the device in the previous steps, we can run a quick test with some simple functions without the driver yet.

BusyBox's `devmem` write and read functions are the way to do so, the following steps will show you how.

- Writing to the device

with `devmem` command you write a value to a specific memory address. the format followed is :
`devmem address width value`

The `address` field: The memory address of the device register.

The `width` field: The data width (8, 16, 32, or 64 bits).

The `value` field: The value to write.

Example: `devmem 0x4000010 32 0x1`

- Read from the device

Use the `devmem` command to read a value from a specific memory address.

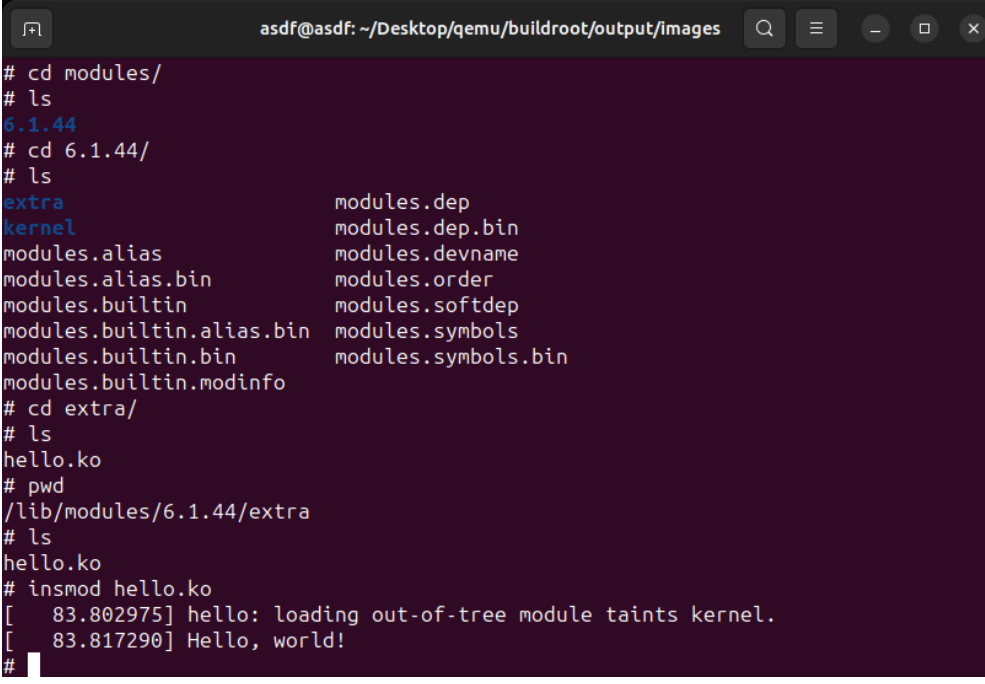
`devmem address width`

The `address` field: The memory address of the device register.

The `width` field: The data width (8, 16, 32, or 64 bits).

Example: `devmem 0x4000010 32`

An example for so: the string 'firat kagitci' was transformed into hex and put in the input region of the device which is 0x4000010, then to enable the device we wrote 0x1 to the enable register 0x4000008, as soon as we enable the device, it gave us the correct output result on the terminal (that print was for the debug code inside the crypto core). The result matched that acquired from an online SHA256 encryption tool.

A terminal window with a dark purple background and white text. The window title is 'asdf@asdf: ~/Desktop/qemu/buildroot/output/images'. The terminal shows a sequence of commands and their outputs. The user navigates to the 'modules/' directory, lists its contents, then enters the '6.1.44/' subdirectory. A second 'ls' command shows a list of files including 'extra', 'kernel', and various module-related files. The user then enters the 'extra/' directory, lists its contents (showing 'hello.ko'), checks the current directory with 'pwd', and finally loads the 'hello.ko' module using 'insmod'. The output shows the module being loaded and a 'Hello, world!' message.

```
asdf@asdf: ~/Desktop/qemu/buildroot/output/images
# cd modules/
# ls
6.1.44
# cd 6.1.44/
# ls
extra          modules.dep
kernel         modules.dep.bin
modules.alias  modules.devname
modules.alias.bin  modules.order
modules.builtin  modules.softdep
modules.builtin.alias.bin  modules.symbols
modules.builtin.bin  modules.symbols.bin
modules.builtin.modinfo
# cd extra/
# ls
hello.ko
# pwd
/lib/modules/6.1.44/extra
# ls
hello.ko
# insmod hello.ko
[ 83.802975] hello: loading out-of-tree module taints kernel.
[ 83.817290] Hello, world!
#
```

Your task: Try different inputs and compare them to this online SHA256 encryption tool : https://xorbin.com/tools/sha256-hash-calculator#google_vignette

CHAPTER 6

Experiment 5 - The SHA-256 Device Driver

The cross compilation steps explained before are to be followed similarly for this driver.

After the cross compilation add the file to the `buildroot/overlay/home/driver` directory so that you can upload the kernel module, the output file with the `.ko` extension to the buildroot development environment.

the complete `driver_sh.c` file is in the material section.

6.1 User Program for Testing

The driver requires one extra level for communication between the user and the device, which is why a user program is necessary to allow data exchange between the user and the device driver with ease. User programs are not able to write directly to the device's memory regions, that is why inside the driver we have used `copy_from_user` and `copy_to_user` functions.

Add the user program to the `buildroot/overlay/home` directory, after that, cross-compile `sha256_test.c` (found in the material folder) file according to our target system with this command:

```
/home/asdf/Desktop/buildroot/output/host/bin/riscv64-buildroot-linux  
-gnu-gcc -o sha256_test sha256_test.c -static
```

Remember to change the directory path according to your system.

After all additions, you need to compile the Buildroot, by running `make` inside `/buildroot` directory. Now your device, device driver, and user program.

CHAPTER 7

Experiment 6 - Device Initialization and Final Test

7.1 Device Registration and test

Start the device, if the following command doesn't work you can always use the `start_qemu.sh` as long as you correct the directory path inside it:

```
exec /Desktop/qemu/build/qemu-system-riscv64 -M virt -bios fw_jump.elf -
kernel Image --append "rootwait root=/dev/vda-ro" --drive file=rootfs.
ext2,format=raw,id=hd0 --device virtio-blk-device,drive=hd0 --netdev
user,id=net0 --device virtio-net-device,netdev=net0 --nographic ${
EXTRA_ARGS} "$@"
```

Add the kernel module to the running kernel by using the following command inside the `/home/-driver` directory:

```
insmod driver_sh.ko
```

You have to see some kernel messages in the shell indicating the success of the device and class creation registration with a major number.

```
Welcome to Buildroot
buildroot login: root
# cd ..
# cd /home/driver/
# insmod driver_sh.ko
[ 58.995779] driver_sh: loading out-of-tree module taints kernel.
[ 59.005570] SHA256: Initializing the SHA256 LKM
[ 59.006407] SHA256: registered correctly with major number 243
[ 59.008573] SHA256: device class registered correctly
[ 59.016983] SHA256: device class created correctly
# pwd
/home/driver
#
```

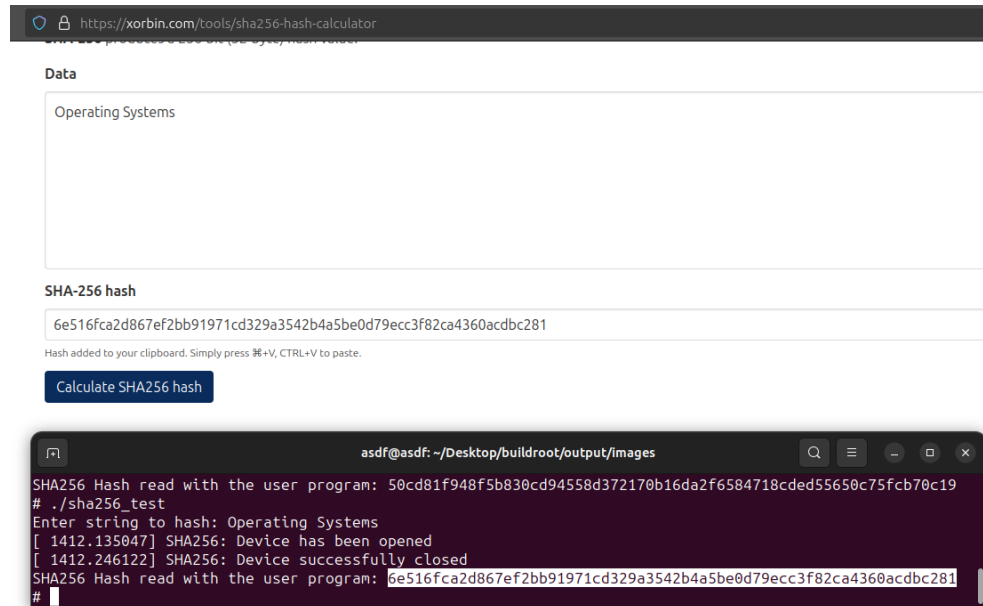
Then go to the directory `/home` and run the user program executable file by writing to shell `./sha256_test`, it will ask you to enter a string.

```
# pwd
/home
# ./sha256_test
Enter string to hash: 
```

```
# ./sha256_test
Enter string to hash: firat kagitci
[ 1323.206412] SHA256: Device has been opened
[ 1323.325338] SHA256: Device successfully closed
SHA256 Hash read with the user program: 50cd81f948f5b830cd94558d372170b16da2f6584718cded55650c75fcb70c19
#
```

```
# ./sha256_test
Enter string to hash: Operating Systems
[ 1412.135047] SHA256: Device has been opened
[ 1412.246122] SHA256: Device successfully closed
SHA256 Hash read with the user program: 6e516fca2d867ef2bb91971cd329a3542b4a5be0d79ecc3f82ca4360acdbc281
#
```

Comparison with the result from an online tool:



Your task: Try some combinations and try to compare their correctness to the tool provided in the last chapter.

7.2 Your turn to explore

SHA256 is a strong hashing algorithm that ensures safety of information. It is designed to work in one-way: computationally, it should not be reversed, and the original input must not be acquired. So, the output of the encryption only gets compared to that in the database to ensure matching.

However, it has been found that in rare cases, very simple text (including upper and lowercase English characters and simple numbers combinations) can be decrypted.

Your task is to investigate the reason behind it, and find two or three modifications to the input that make it impossible to be decrypted. (Online tools are available for encryption/decryption of SHA256)

CHAPTER 8

Potential Attacks to Linux driver

Linux drivers, which operate at the kernel level, are critical components of the operating system. They provide an interface between the hardware and the kernel, allowing the system to interact with and control hardware devices. Due to their privileged access and essential role, Linux drivers are attractive targets for attackers. Below are detailed descriptions of potential attacks that can target Linux drivers:

8.1 Denial of Service (DoS)

Denial of Service (DoS) attacks are a significant threat to the stability and availability of systems and services. In the context of Linux drivers, DoS attacks can exploit vulnerabilities to disrupt normal operation, making the system unresponsive or unavailable. Here is a more detailed introduction to DoS attacks, including mechanisms, examples, and mitigations:

Mechanisms of DoS Attacks

Resource Exhaustion:

Memory Exhaustion: Consuming all available system memory, causing the kernel to kill processes or crash the system due to Out-Of-Memory (OOM) conditions.

Disk I/O Flooding: Generating excessive disk read/write operations, which can saturate the disk I/O bandwidth and degrade performance for other applications.

Resource Locking:

Deadlock Induction: Triggering a condition where two or more processes lock resources in such a way that they prevent each other from proceeding.

Starvation: Continuously keeping critical resources occupied so that other processes are unable to use them, leading to performance degradation or system hang.

Examples of DoS Attacks on Linux Drivers

Malformed Input: Sending malformed input data to a driver that does not adequately validate input can cause the driver to crash or hang. Example: A USB driver might crash if it receives a specially crafted packet that it cannot parse correctly, leading to a DoS condition.

Mitigations for DoS Attacks on Linux Drivers

Input Validation: Thoroughly validate all inputs from user space and hardware interfaces to ensure they are within expected bounds and formats. Use defensive programming techniques to handle unexpected or malformed inputs gracefully.

Resource Management: Implement limits on resource usage to prevent single processes from exhausting critical resources. Use techniques like rate limiting, quotas, and resource accounting to control resource consumption.

Concurrency Control: Use proper synchronization mechanisms to manage access to shared resources and prevent race conditions and deadlocks. Ensure that critical sections of code are properly protected and that resources are released promptly.

Robust Error Handling: Implement comprehensive error handling to deal with exceptional conditions and avoid leaving the system in an unstable state. Ensure that resources are correctly cleaned up in the event of errors to prevent leaks and deadlocks.

8.2 Buffer Overflow

Buffer overflow attacks exploit vulnerabilities where a program writes more data to a buffer than it can hold, leading to memory corruption, crashes, or arbitrary code execution. In the context of Linux drivers, these attacks are particularly dangerous due to the driver's privileged access to system resources.

If a driver reads user input into a buffer without proper size checks, an attacker can overwrite adjacent memory regions.

Mitigation

Input Validation: Always validate the length and format of input data before processing it. Use safe functions like `strncpy` instead of `strcpy` to prevent overflow.

Bounds Checking: Ensure that all buffer accesses are within the bounds of the allocated memory. Utilize compiler features like stack canaries and bounds-checking tools.

8.3 Memory Corruption

Memory corruption occurs when a driver improperly manipulates memory, leading to system instability or security vulnerabilities.

Example Attack

Improper pointer arithmetic or bounds checking in a driver can lead to memory corruption, potentially allowing arbitrary code execution.

Mitigation

Employ safe coding practices and thoroughly test driver code. Use static and dynamic analysis tools to detect memory corruption issues. Implement strict input validation and bounds checking.

8.4 Conclusion

Studying and preventing potential attacks on Linux drivers is vital for maintaining system security and stability. Drivers manage critical functions and sensitive data, and any compromise can lead to significant data breaches or system failures. Preventing denial of service attacks and other disruptions helps maintain the performance and reliability essential for user trust and operational continuity.

Furthermore, mitigating financial and reputational risks associated with security breaches is crucial. By implementing robust input validation, proper resource management, and safe coding practices, developers can significantly reduce exploitation risks. Regular updates and security patches further enhance driver security. Prioritizing the study and prevention of potential attacks on Linux drivers is key to safeguarding user data, maintaining system reliability, and ensuring overall trust in the platform.

A Linux device driver for SHA-256 crypto core in simulated RISC-V environment © 2024 by Fırat Kağıtçı, Yang Zeyu, Malaz, Barkey Demir, Shang Shanchong is licensed under CC BY-NC 4.0. To view a copy of this license, visit <https://creativecommons.org/licenses/by-nc/4.0/>