Politecnico di Torino

Facoltà di Ingegneria

Crypto Core in QEMU: installation guide

Operating systems

Authors: group 12

Luigi Palmisano (319867)
Alberto Castronovo (290238)
Giuseppe Cutrera (318954)
Matteo Carnevale (317718)

January 29, 2024

Contents

CHAPTER 1

AES-256 and crypto-core installation guide

1.1 AES256

The Advanced Encryption Standard (AES), initially known as Rijndael, was established by the U.S. National Institute of Standards and Technology (NIST) in 2001 to replace the Data Encryption Standard (DES). Developed by Belgian cryptographers Joan Daemen and Vincent Rijmen, AES is a symmetric-key algorithm, using the same key for both encryption and decryption.

NIST selected three variants of Rijndael for AES, featuring a fixed block size of 128 bits and key lengths of 128, 192, or 256 bits. AES-256, with its 256-bit key, offers the highest level of security. AES operates through a substitution–permutation network design and has a different structure compared to DES, lacking the Feistel network. The number of transformation rounds varies based on the key length: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. The U.S. government, including the NSA, has certified AES for protecting classified information up to the TOP SECRET level.

AES has a relatively straightforward algebraic framework. In 2002, a theoretical attack called the "XSL attack" was introduced but subsequent research showed the original attack to be unfeasible.

Successful attacks on AES were limited to side-channel attacks on specific implementations until May 2009. The first key-recovery attacks on full AES were published in 2011, utilizing a biclique attack. Currently, there is no known practical attack to decrypt data encrypted by AES when implemented correctly.

Side-channel attacks involve exploiting vulnerabilities in the implementation of the cipher on hardware or software systems. Notable instances include a cache-timing attack announced in April 2005 and further attacks in December 2009 utilizing differential fault analysis on hardware implementations. In November 2010, a practical approach to near real-time key recovery from AES-128 was demonstrated. In March 2016, a side-channel attack on AES implementations capable of recovering the complete 128-bit AES key in just 6–7 blocks of plaintext/ciphertext was presented. While modern CPUs with built-in hardware instructions for AES protect against timing-related side-channel attacks, it's crucial to note that the quantum resistance of AES varies with key sizes. AES-256 is considered quantum-resistant, while AES-192 and AES-128 are not, rendering them insecure in such contexts.

1.2 QEMU

QEMU, short for Quick Emulator, is an open-source virtualization software that provides hardware virtualization capabilities for various architectures. Originally developed by Fabrice Bellard, QEMU allows users to emulate and run virtual machines (VMs) on a host system. It serves as a versatile tool

that supports the emulation of a wide range of processors, including x86, ARM, PowerPC, and more, making it a valuable choice for cross-platform development, testing, and experimentation.

One of QEMU's notable features is its ability to perform full-system emulation, enabling users to run an entire operating system inside a virtual machine without modification. Additionally, QEMU supports various hardware devices, allowing for a comprehensive emulation environment. It's widely used in the fields of software development, testing, and debugging, as well as in scenarios where running software on different architectures or platforms is essential.

QEMU also provides a user-friendly command-line interface and supports the use of graphical frontends, making it accessible to users with different levels of technical expertise. The software's flexibility, combined with its open-source nature, has contributed to its popularity and its adoption in numerous projects and industries.

1.3 BUILDROOT

Buildroot is an open-source build system that simplifies and automates the process of building embedded Linux systems. Developed with a focus on simplicity and efficiency, Buildroot allows users to generate customized Linux distributions tailored to the specific requirements of embedded devices. It handles the compilation of the entire software stack, from the bootloader and kernel to user-space applications, libraries, and system utilities.

The decision to use Buildroot as a system to be simulated inside QEMU is driven by several key factors. Firstly, Buildroot provides a user-friendly and configurable environment that streamlines the often complex task of creating embedded Linux systems. Its menu-driven configuration system enables users to specify the target architecture, select software components, and customize system settings with ease.

Secondly, Buildroot is designed to be lightweight and efficient, producing minimalistic and optimized root filesystems. This is particularly advantageous for embedded systems with limited resources, ensuring that the generated system is tailored to the specific needs of the target device (ISA Risc-V in our case).

Furthermore, Buildroot integrates seamlessly with QEMU, making it an ideal choice for simulation purposes. QEMU supports emulation of various architectures, and Buildroot's ability to generate system images compatible with QEMU simplifies the process of testing and debugging embedded software in a virtualized environment. This combination facilitates rapid development cycles and thorough testing before deploying the system to actual hardware.

1.4 RISC-V

RISC-V is an open-source instruction set architecture (ISA) based on the Reduced Instruction Set Computing (RISC) principles. Unlike other proprietary ISAs, RISC-V is an open standard that is freely available for anyone to use, modify, and implement. This openness has led to its widespread adoption, particularly in the domain of embedded systems, where customization, simplicity, and openness are highly valued. Key features and aspects of RISC-V include: Modularity and Extensibility: RISC-V follows a modular and extensible design philosophy. It provides a base set of instructions, known as the RV32I base integer instruction set, and allows for the addition of optional extensions. This modularity makes RISC-V suitable for a wide range of applications, from microcontrollers to high-performance computing. Open Ecosystem: RISC-V's open nature fosters collaboration and innovation. The ISA's specifications, reference implementations, and compliance tests are publicly available, encouraging a diverse community of developers, researchers, and industry players to contribute to its evolution. Community Support: RISC-V has gained significant momentum, and a vibrant community has emerged around its development. This community actively contributes to the improvement of the

ISA, develops software tools, and collaborates on various RISC-V-based projects. Versatility: RISC-V is versatile and suitable for a broad range of applications, from embedded systems and Internet of Things (IoT) devices to high-performance computing and data centers. Its adaptability makes it a compelling choice for projects with diverse computational requirements. Industry Adoption: RISC-V has been embraced by both academia and industry. Many companies, ranging from startups to established technology giants, are exploring or implementing RISC-V-based solutions. This adoption is driven by the desire for more control, reduced licensing costs, and the ability to customize processor designs. Educational Value: RISC-V's open architecture and clean design make it an excellent choice for educational purposes. It provides a transparent and accessible platform for teaching computer architecture and embedded systems design.

1.5 Setup instructions

All the instructions are tested in a Peppermint Linux distro using a 64 bit architecture.

1.5.1 QEMU setup

Considering that we want to use a custom qemu environment with a simulated cryptocore, the first command to execute is to download the qemu project from the gitlab source by running the following:

```
git clone https://gitlab.com/qemu-project/qemu.git
```

Then we need to checkout a stable and tested version:

```
cd qemu && git checkout stable-8.1
```

Then we download all the necessary dependencies:

```
sudo apt-get install autoconf automake autotools-dev curl libmpc-dev
    libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo
    gperf libtool patchutils bc zlib1g-dev libexpat-dev git ninja-build
    libpixman-1-dev rsync sphinx glibc-source
```

Then we select the necessary configuration for 64 bit RISC-V and compile the setup:

```
./configure --target-list=riscv64-softmmu --enable-slirp --enable-debug
make
```

1.5.2 Buildroot setup

We start by downloading the source as we did with qemu:

```
git clone https://github.com/buildroot/buildroot.git
```

Then we select the riscv64 default configuration:

```
cd buildroot && make qemu_riscv64_virt_defconfig
```

Then we enable OpenSSH by running the following command and navigating to Target Packages, Networking applications and selecting openssh:

```
make menuconfig
```

Finally to compile the whole configuration:

```
make
```

We now need to make some changes to the file *start−qemu.sh* by navigating inside */buildroot/output/images/*: We have to substitute qemu-system-riscv64 at the end of the file with the full global path. In the same line because we want to connect to the local machine by ssh add what's missing after the 'id=net0,' part resulting in something like this:

```
netdev user,id=net0,hostfwd=tcp::2222-:22
```

Leave the rest before and after untouched. We can verify that the setup is working by running:

```
./start-qemu.sh
```

Loggin with username: root And quitting by pressing CTRL+A followed by pressing X

1.5.3 Device emulation setup

The following changes were made to create virtual device capable of behaving like a physical cryptocore; First thing we need to add the following lines to *qemu/hw/misc/Kconfig* in a relatively arbitrary position:

```
config CRYPTO_CORE
    bool
```

We need to copy the *crypto_core.c* file in *qemu/hw/misc/*. We add the following line to the file *qemu/hw/misc/meson.build* in an arbitrary position:

```
system_ss.add(when: 'CONFIG_CRYPTO_CORE', if_true: files('crypto_core.c'))
```

We add the following line in *qemu/hw/riscv/Kconfig* inside the "select" list:

```
select CRYPTO_CORE
```

Following the line *config RISCV_VIRT*.

Then we copy the *crypto_core.h* file in *qemu/include/hw/misc/* (different to the one from before) We modify the *qemu/include/hw/riscv/virt.h* file adding the following element to the end of the first "enum":

```
VIRT_CRYPTO_CORE
```

We also need to add a comma to the end of the one before last element to correctly compile the list. Lastly we modify the file *qemu/hw/riscv/virt.c* by adding different elements in different parts: We add the following between the includes at the beginning:

```
#include "hw/misc/crypto_core.h"
```

After the line:

```
[VIRT_PLATFORM_BUS] = {  0x4000000,     0x2000000 },
```

We add:

```
[VIRT_CRYPTO_CORE]  = {  0x8000000,          0x200 },
```

Inside the *static const MemMapEntry virt_memmap[]* definition, in order to add a memory space dedicated to the virtual *CRYPTO_CORE*, we add the following:

```
crypto_core_create(memmap[VIRT_CRYPTO_CORE].base);
```

Inside the function *virt_machine_init* just after the line $sifive\_test\_create(memmap[VIRT\_TEST].base)$;. We also declare the following function just before the line *static void* $create\_fdt(\ldots$:

```
static void create_fdt_crypto_core(RISCVVirtState *s, const MemMapEntry *memmap)
{
    MachineState *ms = MACHINE(s);
    char *nodename;
    hwaddr base = memmap[VIRT_CRYPTO_CORE].base;
    hwaddr size = memmap[VIRT_CRYPTO_CORE].size;
    nodename = g_strdup_printf("/crypto_core@%" PRIx64, base);




    qemu_fdt_add_subnode(ms->fdt, nodename);
    qemu_fdt_setprop_string(ms->fdt, nodename, "compatible", "crypto-core");
    qemu_fdt_setprop_sized_cells(ms->fdt, nodename, "cryptoreg", 2, base, 2, size);




    g_free(nodename);
}
```

Inside the function *static void* $create\_fdt(...$, just after the line $create\_fdt\_fw\_cfg(s, memmap)$; we add:

```
create_fdt_crypto_core(s, memmap);
```

After that we need to rebuild and compile qemu as a whole by repeating the commands at 1.d

1.5.4 Guest system setup

We now need to pass the files of the driver and the test file inside the guest system runned by the buildroot start-qemu.sh file. The first step inside the guest system is to run the following command in order to permit login without a password as root:

```
vi /etc/ssh/sshd_config
```

And inside we need to change the following settings also removing the # comment separator:

```
PermitRootLogin yes
PermitEmptyPasswords yes
```

Finally we restart ssh by running the following command and leaving the machine running:

```
/etc/init.d/S50sshd restart
```

We now need to obtain the $crypto - core.ko$, *test_program.o* files: Inside their own folders we worked on the driver file and the test_program and we compiled them by running *make_command.sh* inside the driver folder and running the make command inside the test folder (following the Makefile instructions). We now need to run the terminal inside each of these two folders and write the following commands (obviously for the driver folder we need to run the command relative to the driver file we obtained and the same has to be done for the test_program file):

```
scp -P 2222 crypto-core.ko root@localhost:/root/
scp -P 2222 test_program.o root@localhost:/root/
```

We now need to go back to the guest system and run the following commands:

```
chmod 777 test_program.o
chmod 777 crypto-core.ko
insmod crypto-core.ko
```

Finally inside the guest system we create an *input.txt* file used to test the *test_program* file where we can write the message to encrypt/decrypt inside. We can test if the program is working by running something like this:

```
./test_program.o KEY IV MODE FORMAT
```

For example we can run

```
./test_program.o 00112233445566778899AABBCCDDEEFF 0001020304050607 0 0
```

1.6 Mentions

https://www.n4b.it/sicurezza-it/
sicurezza-comprensione-della-crittografia-aes-256-advanced-encryption-standard/
https://en.wikipedia.org/wiki/Side-channel_attack
https://www.onoratoinformatica.it/ransomware-news-attack/crittografia-aes-256/
https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
https://en.wikipedia.org/wiki/QEMU
https://www.qemu.org/docs/master/
https://en.wikipedia.org/wiki/Buildroot
http://buildroot.org/downloads/manual/manual.html
https://www.thirtythreeforty.net/
posts/2020/01/mastering-embedded-linux-part-3-buildroot/
https://elinux.org/images/0/05/Petazzoni--buildroot_a_deep_dive_into_the_core.pdf
https://riscv.org/about/
https://it.wikipedia.org/wiki/RISC-V
https://www.allaboutcircuits.com/technical-articles/
introductions-to-risc-v-instruction-set-understanding-this
-open-instruction-set-architecture/