

Laboratoire 05 - 06 - 07 :
SSHD - File System - TPM

Département : EIE
Unité d'enseignement : MA_SeS

Auteurs : Quentin Müller & Tristan Traiber
Professeur : Jean-Roland Schuler
Classe : C2 - C3
Date : 8 avril 2022

Table des matières

1	Introduction	3
2	Laboratoire 5 sshd	4
2.1	Validation de la signature de openssh	4
2.2	Configuration du paquet pour l'installation	5
2.2.1	Configuration pour l'environnement Intel	5
2.2.2	Configuration pour l'environnement du NanoPi	5
2.3	Copie des fichiers sur le NanoPi et création des clés	7
2.4	Reconfiguration du paquet OpenSSH	7
2.5	Modification du nom de version de openssh	8
3	Laboratoire 6 File System	10
3.1	Question 1 : EXT4	10
3.1.1	Comment le kernel connaît l'emplacement du rootfs?	10
3.1.2	Monter la première partition de la carte SD sur le NanoPi	10
3.1.3	Trouver le nombre mineur et majeur du <i>node file</i> de la carte microSD	11
3.2	Question 2 : filesystem	12
3.2.1	Création de deux nouvelles partitions	15
3.2.2	Attribution des <i>files sytems</i>	18
3.3	Question 2.3 - Mesure des performances de chaque partition	20
3.3.1	Programme C d'écriture de fichier	20
3.3.2	Mesures et résultats	22
3.4	Question 3 : LUKS, cryptsetup, dmccrypt	23
3.4.1	Question 3.1 : Création d'une partition LUKS	23
3.4.2	Question 3.2 : Partition LUKS	24
3.4.3	Question 3.3 : Rootfs dans la partition LUKS	29
3.5	Question 4 : Initramfs	31
3.6	Partition Initramfs-LUKS	37
4	Laboratoire 7 TPM	46
4.1	Installation des outils TPM2	46
4.2	Question 1 - Créer des <i>"load-save primary keys"</i>	47
4.3	Question 2 - Créer des <i>load-save child keys</i>	49
4.4	Question 3 - Décrypter avec TPM	52
4.5	Question 4 - Politique PCR	53
4.5.1	U-boot check linux	53
4.5.2	Installation d'un nouveau kernel linux	55
5	Conclusion	56
6	Signatures	56

1 Introduction

L'objectif de cette deuxième série de laboratoires est d'appréhender et de mettre en pratique plusieurs sujets relatifs au partage de fichier, au contrôle de certificat d'authenticité de logiciel, au cryptage de données, au cryptage de partition ainsi que l'utilisation de TPM (*"Trusted Platform Module"*).

Ce rapport est composé de trois laboratoires couvrant les sujets suivants :

1. Laboratoire 5 - sshd
 - Configuration et utilisation de OpenSSH qui est un ensemble d'outils informatiques libres permettant des communications sécurisées sur un réseau à l'aide du protocole SSH
2. Laboratoire 6 - File System
 - Compréhension générale du système de fichier de type **EXT4**
 - Discrimination des différents types de système de fichier disponible sur Linux et illustration de leur type de fonctionnement
 - Mesure et comparaison de performances de plusieurs types de systèmes de fichier en réalisant une écriture de plusieurs fichiers
 - Découverte et utilisation de **LUKS**, **cryptsetup** et **dmccrypt**
 - Configuration et utilisation d'un **initramfs**
 - Création et utilisation d'une partition **LUKS** crypté et démarré à l'aide d'un **initramfs**
3. Laboratoire 7 TPM (*"Trusted Platform Module"*)
 - Installation des outils **TPM2**
 - Configuration de clés primaires (*"primary keys"*) et de clés enfant (*"child keys"*)
 - Décryptage à l'aide d'un TPM
 - Introduction aux politiques des registres de configuration de plateforme (PCR)

2 Laboratoire 5 sshd

2.1 Validation de la signature de openssh

Lors du téléchargement d'un logiciel sur internet, il est fortement indiqué de contrôler l'authentification du créateur du logiciel ou alors de son publicateur. Ceci afin de vérifier l'authenticité du logiciel et de prévenir l'utilisation de logiciels malveillants.

Dans notre cas, on souhaite vérifier l'authenticité d'une application, car celle-ci obtiendra des droits *super user* (`sudo`) lors de son installation finale sur notre machine. Si le logiciel a été modifié de manière frauduleuse ou par des personnes malintentionnées nous courrons le risque de donner des droits administrateurs à leur application. La vérification du logiciel est faite par le biais d'une commande native de Linux `gpg --verify`.

```
1 gpg --verify "Logiciel à vérifié"
2 gpg --keyserver keyserver.ubuntu.com --search-keys "Clé public
  de l'annonceur"
```

Listing 1 – Commandes pour la vérification de la signature d'un logiciel

La commande de la ligne 1 du listing 1 sert à vérifier que le logiciel téléchargé correspond bien à la version postée par son propriétaire, lui-même authentifié. Il est cependant très probable que la première fois que l'on lance cette commande le résultat soit négatif. Ceci est dû au fait que nous ne possédons pas la clé publique de l'annonceur. Pour remédier à ce problème, il est nécessaire d'utiliser la deuxième commande. Cette commande réalise une recherche sur un site, officiel et fiable, de la clé public du propriétaire du logiciel. Une fois cette étape exécutée, on peut à nouveau exécuter la première commande et vérifier l'authenticité du logiciel.

```
[lmi@fedora Downloads]$ gpg --verify openssh-8.8p1.tar.gz.asc
gpg: assuming signed data in 'openssh-8.8p1.tar.gz'
gpg: Signature made Sun 26 Sep 2021 16:07:27 CEST
gpg: using RSA key 7168B983815A5EEF59A4ADFD2A3F414E736060BA
gpg: Can't check signature: No public key
[lmi@fedora Downloads]$ gpg --keyserver keyserver.ubuntu.com --search-keys 7168B983815A5EEF59A4ADFD2A3F414E736060BA
gpg: data source: http://162.213.33.9:11371
(1) Damien Miller <djm@mindrot.org>
    4096 bit RSA key 2A3F414E736060BA, created: 2021-01-01
Keys 1-1 of 1 for "7168B983815A5EEF59A4ADFD2A3F414E736060BA". Enter number(s), N)ext, or Q)uit > 1
gpg: key 2A3F414E736060BA: public key "Damien Miller <djm@mindrot.org>" imported
gpg: Total number processed: 1
gpg: imported: 1
[lmi@fedora Downloads]$ gpg --verify openssh-8.8p1.tar.gz.asc
gpg: assuming signed data in 'openssh-8.8p1.tar.gz'
gpg: Signature made Sun 26 Sep 2021 16:07:27 CEST
gpg: using RSA key 7168B983815A5EEF59A4ADFD2A3F414E736060BA
gpg: Good signature from "Damien Miller <djm@mindrot.org>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg: There is no indication that the signature belongs to the owner.
Primary key fingerprint: 7168 B983 815A 5EEF 59A4 ADFD 2A3F 414E 7360 60BA
```

FIGURE 1 – Vérification de la signature de OpenSSH

Sur la figure 1, nous pouvons observer les trois commandes exécutées, comme expliqué dans le paragraphe précédent. De plus, il est possible de constater sur la figure, que le logiciel téléchargé possède une signature correcte.

2.2 Configuration du paquet pour l'installation

Une fois que le logiciel authentifié a été téléchargé sur l'ordinateur, il est nécessaire de paramétrer le paramétrer en vue de sa future installation. Pour ce faire, nous devons respecter ces trois points.

1. Activer le hardening
2. Changer le répertoire d'installation
3. Choisir l'hôte sur lequel le logiciel sera installé

2.2.1 Configuration pour l'environnement Intel

Tout d'abord nous allons réaliser une configuration pour une installation sur une machine de type Intel, soit notre PC. L'option de hardening étant activée par défaut, il suffit juste de spécifier le dossier dans lequel les fichiers seront installés à l'aide du mot clé `-prefix`.

```
1 ./configure --prefix=/home/lmi/SeS/Labo05_ssh/intel
```

Une fois le `make install` fini, nous pouvons alors vérifier si le fichier `bin` du `ssh` a bien été strippé. La figure 2 illustre le contrôle du fichier `ssh`.

```
[lmi@fedora Labo05_ssh]$ cd intel/bin/
[lmi@fedora bin]$ file ssh
ssh: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
 /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=24e36946b26f93812e6d2c3bae172e7e3ba0af76, for G
NU/Linux 3.2.0, stripped
[lmi@fedora bin]$
```

FIGURE 2 – Vérification du strip après installation de OpenSSH sur intel

2.2.2 Configuration pour l'environnement du NanoPi

La configuration pour le NanoPi est plus complexe, car il faut préciser un hôte différent à l'aide de l'option `-host`. Il faut aussi préciser le dossier depuis lequel il sera exécuté sur le NanoPi lors de son utilisation, cela à l'aide de l'option `sysconfd`. Une fois que la commande `configure` a été exécutée, il faut encore modifier le programme de strip dans le fichier du Makefile. Les instructions complètes sont illustrées dans le listing suivant :

```
1 ./configure --prefix=/home/lmi/SeS/Labo05_ssh/NanoPi --host=aarch64
  -none-linux-gnu --sysconfd=/root/sshd
2
3 sed -i 's/STRIP_OPT=-s/STRIP_OPT=-s --strip-program=aarch64-none-
  linux-gnu-strip/g' Makefile
```

Une fois que c'est deux commandes ont été exécutées, il est enfin possible d'installer le programme pour le NanoPi avec l'instruction `make install`.

L'arborescence du dossier ainsi installé est illustré sur la figure 3. Cette figure a été réalisée à l'aide de la commande `tree`.

```
[lmi@fedora Labo05_ssh]$ cd nanopi/
[lmi@fedora nanopi]$ tree
.
├── bin
│   ├── scp
│   ├── sftp
│   ├── ssh
│   ├── ssh-add
│   ├── ssh-agent
│   ├── ssh-keygen
│   └── ssh-keyscan
├── etc
│   ├── moduli
│   ├── ssh_config
│   └── sshd_config
├── libexec
│   ├── sftp-server
│   ├── ssh-keysign
│   ├── ssh-pkcs11-helper
│   └── ssh-sk-helper
├── sbin
│   └── sshd
├── share
└── man
    ├── man1
    │   ├── scp.1
    │   ├── sftp.1
    │   ├── ssh.1
    │   ├── ssh-add.1
    │   ├── ssh-agent.1
    │   ├── ssh-keygen.1
    │   └── ssh-keyscan.1
    ├── man5
    │   ├── moduli.5
    │   ├── ssh_config.5
    │   └── sshd_config.5
    └── man8
        ├── sftp-server.8
        ├── sshd.8
        ├── ssh-keysign.8
        ├── ssh-pkcs11-helper.8
        └── ssh-sk-helper.8

9 directories, 30 files
[lmi@fedora nanopi]$
```

FIGURE 3 – Arborescence du dossier d'installation pour le NanoPi

Comme les options du programme de stripage ont été modifiées, nous pouvons aussi contrôler le stripage de l'installation réalisée pour l'environnement du NanoPi. Comme illustré sur la figure 4

```
[lmi@fedora bin]$ file ssh
ssh: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (SYSV), dynamically linked, inter
preter /lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0, stripped
```

FIGURE 4 – Vérification du stripage après installation de OpenSSH pour le NanoPi

2.3 Copie des fichiers sur le NanoPi et création des clés

Une fois l'installation terminée, il est possible de transférer des fichiers sur le NanoPi. Les fichiers utiles au bon fonctionnement de OpenSSH sont les suivants : *sshd*, *ssh-keygen*, *moduli*, *sshd_config* ainsi que le fichier *config_pi.sh* qui permet de créer les dossiers et les utilisateurs nécessaires à l'exécution du logiciel.

Le listing 2 illustre en détail les commandes utilisées.

```
1 #!/bin/sh
2 echo "-----Create directory-----"
3 mkdir /home
4 mkdir /home/sshd
5 echo "-----Create user -----"
6 adduser sshd -h /home/sshd # ajouter une mot de passe 'ilovelmi'
7 echo "-----Stop dropbear-----"
8 /etc/init.d/S50dropbear stop #couper l'autre soft
9 echo "-----Create key-----"
10 cd /root/sshd
11
12 yes '' | ./ssh-keygen -t rsa -b 4096 -f rsa_4096 -q -N ""
13 yes '' | ./ssh-keygen -t dsa -b 1024 -f dsa_1024 -q -N ""
14 yes '' | ./ssh-keygen -t ecdsa -b 521 -f ecdsa_512 -q -N ""
15 yes '' | ./ssh-keygen -t ed25519 -b 256 -f ed25519_256 -q -N ""
16 # Generate host keys
17 ./ssh-keygen -A
18
19 cat ssh_host_ed25519_key.pub
20 echo "-----Start ssh-----"
21 mkdir /var/empty
22 /root/sshd/sshd
```

Listing 2 – Listing du script d'initialisation du ssh

2.4 Reconfiguration du paquet OpenSSH

Après avoir rapidement tester le bon fonctionnement du logiciel OpenSSH, il est nécessaire d'apporter quelques modifications afin d'améliorer la sécurité.

Les modifications suivantes sont réalisées :

- Le SSHD utilise uniquement l'IPV4
- Désactiver la redirection de port (port forwarding)
- Activer les algorithmes de cryptage et de hachage suivants : *Ciphers aes256-cbc, aes256-ctr, aes128-cbc, hmac-sha-256* et *hmac-sha1*
- Désactiver le login *root*
- Indication de connexion à l'aide d'une bannière

Les modifications à réaliser sont illustrées dans le listing suivant :

```
1 #AddressFamily any
2 AddressFamily inet
3
4 #AllowTcpForwarding yes
5 AllowTcpForwarding no
6
7 #PermitRootLogin prohibit-password
8 PermitRootLogin no
9
10 #Ajouter
11 Ciphers aes256-cbc,aes256-ctr,aes128-cbc,hmac-sha-256,hmac-sha1
12
13 #Banner none
14 Banner "Labo05 SSH Quentin et Tristan"
```

Il est possible de constater que les modifications ont bien été implémentées en réalisant une connexion à l'aide de l'utilisateur *root*.

Cette vérification est illustrée par la figure 5 :

```
[lmi@fedora Labo05_ssh]$ ssh root@192.168.0.11
root@192.168.0.11's password:
Permission denied, please try again.
```

FIGURE 5 – Connections ssh en mode root refusée

Nous pouvons effectivement observer que la connexion à l'aide de l'utilisateur *root* n'est pas autorisée.

La connexion est tout de même possible à l'aide de l'utilisateur *sshd* comme le montre la figure 6 :

```
[lmi@fedora Labo05_ssh]$ ssh sshd@192.168.0.11
sshd@192.168.0.11's password:
$ pwd
/home/sshd
$ whoami
sshd
$ □
```

FIGURE 6 – Connections ssh en mode sshd

2.5 Modification du nom de version de openssh

La figure 7 illustre la dernière étape de ce laboratoire. Malgré les modifications réalisées, il persiste une faille avec ce logiciel. Avec la commande `nmap -sV -p 22`, il est possible d'afficher tous les ports TCP ouverts ainsi que le logiciel utilisé par ces ports. Ceci est un problème, car n'importe qui peut avoir accès à cette information et si cette version du logiciel comporte une faille connue alors il est possible qu'elle soit exploitée. C'est pour cela que nous allons essayer de supprimer cette information de la liste des données visibles de *nmap*.


```
[lmi@fedora Labo05_ssh]$ nmap -sV -p 22 192.168.0.11
Starting Nmap 7.80 ( https://nmap.org ) at 2022-01-07
15:32 CET
Nmap scan report for 192.168.0.11
Host is up (0.0013s latency).

PORT      STATE SERVICE VERSION
22/tcp    open  ssh      OpenSSH 8.8 (protocol 2.0)

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 0.78 s
```

FIGURE 7 – *nmap* de l'IP du NanoPi

Pour ce faire, il est nécessaire d'aller chercher cette information dans le fichier exécutable de OpenSSH afin de supprimer la mention de la version du logiciel. Ce qui donne le résultat obtenu dans la figure 8.

```
[lmi@fedora Labo05_ssh]$ nmap -sV -p 22 192.168.0.11
Starting Nmap 7.80 ( https://nmap.org ) at 2022-01-10 10:39 CET
Nmap scan report for 192.168.0.11
Host is up (0.0013s latency).

PORT      STATE SERVICE VERSION
22/tcp    open  ssh      (protocol 2.0)
1 service unrecognized despite returning data. If you know the service/version, please
submit the following fingerprint at https://nmap.org/cgi-bin/submit.cgi?new-service :
SF-Port22-TCP:V=7.80%I=7%D=1/10%Time=61DBFEE9%P=x86_64-redhat-linux-gnu%(
SF:NULL,A,"SSH-2\0-\r\n");

Service detection performed. Please report any incorrect results at https://nmap.org/s
ubmit/ .
Nmap done: 1 IP address (1 host up) scanned in 6.92 seconds
```

FIGURE 8 – *nmap* de l'IP du NanoPi sans la version de OpenSSH

3 Laboratoire 6 File System

Ce laboratoire se concentre sur les aspects concernant la configuration et l'utilisation de différent type de système de fichier (*file system*) disponible sur Linux.

3.1 Question 1 : EXT4

Dans cette partie nous allons répondre à quelques questions générales concernant le système de fichier de type **EXT4**. Ce système est l'un des plus répandus pour Linux et est celui que nous avons configuré tout au début du cours de SeS.

3.1.1 Comment le kernel connaît l'emplacement du rootfs ?

Plus particulièrement comment le kernel sait que *rootfs* se trouve dans la seconde partition de la carte microSD ?

Il est possible de trouver cette information sur la VM. Dans le dossier `cd workspace/nano/buildroot/output/images/`, on peut trouver le fichier *boot.scr*. Dans ce fichier, la première ligne nous indique que la *root* se trouve dans `/dev/mmcblk0p2`, soit dans la deuxième partition de la carte microSD externe.

L'emplacement et le contenu du fichier *boot.scr* est illustré sur la figure 9.

```
[lmi@fedora ~]$ cd workspace/nano/buildroot/output/images/
[lmi@fedora images]$ ls
bl31.bin  brcm      nanopi-neo-plus2.dtb  rootfs.tar          sunxi-spl.bin
boot.scr  extlinux  rootfs.ext2           sdcard.img          u-boot.bin
boot.vfat Image    rootfs.ext4           sun50i-h5-nanopi-neo-plus2.dtb  u-boot.itb
[lmi@fedora images]$ cat boot.scr
'V0400a0900cc00setenv bootargs console=ttyS0,115200 earlyprintk root=/dev/mmcblk0p2 rootwait

ext4load mmc 0 $kernel_addr_r Image
ext4load mmc 0 $fdt_addr_r nanopi-neo-plus2.dtb

booti $kernel_addr_r - $fdt_addr_r
```

FIGURE 9 – Capture d'écran du contenu du fichier boot.scr

3.1.2 Monter la première partition de la carte SD sur le NanoPi

Afin de monter la première partition de la carte microSD, il faut réaliser les deux commandes suivantes :

1. Sur le NanoPi exécuter la commande `mkdir /mnt`
2. Exécuter `mount /dev/mmcblk0p1 /mnt`

3.1.3 Trouver le nombre mineur et majeur du *node file* de la carte microSD

Sur la figure 10, nous pouvons observer de manière simplifiée comment le userspace peut avoir accès à un périphérique. Pour ceci, il existe les *nodes files* qui sont les emplacements des périphériques, comme la carte microSD qui contient tous les fichiers du NanoPi.

Les *nodes files* sont composés de trois informations principales : NOM, NUM Majeur, NUM Mineur. Ces trois informations sont visibles sur la figure 11. Le nom sert à mieux identifier leur fonction. Le nombre majeur sert à identifier une fonction globale, comme les gestionnaires de carte SD ou tous les nodes files `/dev/mmc*` ont le même nombre majeur 179. Par contre, le nombre mineur est une information plus précise sur quelle partie de la carte microSD on se trouve. On peut voir sur la figure 11 que chaque node file a un nombre mineur différent. Car il est possible de passer d'une partition de type ext4 à une autre de type f2fs par exemple et il faut donc d'autre driver pour utiliser ces partitions.

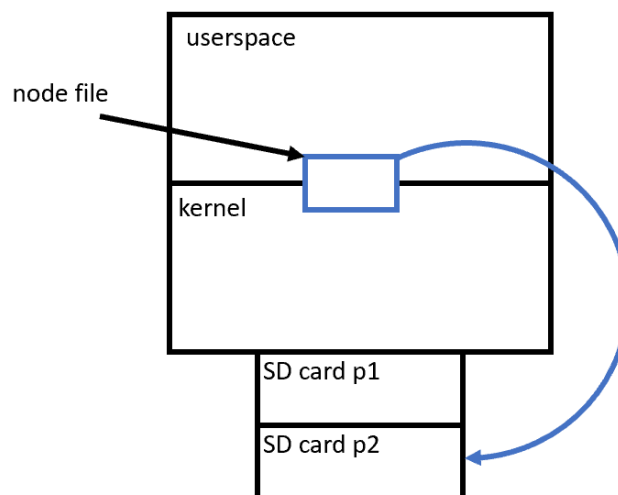


FIGURE 10 – Schéma de l'utilisation des nodes files

Pour trouver ces deux nombres, il faut d'abord connaître le nom du *node file* lié à la carte microSD. Pour cela, il est possible d'utiliser la commande `ls /dev/root -al` qui permet d'afficher le nom *du node file* du *rootfs* qui se trouve sur la carte microSD.

Une fois cette information trouvée, il ne reste plus qu'à utiliser la commande `ls -al /dev/mmc*` qui permet d'afficher toutes les informations liées à la carte microSD.

Le résultat de cette recherche est illustré sur la figure 11 :

```
# mount
/dev/root on / type ext4 (rw,relatime)
devtmpfs on /dev type devtmpfs (rw,relatime,size=221372k,nr_inodes=55343,mode=755)
proc on /proc type proc (rw,relatime)
devpts on /dev/pts type devpts (rw,relatime,gid=5,mode=620,ptmxmode=666)
tmpfs on /dev/shm type tmpfs (rw,relatime,mode=777)
tmpfs on /tmp type tmpfs (rw,relatime)
tmpfs on /run type tmpfs (rw,nosuid,nodev,relatime,mode=755)
sysfs on /sys type sysfs (rw,relatime)
# ls -al /dev/mmc*
brw-rw---- 1 root root 179, 0 Jan 1 00:00 /dev/mmcblk0
brw-rw---- 1 root root 179, 1 Jan 1 00:00 /dev/mmcblk0p1
brw-rw---- 1 root root 179, 2 Jan 1 00:00 /dev/mmcblk0p2
brw-rw---- 1 root root 179, 32 Jan 1 00:00 /dev/mmcblk2
brw-rw---- 1 root root 179, 64 Jan 1 00:00 /dev/mmcblk2boot0
brw-rw---- 1 root root 179, 96 Jan 1 00:00 /dev/mmcblk2boot1
brw-rw---- 1 root root 179, 33 Jan 1 00:00 /dev/mmcblk2p1
brw-rw---- 1 root root 179, 34 Jan 1 00:00 /dev/mmcblk2p2
# ls /dev/root
/dev/root
# ls /dev/root -al
lrwxrwxrwx 1 root root 9 Jan 1 00:00 /dev/root -> mmcblk0p2
# █
```

FIGURE 11 – Résultat de la commande `ls /dev/root -al`

Sur la figure 11, nous pouvons observer les informations nécessaires pour répondre à la question. Le nombre majeur est **179** et le mineur est **0** pour la carte microSD puis mineur **1** et le majeur **2** pour les deux partitions *mmcblk0p1* et *mmcblk0p2*.

3.2 Question 2 : filesystem

Il existe pour les systèmes embarqués deux catégories de système de fichier :

1. Volatil : mémoire **RAM**

1.1 TMPFS

1.2 InitRAMFS

2. Persistent : mémoire **FLASH** (NOR ou NAND)

2.1 MTD : Memory Technology Device

- JFFS2 : Journaling Flash File System 2
- YAFFS2 : Yet another Flash File System 2
- SQUASHFS
- UBI
 - Glubi : MTD-based FS (JFFS2, SQUASHFS, etc)
 - UBIFS : Unsorted Block Image File System

2.2 MMC/SD-Card : Multi-Media-Card / Secure Digital

- EXT3 / EXT4 : Linux Journaling File System
- FAT : MS-DOS File System
- BTRFS : B-Tree File System
- F2FS : Flash Friendly File System

Dans cette partie du laboratoire, nous avons décidé de tester le système de fichier **EXT4FS** qui est un système **journalisé** ("Journalized FS"), **BTRFS** qui est un système **B-Tree/CoW** et **F2FS** qui est un système avec structure **Log**.

Les **systèmes journalisés** réalisent (comme le nom l'indique) un suivi complet de toutes les modifications réalisées dans un journal situé dans un espace dédié. Cela rend ainsi possible la restauration de système corrompu.

Le comportement correspondant à ce type de système de fichier est la suivante :

1. Les modifications sont d'abord enregistrées dans le journal
2. Les modifications préalablement enregistrés dans le journal sont ensuite appliquées au système
3. Si une corruption de données intervient : Le système de fichier décide alors de garder ou non la modification en fonction du contenu du journal
 - 3.1 Si les données concernant la modification sont consistantes :
Le système applique à nouveau la modification au système
 - 3.2 Si les données concernant la modification ne sont pas consistantes :
Le système abandonne la modification du système et du journal

Les **systèmes B-Tree/CoW** sont des structures de données organisées en arbre binaire. Le **CoW** (Copy on Write) est utilisé pour garantir qu'aucune corruption n'intervienne durant l'exécution du système.

Le comportement de ce type de système de fichier est décrit dans la figure 12 :

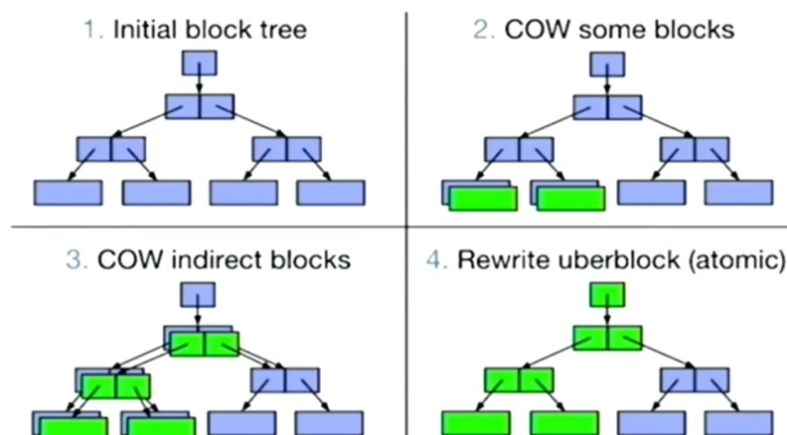


FIGURE 12 – Séquence d'écriture des systèmes de fichier de type B-Tree/CoW - source : *File Systems - Jean-Roland Schuler*

Il est possible de constater sur la figure 12 le comportement suivant :

1. Le stockage original n'est jamais modifié. Lorsqu'une demande de modification intervient, les données sont écrites dans une autre zone de stockage
2. Le stockage original est préservé jusqu'à ce qu'une modification soit appliqué

3. Si une interruption intervient durant l'écriture de la nouvelle zone de stockage, le stockage original est utilisé
4. Une fois l'écriture complète de la modification dans la nouvelle zone de stockage est terminée. Cette nouvelle zone est superposée à la zone de stockage original (Processus atomique)

Finalement les systèmes de fichier avec **structure Log** utilisent le support de stockage (mémoire) comme un buffer circulaire. Tout nouveau bloc (modification) est écrit à la fin du buffer. Ces systèmes sont souvent utilisés sur des supports persistants (flash) car son comportement crée naturellement une répartition de l'usure des cellules de la mémoire (wear-leveling). Cette structure est une forme particulière de **CoW** (Copy on Write).

Le comportement de ce type de système de fichier est décrit dans la figure 13 :

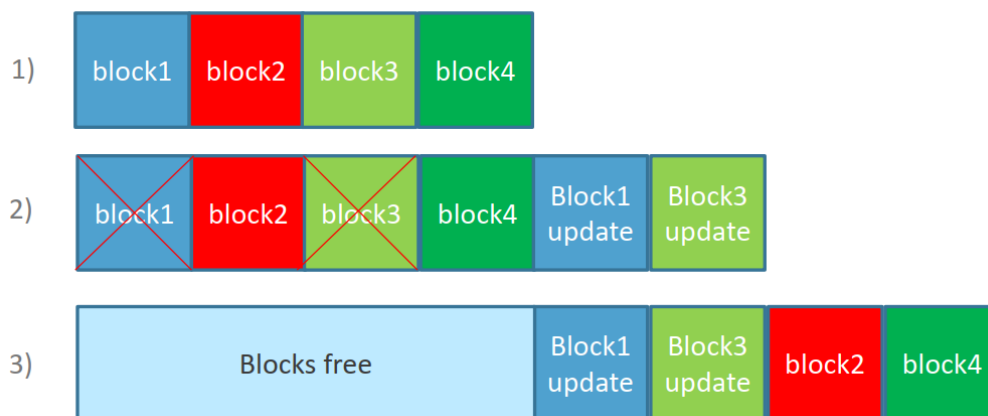


FIGURE 13 – Séquence d'écriture des systèmes de fichier de type Log - source : *File Systems - Jean-Roland Schuler*

Il est possible de constater sur la figure 13 le comportement suivant :

1. État initial du système
2. Les blocs 1 et 3 sont modifiés, les anciennes versions de ces blocs sont ignorées
3. Une copie des blocs non modifiés est réalisée à la suite des blocs modifiés et l'espace inutilisé est libéré.

Le but de cette partie du laboratoire est de tester les performances de ces systèmes en comparant le temps d'exécution d'un programme qui crée un grand nombre de petits fichiers et un seul grand fichier. Pour pouvoir faire le test, il est d'abord nécessaire de créer deux nouvelles partitions. Une de type **BTRFS** et une de type **F2Fs**.

3.2.1 Création de deux nouvelles partitions

Afin de créer deux nouvelles partitions, il est possible d'utiliser soit les fonctionnalités de la commande `parted` ou bien ceux de la commande `fdisk`. Dans cette partie du laboratoire nous avons décidé d'utiliser `fdisk` car nous avons déjà pu expérimenter l'utilisation de la commande `parted` dans les précédents laboratoires.

Avant de commencer la création des nouvelles partitions, il est nécessaire de clarifier plusieurs points. Premièrement, il faut choisir le type de filesystem que nous voulons créer. Dans notre cas la nouvelle partition que nous voulons créer correspond à la troisième partition de la carte microSD. Nous voulons que cette partition possède un filesystem de type `btrfs`. Nous voulons aussi que cette partition possède une taille de **400MB**. La commande `fdisk` permet de créer des partitions et d'en définir la taille en indiquant l'adresse de début de notre partition *First sector* et l'adresse de fin *Last sector*. La taille de la partition est donc donné par la formule :

$$Partition_{size} = Last_{sector} - First_{sector} \quad (1)$$

Les secteurs (*sectors*) correspondent à des paquets de 512 Bytes. Afin définir correctement l'emplacement de notre partition, il est nécessaire de respecter les points suivants :

- La partition ne doit pas recouvrir un espace déjà utilisé
- Le début de la partition (*First sector*) doit être un multiple de 2^{20} (**1'048'576 Bytes**). Cela pour garantir de meilleures performances

Afin de répondre à ces spécifications, nous avons d'abord identifié la fin de la partition numéro 2 (*rootfs*). Son *Last sector* vaut 4'358'143. Nous avons ensuite incrémenté cette valeur de 1 puis nous avons arrondi sa valeur au prochain multiple de 2^{20} . Ce qui donne la formule suivante :

$$First_{sector} = \lceil \frac{4'358'143 + 1}{2^{20}} \rceil \cdot 2^{20} = 5'242'880 \quad (2)$$

Une fois le *First sector* déterminé, il reste à définir la valeur du *Last sector* en ajoutant 400MB à la valeur du *First sector*. Ce qui donne :

$$Last_{sector} = First_{sector} + (400 \cdot 2048) - 1 = 6'062'079 \quad (3)$$

Si nous contrôlons la taille de la partition définie à l'aide de l'équation 1, nous obtenons :

$$Partition_{size} = \frac{Last_{sector} - First_{sector}}{2048} + 1 = 400MB \quad (4)$$

Nous pouvons maintenant créer la nouvelle partition à l'aide de la commande `fdisk` en procéder de la manière suivante :

1. Contrôler le nom du disk que l'on souhaite utiliser à l'aide de la commande `hwinfo --block --short`
2. Créer une nouvelle partition à l'aide de la commande `sudo fdisk /dev/sdb` (`sdb` correspond au disk que nous voulons utiliser)
3. Taper `n` pour créer une nouvelle partition
4. Taper `p` pour choisir une partition *primary*
5. Indiquer le numéro de la partition en tapant `3`
6. Indiquer le numéro de *First sector* avec `5242880`
7. Indiquer le numéro du *Last sector* avec `6062079`

La figure 14 illustre la création de la partition 3 à l'aide de `fdisk` :

```
[lmi@fedora buildroot]$ sudo fdisk /dev/sdb
[sudo] password for lmi:

Welcome to fdisk (util-linux 2.36.2).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.


Command (m for help): n
Partition type
  p   primary (2 primary, 0 extended, 2 free)
  e   extended (container for logical partitions)
Select (default p): p
Partition number (3,4, default 3): 3
First sector (2048-31176703, default 2048): 5242880
Last sector, +/-sectors or +/-size{K,M,G,T,P} (5242880-31176703, default 31176703): 6062080

Created a new partition 3 of type 'Linux' and of size 400 MiB.
```

FIGURE 14 – Création de la partition 3 à l'aide de `fdisk`

Il faut noter que cette capture possède une petite erreur. En effet, nous avons oublié de soustraire 1 à la valeur du *Last sector*. Cette erreur a été corrigée par la suite.

Nous avons ensuite procédé de la même manière pour créer la partition 4. En utilisant les mêmes formules vu précédemment nous avons obtenus les valeurs de *First sector* et de *Last sector* suivantes :

$$First_{sector} = \lceil \frac{6'062079 + 1}{2^{20}} \rceil \cdot 2^{20} = 6291456 \quad (5)$$

$$Last_{sector} = First_{sector} + (400 \cdot 2048) - 1 = 7110655 \quad (6)$$

$$Partition_{size} = \frac{Last_{sector} - First_{sector}}{2048} + 1 = 400MB \quad (7)$$

La figure 15 illustre la création de la partition 4 à l'aide de [fdisk](#) :

```
[lmi@fedora ~]$ sudo fdisk /dev/sdb

Welcome to fdisk (util-linux 2.36.2).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.


Command (m for help): n
Partition type
   p   primary (3 primary, 0 extended, 1 free)
   e   extended (container for logical partitions)
Select (default e): p

Selected partition 4
First sector (2048-31176703, default 2048): 6291456
Last sector, +/-sectors or +/-size{K,M,G,T,P} (6291456-31176703, default 31176703): 7110655

Created a new partition 4 of type 'Linux' and of size 400 MiB.

Command (m for help): w
The partition table has been altered.
Syncing disks.
```

FIGURE 15 – Création de la partition 4 à l'aide de **fdisk**

Une fois les deux nouvelles partitions créées, il est encore nécessaire de leur attribuer leur **file system**.

3.2.2 Attribution des *files sytems*

Comme mentionné dans les points précédant, nous avons décidé d'utiliser les *files systems* **btrfs** et **f2fs**. Afin de les utiliser, il est nécessaire de les installer à l'aide de la commande `sudo dnf install <FS type>`. Soit dans notre cas `sudo dnf install btrfs` et `sudo dnf install f2fs`.

La figure 16 illustre l'installation de **btrfs** et **f2fs** :

```
[lmi@fedora buildroot]$ sudo dnf install btrfs-progs.x86_64
[sudo] password for lmi:
Fedora 34 - x86_64 - Updates                27 kB/s | 16 kB    00:00
Fedora 34 - x86_64 - Updates                1.6 MB/s | 5.8 MB  00:03
Fedora Modular 34 - x86_64 - Updates       208 kB/s | 21 kB   00:00
Fedora Modular 34 - x86_64 - Updates       340 kB/s | 509 kB  00:01
Visual Studio Code                        16 kB/s | 3.0 kB   00:00
Visual Studio Code                        18 MB/s | 20 MB    00:01
Last metadata expiration check: 0:00:01 ago on Mon 15 Nov 2021 15:00:11 CET.
Package btrfs-progs-5.14.2-1.fc34.x86_64 is already installed.
Dependencies resolved.
Nothing to do.
Complete!
[lmi@fedora buildroot]$ sudo dnf install f2fs-tools.x86_64
Last metadata expiration check: 0:01:09 ago on Mon 15 Nov 2021 15:00:11 CET.
Dependencies resolved.
=====
Package                Architecture      Version           Repository        Size
=====
Installing:
f2fs-tools           x86_64           1.14.0-2.fc34    fedora            243 k
=====
Transaction Summary
=====
Install 1 Package

Total download size: 243 k
Installed size: 622 k
Is this ok [y/N]: y
Downloading Packages:
f2fs-tools-1.14.0-2.fc34.x86_64.rpm        1.8 MB/s | 243 kB    00:00
-----
Total                                       355 kB/s | 243 kB    00:00
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing      :                                1/1
  Installing     : f2fs-tools-1.14.0-2.fc34.x86_64 1/1
  Running scriptlet: f2fs-tools-1.14.0-2.fc34.x86_64 1/1
  Verifying      : f2fs-tools-1.14.0-2.fc34.x86_64 1/1

Installed:
  f2fs-tools-1.14.0-2.fc34.x86_64

Complete!
[lmi@fedora buildroot]$
```

FIGURE 16 – Installation de **btrfs** et **f2fs**

Pour formater les partitions, il est nécessaire d'utiliser la commande `sudo mkfs.<FS type> /dev/sdb<partition number> -L <partition label>`. Soit dans notre cas `sudo mkfs.btrfs /dev/sdb3 -L btrfs` et `sudo mkfs.f2fs/dev/sdb4 -L f2fs`.

La figure 17 et la figure 18 illustre le formatage des partitions 3 (**btrfs**) et 4 (**f2fs**) :

```
[lmi@fedora ~]$ sudo mkfs.btrfs /dev/sdb3 -L btrfs
btrfs-progs v5.14.2
See http://btrfs.wiki.kernel.org for more information.

Performing full device TRIM /dev/sdb3 (400.00MiB) ...
Label:                btrfs
UUID:                 376b22c8-1836-4997-95a5-7bebbd687c95
Node size:            16384
Sector size:          4096
Filesystem size:      400.00MiB
Block group profiles:
  Data:                single                8.00MiB
  Metadata:            DUP                  32.00MiB
  System:              DUP                   8.00MiB
SSD detected:         no
Zoned device:         no
Incompat features:    extref, skinny-metadata
Runtime features:
Checksum:             crc32c
Number of devices:    1
Devices:
   ID     SIZE  PATH
   1    400.00MiB /dev/sdb3
```

FIGURE 17 – Formatage de la partition 3

```
[lmi@fedora ~]$ sudo mkfs.f2fs /dev/sdb4 -l f2fs

F2FS-tools: mkfs.f2fs Ver: 1.14.0 (2020-08-24)

Info: Disable heap-based policy
Info: Debug level = 0
Info: Label = f2fs
Info: Trim is enabled
Info: [/dev/sdb4] Disk Model: STORAGE DEVICE
Info: Segments per section = 1
Info: Sections per zone = 1
Info: sector size = 512
Info: total sectors = 819200 (400 MB)
Info: zone aligned segment0 blkaddr: 512
Info: format version with
      "Linux version 5.15.4-101.fc34.x86_64 (mockbuild@bkernel02.iad2.fedoraproject.org) (gcc (G
      CC) 11.2.1 20210728 (Red Hat 11.2.1-1), GNU ld version 2.35.2-6.fc34) #1 SMP Tue Nov 23 18:5
      8:50 UTC 2021"
Info: [/dev/sdb4] Discarding device
Info: This device doesn't support BLKSECDISCARD
Info: This device doesn't support BLKDISCARD
Info: Overprovision ratio = 10.000%
Info: Overprovision segments = 44 (GC reserved = 28)
Info: format successful
```

FIGURE 18 – Formatage de la partition 4

3.3 Question 2.3 - Mesure des performances de chaque partition

Une fois les deux partitions créées et formatées, il est intéressant de comparer leur différence à l'aide d'une mesure de performance de chaque partition. Pour cela nous avons implémenté un programme qui permet d'écrire 1000 petits fichiers de 1024 bytes et un gros fichier de 1MB. Nous avons ensuite mesuré le temps d'exécution de ce programme sur les trois partitions (ext4, btrfs, f2fs).

3.3.1 Programme C d'écriture de fichier

Le listing suivant illustre le programme d'écriture de fichier qui a été implémenté :

```
1  #define _GNU_SOURCE
2  #include <sched.h>
3  #include <sys/time.h>
4  #include <sys/resource.h>
5  #include <unistd.h>
6  #include <time.h>
7  #include <stdio.h>
8  #include <string.h>
9  #include <stdlib.h>
10
11 #define SIZE 31
12 #define NB_SMALL_FILE 1000
13 #define NB_BIG_FILE 1
14 #define SIZE_SMALL 1024
15 #define SIZE_BIG 1048576
16
17 static double diff(struct timespec start, struct timespec
18                   end);
19 static int writeSmallFiles();
20 static int writeBigFile();
21
22 int main (int argc, char *argv[])
23 {
24     int ret;
25     int cpu;
26     cpu_set_t set;
27     struct sched_param sp = {.sched_priority = 50, };
28     struct timespec tStart, tEnd;
29
30     // cpu and task priority
31     if (argc == 2)
32     {
33         cpu = atoi(argv[1]);
34     }
35     else
36     {
37         cpu = 0;
38     }
39     CPU_ZERO (&set);
40     CPU_SET (cpu, &set);
41     ret=sched_setaffinity (0, sizeof(set), &set);
```

```
41     printf ("setaffinity=%d\n", ret);
42
43     ret=sched_setscheduler (0, SCHED_FIFO, &sp);
44     printf ("setscheduler=%d\n", ret);
45
46     clock_gettime(CLOCK_REALTIME, &tStart);
47     writeSmallFiles();
48     clock_gettime(CLOCK_REALTIME, &tEnd);
49     printf ("Write small files, time[s]: %f\n", diff(tStart,
50         tEnd));
51
52     clock_gettime(CLOCK_REALTIME, &tStart);
53     writeBigFile();
54     clock_gettime(CLOCK_REALTIME, &tEnd);
55     printf ("Write big files, time[s]: %f\n", diff(tStart,
56         tEnd));
57
58     return (0);
59 }
60 static int writeSmallFiles()
61 {
62     char filename[SIZE];
63     char tab[SIZE_SMALL];
64     FILE * fp;
65
66     for (int i = 0; i < NB_SMALL_FILE; i++)
67     {
68         sprintf(filename, "Output/small/smallfile_%03d.ses",
69             i);
70         fp = fopen(filename, "w");
71         if (fp == NULL)
72         {
73             printf( "Cannot open file %s\n", filename);
74             return -1;
75         }
76         for (int i = 0; i < SIZE_SMALL; i++)
77         {
78             tab[i] = 'a';
79         }
80         fprintf(fp, tab);
81         fclose(fp);
82     }
83     return 0;
84 }
85 static int writeBigFile()
86 {
87     char tab[SIZE_BIG];
88     FILE * fp;
89
90     fp = fopen("Output/big/bigfile.ses", "w");
91     if (fp == NULL)
92     {
93         printf( "Cannot open file file_big\n");
94         return -1;
95     }
96     for (int i = 0; i < SIZE_BIG; i++)
```

```

94     {
95         tab[i] = 'a';
96     }
97
98     fprintf(fp, tab);
99     fclose(fp);
100    return 0;
101 }
102
103 static double diff(struct timespec start, struct timespec
104                    end)
105 {
106     double t1, t2;
107
108     t1 = (double)start.tv_sec;
109     t1 = t1 + ((double)start.tv_nsec)/1000000000.0;
110     t2 = (double)end.tv_sec;
111     t2 = t2 + ((double)end.tv_nsec)/1000000000.0;
112     return (t2-t1);
113 }

```

3.3.2 Mesures et résultats

Sur la figure 19, nous pouvons observer le temps d'exécution du programme en fonction des différents systèmes de fichier utilisés. Le type de système de fichier le plus rapide est le F2FS avec un temps de 44.9[ms] pour l'écriture des petits fichiers comparé au 62[ms] pour les deux autres systèmes de fichier. Ceci s'explique par la façon dont est conçu le F2FS. Les systèmes de fichier utilisant des logs sont très utilisés pour la mémoire flash, car ils s'adaptent très bien et sont très performants pour ce type de mémoire. On peut voir aussi que la différence entre l'architecture B-Tree et journalisé est très faible avec des performances légèrement meilleures pour le système de fichier EXT4 de type journalisé.

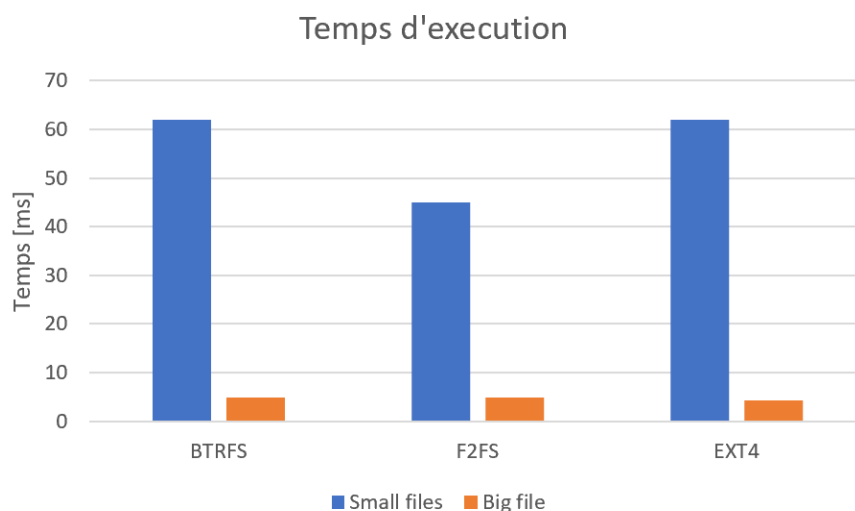


FIGURE 19 – Histogramme des temps d'exécution du programme en fonction des systèmes de fichier

3.4 Question 3 : LUKS, cryptsetup, dmccrypt

3.4.1 Question 3.1 : Création d'une partition LUKS

LUKS (Linux unified Key Setup) est le standard associé à Linux pour le cryptage (chiffrement) de disque. LUKS permet de crypter l'intégralité d'un disque en garantissant que celui-ci soit utilisable sur d'autres plateformes de Linux ou d'autre système d'exploitation. Il a l'avantage de supporter des mots de passes multiples permettant à plusieurs utilisateurs de décrypter le même volume sans pour autant partager leur mot de passe.

En résumé, LUKS possède les avantages suivants :

- Une meilleure compatibilité par la standardisation
- Une sécurité contre les attaques
- Peut supporter plusieurs clés (keys)
- Une suppression efficace des passphrases
- Est gratuit

Cryptsetup est l'implémentation de référence de LUKS sous le noyau Linux. **Cryptsetup** utilise **dm-crypt** pour le cryptage de volume. **Dm-crypt** est un sous-système de cryptage de disque, il fait partie de l'infrastructure **device-mapper**, ce qui lui permet de crypter des disques entiers, des partitions ou bien simplement des fichiers. **Dm-crypt** est le successeur de **cryptoloop**. Contrairement à son prédécesseur **dm-crypt** prend en charge des modes d'opération avancés (XTS, LRW, ESSIV) dans le but de contrer des attaques de type *watermarking* (tatouage numérique), technique consistant à cacher des informations, des données ou des messages dans des images, des pistes audios, des vidéos ou d'autres types de fichier.

L'utilisateur peut simplement spécifier l'un des chiffrements symétriques disponibles, un mode de cryptage, une clé (de n'importe quelle taille autorisée), un mode IV de génération (plain, plain64, plain64be, essiv, benbi, null, lmk, tcw, random ou eboiv), puis l'utilisateur peut créer un nouveau *node* de périphérique dans `/dev/mapper`.

Dans cette partie du laboratoire, il est demandé de développer quelques points importants concernant LUKS.

1. Différence entre "**Plain Mode**" et "**LUKS extension mode**" : **Cryptsetup** est utilisé pour configurer facilement le périphérique géré par le *device-mapper* **dm-crypt**. Ceux-ci incluent les volumes "**plain dm-crypt**" et les volumes **LUKS**.
 - **Plain dm-crypt** permet de crypter l'appareil secteur par secteur avec un hachage unique et utilise une "*passphrase*" "*non-salted*". Aucun contrôle n'est effectué et aucune métadonnée n'est utilisée. Il n'y a pas non plus d'opération de formatage. Lorsque le périphérique brut est mappé, les opérations de périphérique habituelles peuvent être utilisées, y compris la création de système de fichiers. Les périphériques mappés se trouvent généralement dans `/dev/mapper/<nom>`.

- **LUKS extension mode** utilise un en-tête de métadonnées et peut donc offrir plus de fonctionnalités que **plain dm-crypt**. Le désavantage est que l'en-tête est visible et vulnérable aux dommages.
- 2. Signification de l'option **-hash**
 - Permet de spécifier le hachage utilisé dans le schéma de configuration de clé **LUKS** et le "*volume key digest*" pour **luksFormat**. Le hachage spécifié est utilisé comme paramètre de hachage pour **PBKDF2** et pour le AF splitter (Anti-Forensic information splitter).
- 3. Définition de "**default cipher**" du mode LUKS
 - Le chiffrement par défaut ("**default cipher**") du mode LUKS est du type **aes-xts-plain64**
- 4. Définition de l'option **-key-file**
 - Permet de lire une **passphrase** depuis un fichier spécifié par son chemin d'accès

3.4.2 Question 3.2 : Partition LUKS

Une fois les fonctionnalités de *cryptsetup* ainsi que les définitions des différentes options du mode *LUKS* documentées, il est maintenant nécessaire de réaliser la configuration de la partition LUKS. Pour cela, nous avons réalisé les étapes suivantes :

1. Copier un fichier dans la partition LUKS
 - Initialiser la partition LUKS avec l'option **-pbkdf pbkdf2**
 - Formater la partition LUKS en **EXT4**
 - Monter la partition LUKS dans le dossier [/mnt/usr](#)
2. Ajouter une nouvelle **passphrase** à la partition LUKS
3. Vider le header de la partition et la crypter avec la **master key**
4. Avec la commande **dd**, copier 1 Mbytes de la partition [/dev/sdb3](#) dans un fichier
 - Trouver le header de la partition
 - Trouver la **master key** cryptée
5. Connecter la carte microSD sur le NanoPi et activer la partition cryptée

La première étape consiste à modifier le fichier *generate.sh* afin d'ajouter la création d'une troisième partition **LUKS** de type **EXT4**.

```
1 sudo parted $SD_ROOT_FOLDER mkpart primary ext4 5242880s 9338879s
2 sudo mkfs.ext4 $SD_ROOT_PART3 -L LUKS
```

Ensuite, il faut ajouter les commandes d'initialisation de la partition cryptée disponible à l'aide de [cryptsetup](#). Voici le listing des modifications à ajouter au fichier *generate.sh* :

```
1 DEVICE=/dev/sdb3
2 sudo cryptsetup --debug --pbkdf pbkdf2 luksFormat $DEVICE -q
3 sudo cryptsetup luksDump $DEVICE
4 sudo cryptsetup --debug open --type luks $DEVICE usrfs1 ---key-file
5 sudo mkfs.ext4 /dev/mapper/usrfs1
6 sudo mount /dev/mapper/usrfs1 /mnt/usrfs
```


Afin de copier un fichier dans la partition LUKS il est nécessaire d'utiliser la commande suivante :

```
1 sudo cp $LOCAL_PATH/copie_test.txt /mnt/usrfs
```

Il est possible de contrôler que la copie a bien été réalisé et que **usrfs** comporte bien le fichier *copie_test.txt*

```
[lmi@fedora VM_SeS]$ nano copie_test.txt
[lmi@fedora VM_SeS]$ sudo cp copie_test.txt /mnt/usrfs/
[lmi@fedora VM_SeS]$ cd /mnt/usrfs/
[lmi@fedora usrfs]$ ls
copie_test.txt  lost+found  t.txt
[lmi@fedora usrfs]$
```

FIGURE 20 – Copie d'un fichier dans la partition LUKS

Il est aussi possible d'ajouter une passphrase supplémentaire avec les commandes :

```
1 DEVICE=/dev/sdb3
2 sudo cryptsetup luksAddKey $DEVICE
```

Une fois la passphrase supplémentaire ajouté, il est possible de les afficher dans la console afin de contrôler leur caractéristiques. Cela est illustré dans la figure 21 :

```
0: crypt
  offset: 16777216 [bytes]
  length: (whole device)
  cipher: aes-xts-plain64
  sector: 512 [bytes]

Keyslots:
0: luks2
  Key: 512 bits
  Priority: normal
  Cipher: aes-xts-plain64
  Cipher key: 512 bits
  PBKDF: pbkdf2
  Hash: sha256
  Iterations: 2668132
  Salt: 61 50 23 f6 95 77 01 41 da c7 e5 18 f7 8b 7d a6
      98 95 91 a4 2a 1a e1 4d b9 6c a3 4d 7e d7 90 2d
  AF stripes: 4000
  AF hash: sha256
  Area offset: 32768 [bytes]
  Area length: 258048 [bytes]
  Digest ID: 0

1: luks2
  Key: 512 bits
  Priority: normal
  Cipher: aes-xts-plain64
  Cipher key: 512 bits
  PBKDF: argon2i
  Time cost: 4
  Memory: 957774
  Threads: 2
  Salt: 21 f7 9f 21 d6 0e cc c8 9c 39 d7 61 61 4e fb ee
      01 7c e4 1d 15 22 09 b1 56 08 26 d5 b0 fc 41 65
  AF stripes: 4000
  AF hash: sha256
  Area offset: 290816 [bytes]
  Area length: 258048 [bytes]
  Digest ID: 0

Tokens:
Digests:
0: pbkdf2
  Hash: sha256
  Iterations: 172918
  Salt: ea 4c 76 27 20 b5 bc 59 5c bc 19 cf 8d 94 91 56
      dc 45 ad b4 9b 53 34 a6 72 c7 76 b1 72 95 99 cd
  Digest: 13 57 59 9f 21 5e 6f 3b 6e 94 ce 9e dd f0 ed 00
      5e 47 c6 b2 ec 9c e1 b8 7a ee 1b 87 7a 5f e2 d2
[lmi@fedora usrfs]$
```

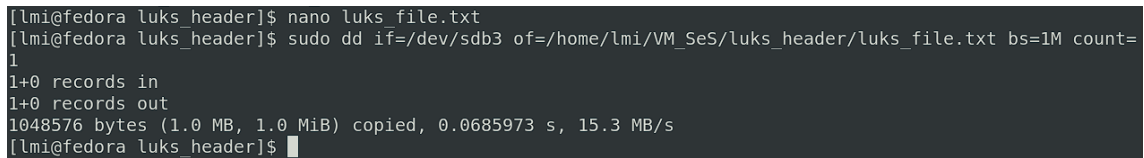
FIGURE 21 – Ajout d'une seconde passphrase

Le nettoyage de l'entête (header dump) se réalise avec les commandes :

```
1 DEVICE=/dev/sdb3
2 sudo cryptsetup luksDump $DEVICE
```

Ensuite, afin de trouver le header de la partition ainsi que la **master key** crypté, il est demandé de copier 1MB de la partition `/dev/sdb3`. Ceci se réalise avec la commande :

```
1 sudo dd if=/dev/sdb3 of=${PATH_LOCAL}/file.txt bs=1M count=1
```



```
[lmi@fedora luks_header]$ nano luks_file.txt
[lmi@fedora luks_header]$ sudo dd if=/dev/sdb3 of=/home/lmi/VM_SeS/luks_header/luks_file.txt bs=1M count=1
1
1+0 records in
1+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.0685973 s, 15.3 MB/s
[lmi@fedora luks_header]$
```

FIGURE 22 – Copie de la partition LUKS avec la commande `dd`

Il est possible de lire le header de la partition LUKS en contrôlant le contenu du fichier binaire que nous avons copié précédemment en le lisant comme un fichier texte. Nous trouvons les informations concernant les deux **master key** au début du fichier en réalisant une recherche avec le mot clés `"key"`.

Voici le header de la première **master key** :

```
{
  "keyslots": {
    "0": {
      "type": "luks2",
      "key_size": 64,
      "af": {
        "type": "luks1",
        "stripes": 4000,
        "hash": "sha256",
        "area": {
          "type": "raw",
          "offset": 32768,
          "size": 258048,
          "encryption": "aes-xts-plain64",
          "key_size": 64
        },
        "kdf": {
          "type": "pbkdf2",
          "hash": "sha256",
          "iterations": 2668132,
          "salt": "YVAj9pV3AUHax+UY94t9ppiVkaQqGuFNUWyjTX7XkC0="
        }
      },
    },
    "1": {
      "type": "luks2",
      "key_size": 64,
      "af": {
        "type": "luks1",
        "stripes": 4000,
        "hash": "sha256",
        "area": {
          "type": "raw",
          "offset": 290816,
          "size": 258048,
          "encryption": "aes-xts-plain64",
          "key_size": 64
        },
        "kdf": {
          "type": "argon2i",
          "time": 4,
          "memory": 957774,
          "cpus": 2,
          "salt": "IfefIdY0zMic0ddhYU777gF85B0VlgmxVggm1bD8QWU="
        }
      },
    },
    "tokens": {},
    "segments": {
      "0": {
        "type": "crypt",
        "offset": 16777216,
        "size": "dynamic",
        "iv_tweak": "0",
        "encryption": "aes-xts-plain64",
        "sector_size": 512
      },
    },
    "digests": {
      "0": {
        "type": "pbkdf2",
        "keyslots": ["0", "1"],
        "segments": ["0"],
        "hash": "sha256",
        "iterations": 172918,
        "salt": "6kx2JyC1vFlcvBnPjZSRVtxFrbSbUzSmcsd2sXKVMc0=",
        "digest": "E1dZnyFebztulM6e3fDtAF5HxrLsnOG4eu4bh3pf4tI="
      },
    },
    "config": {
      "json_size": 12288,
      "keyslots_size": 16744448
    }
  }
}
```

Voici le header de la deuxième **master key** :

```
{
  "keyslots": {
    "0": {
      "type": "luks2",
      "key_size": 64,
      "af": {
        "type": "luks1",
        "stripes": 4000,
        "hash": "sha256"
      },
      "area": {
        "type": "raw",
        "offset": "32768",
        "size": "258048",
        "encryption": "aes-xts-plain64",
        "key_size": 64
      },
      "kdf": {
        "type": "pbkdf2",
        "hash": "sha256",
        "iterations": 2668132,
        "salt": "YVAj9pV3AUHax+UY94t9ppiVkaQqGuFNUWyjTX7XkC0="
      }
    },
    "1": {
      "type": "luks2",
      "key_size": 64,
      "af": {
        "type": "luks1",
        "stripes": 4000,
        "hash": "sha256"
      },
      "area": {
        "type": "raw",
        "offset": "290816",
        "size": "258048",
        "encryption": "aes-xts-plain64",
        "key_size": 64
      },
      "kdf": {
        "type": "argon2i",
        "time": 4,
        "memory": 957774,
        "cpus": 2,
        "salt": "IfefIdY0zMic0ddhYU777gF85B0VIgmxVggm1bD8QWU="
      }
    }
  },
  "tokens": {},
  "segments": {
    "0": {
      "type": "crypt",
      "offset": "16777216",
      "size": "dynamic",
      "iv_tweak": "0",
      "encryption": "aes-xts-plain64",
      "sector_size": 512
    }
  },
  "digests": {
    "0": {
      "type": "pbkdf2",
      "keyslots": ["0", "1"],
      "segments": ["0"],
      "hash": "sha256",
      "iterations": 172918,
      "salt": "6kx2JyC1vFlcvBnPjZSRVtxFrbSbUzSmcsd2sXKVmc0=",
      "digest": "E1dZnyFebztulM6e3fDtAF5HxrLsnOG4eu4bh3pf4tI="
    }
  },
  "config": {
    "json_size": "12288",
    "keyslots_size": "16744448"
  }
}
```

Ce qui est surtout important de noter, ce sont les deux valeurs des positions des **master keys** cryptées. Ces positions sont données par les valeurs *"offset" : "32768"* et *"offset" : "290816"*. Il est possible de trouver les deux **master keys** à ces deux positions d'offset directement dans le fichier.

Afin de lire ce fichier nous avons utilisé l'outil [ghex](#), la figure 23 et la figure 24 illustre les deux **master keys** :

```
00007FC8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00007FE5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 40 ED
00008002 3B 87 FB AC EF 6A AC A4 3B 64 F7 CE 5B FD F4 7F 93 67 47 36 5C 7E 27 95 91 1F 49 19 07
0000801F E4 F6 E1 9D 3E 99 BD C1 E1 33 7F BF 89 94 AC E3 AD DF 8F BB F7 B4 04 CE A3 A0 79 6F FB
0000803C E0 AB EA 5E 5F C6 A4 D8 49 F2 E4 CC 31 77 1B 0E F2 3D 51 B1 92 FE 7E 15 A6 56 A7 01 F9
00008059 20 EC 05 E3 E6 A0 0F 1A 0C 92 B5 C2 F9 3A 58 3D 5C 36 05 49 67 73 77 BA 0A 14 89 D5 E1
00008076 01 0E 27 52 CD E3 58 E1 EB 8E AB DC AD 59 A0 BA 98 79 D2 F5 9D 0F 5F 26 16 80 C3 C4 4D
00008093 98 0F 8F C5 55 93 EC 1D 1A D7 D1 80 94 35 4A A5 F0 BF BB EA 0A 9D 75 FB 7F F1 5A F9 AF
000080B0 54 6F 08 A3 B3 E1 9C FC BC 00 F6 8A 03 F4 6D 06 6A 87 14 8C E3 0F C7 59 64 36 99 E2 F1
000080CD C8 14 E1 69 E5 89 A9 B5 98 C2 12 17 AB 74 FA EE 7E 66 80 E2 3D 29 0B 02 28 85 B3 05 01
000080EA 5E 84 1D 60 92 70 00 AE AC AC 29 0B CB 35 66 AC 75 F5 D7 24 F7 C4 CB D2 E8 69 1C AA A9
00008107 C4 E9 0B E6 C5 EF A6 6D 16 38 7A 37 20 E7 0B 23 5D 07 9D 2B 96 F9 22 55 87 8B 5E 1B F6
00008124 0E 93 7E 55 75 93 5E 04 78 64 EC D2 B6 F1 23 D2 80 62 06 31 57 79 6E B4 E5 94 8C 3E D0
00008141 49 B1 BA D9 D3 0A E1 92 C9 0F 57 8A B4 BE FD 36 28 F0 57 49 74 27 4A 6C 97 43 79 B5 28
0000815E B0 94 33 C7 F0 17 77 9F 1F AC C9 60 82 B6 5A B0 88 09 31 E9 C1 85 A1 5A 10 12 CF A6 ED
0000817B B9 07 C9 19 20 BE 9D 29 B2 62 47 E2 86 DC 2F E9 2A 9E 78 47 E6 90 40 5D 84 CF 38 0A 20
00008198 26 9B 41 79 23 5E EE 82 1F BA 77 A4 3C DC 1F 15 14 2D 6D DE A3 1A C1 2A A8 21 AC BC 6B
000081B5 72 8C C3 B1 64 6B 46 41 66 81 C8 95 44 AE 8F 85 88 87 CE 25 FD 32 60 28 24 93 DE BA 14
000081D2 B8 43 60 65 1F 34 A0 2D D3 37 B5 08 87 C5 F5 2C 11 A4 AB F7 EF B3 66 BC 33 11 F4 A3 C6
000081EF C5 3E 7A 6D E3 01 76 CF 67 8B AB 90 D9 0C 8D B3 6D 17 DC 4E 05 26 08 AB 9C 1A F3 3C E6
0000820C E2 95 AA 3D 15 98 7D C5 74 98 7B 3A 95 EF 9F 4D EA C2 B2 B3 EE 24 03 9B 39 39 BB 17 EA
```

FIGURE 23 – Master Key 1

```

00046FDF D7 7C 09 37 EE E0 35 D5 A0 4E 89 54 5E 83 C0 27 FC B7 AE 3F C3 C3 BC 52 8F 7A E5 9D 0E
00046FFC C5 4A 4E 3F D8 E2 87 47 69 44 5C 55 C9 14 02 E3 87 C0 5D EB 14 60 E2 65 BF 4F D7 92 24
00047019 66 F6 B3 66 1B 49 63 82 8C 0F AD F9 57 D9 1E B1 B2 DB 81 75 32 8A 9D 09 01 DD EF 2F A6
00047036 2E 61 EB EE 30 57 8D F1 30 CC 47 EF A0 1D 9D FB E7 D5 67 1D FA 4C 33 EE 09 6E 11 B1 82
00047053 B0 D1 2C FC 57 D4 D4 35 C8 64 67 DC 63 6F 29 28 D7 D3 3D 6C 83 A5 6C A9 C8 0F A5 8C F3
00047070 39 EE B1 3D 68 AC 9D A8 1A CD 5A DA DA F2 D5 D5 78 27 CA E3 8E F3 5D 49 13 8F 22 44 3A
0004708D 23 CD DA 9A 5D 34 32 DF 2A 38 B6 E6 70 1A 06 93 3D B8 93 C3 CA CE ED D9 7F CF AC 71 CF
000470AA 2A 43 7C A5 92 49 60 69 45 C6 6A 7D 67 C7 3C 3C C7 45 53 95 7E B5 24 30 3C 39 CD 71 A1
000470C7 D4 EE 3E B1 CF 03 70 E5 8D 49 DD D1 81 5B E0 1A 82 74 FF FA B8 C2 23 9A 7F CC 3D 7C DE
000470E4 F2 CB E5 11 AB EE FB AE 8B DA 23 1A 0C FB 37 C7 47 FE 35 C2 D5 15 BB 42 D0 2F 5A F1 7D
00047101 52 0C AC 04 C1 E0 28 D9 78 78 85 3D 46 EE F9 2B 4A 20 43 4F E8 C9 23 B7 1F FD 81 F2 EE
0004711E 90 75 21 A3 7C FE 29 91 7D D5 42 D2 B2 6C 9D 18 80 65 67 7B 04 1D 7A 71 3C B8 D8 90 A9
0004713B 99 EF 4D 2E 40 71 93 8F 81 95 B4 29 BE 99 7F 65 2F 5B 01 37 0B 47 66 F1 66 93 15 D5 0D
00047158 64 77 4E EF C4 AE 7F CE F8 DC B0 6A F7 27 25 0A 06 87 B8 3D 53 6A E6 00 EE 58 0F DA 8C
00047175 92 AA 7C 3E 64 82 8D 04 4D 0C C5 7A CD 8A 8D 37 6D 1C D9 E3 DB 14 AB 9D 38 95 D6 E8 95
00047192 B8 56 99 E7 A2 7A 5E C7 86 A0 0C D6 A5 B4 4C DE 6D FC EB EC F1 63 2D 9D 1A DD 32 81 8E
000471AF A3 26 03 40 9A 3D 66 1A 32 4D 9B 74 DD 27 80 B7 4F 13 A6 4B 0D 26 9D 1E 76 72 C8 6E C1
000471CC 16 33 50 64 37 92 6D D2 AE 91 5E 82 69 AE C6 7E 01 08 2B 80 91 3C 38 49 98 0E 6A 22 70
000471E9 B1 31 B3 BE 80 76 37 69 88 BE FC 8C F3 B1 0E 47 DF 92 62 97 24 8D BE 4C 06 5F D4 48 C2
00047206 4F 18 17 62 C1 27 5D 6A BC 21 CD 38 F4 A4 CB 0D 43 15 B7 DA CE 1B 90 D0 F2 A4 73 33 6B

```

FIGURE 24 – Master Key 2

Finalement, une fois la partition cryptée implémentée sur la carte microSD, il est encore nécessaire de réaliser le démarrage du NanoPi. Pour réaliser cela, il est nécessaire de suivre les instructions suivantes :

1. Dans minicom faire le boot du NanoPi
2. Lancer la commande `cryptsetup --debug open --type luks \ $DEVICE usrfs1`
3. Introduire la passphrase
4. Lancer la commande `mkdir /mnt/usrfs` afin de créer le dossier qui sera monté
5. Monter la partition avec la commande `mount /dev/mapper/usrfs1 /mnt/usrfs`

La figure 25 illustre le démarrage de la partition cryptée correspondant à la démarche expliqué précédemment. Nous avons aussi ouvert le fichier `copie_text.txt` afin d'illustrer que la partition démarrée correspondait bien à ce qui a été réalisée dans les questions précédentes.

```

Welcome to FriendlyARM Nanopi NEO Plus2
buildroot login: root
# ls
# sudo cryptsetup open --type luks /dev/mmcblk0p3 usrfs1
--sh: sudo: not found
# cryptsetup open --type luks /dev/mmcblk0p3 usrfs1
WARNING: Locking directory /run/cryptsetup is missing!
Enter passphrase for /dev/mmcblk0p3:
# cd /mnt
# ls
# pwd
/mnt
# mkdir /mnt/usrfs
# mount /dev/mapper/usrfs1 /mnt/usrfs
[ 233.882145] EXT4-fs (dm-0): recovery complete
[ 233.889757] EXT4-fs (dm-0): mounted filesystem with ordered data mode. Opts: (null)
# cd /mnt/usrfs/
# ls
copie_test.txt  lost+found      t.txt
# cat copie_test.txt
Classified informations !
#

```

FIGURE 25 – Démarrage du NanoPi avec la partition cryptée

3.4.3 Question 3.3 : Rootfs dans la partition LUKS

Une fois que la configuration de partition LUKS est comprise, nous souhaitons maintenant y ajouter **Rootfs** dans le but de pouvoir crypter et utiliser un noyau Linux fonctionnel et complet.

Cette étape est décomposée en deux parties, premièrement sur la machine virtuelle (PC), les étapes suivantes sont réalisées.

1. Générer et enregistrer une passphrase aléatoire dans un fichier
2. Initialiser une partition LUKS avec les options suivantes
 - pbkdf pbkdf2
 - Key-size : 512
 - Passphrase : dans le fichier "passphrase"
3. Créer un mapping de `dev/mapper/usrfs1`
4. Formater la partition LUKS en **EXT4**
5. Copier le contenu de **rootfs** de la partition LUKS

Afin de générer une passphrase aléatoire il est nécessaire d'utiliser la commande suivante :

```
1 sudo dd if=/dev/urandom of=${PATH_LOCAL}/rand_key.txt bs=64 count=1
```

Ensuite il faut utiliser l'option `key-file` pour utiliser un fichier comme source de passphrase.

Comme illustré par la commande complète :

```
1 sudo cryptsetup luksAddKey $DEVICE --key-file ${PATH_LOCAL}/  
   rand_key.txt
```

Puis nous réalisons une copie allégée de **rootfs** de 4MB dans la partition LUKS à l'aide de la commande :

```
1 sudo dd if=~/.workspace/nano/buildroot/output/images/rootfs.ext4 of  
   =/dev/mapper/usrfs1 bs=4M
```

Une fois la partition LUKS contenant rootfs implémentée, nous pouvons nous intéresser à son utilisation sur le NanoPi. Pour cela nous réalisons les étapes suivantes :

1. Démarrer le NanoPi et monter manuellement la partition LUKS
2. Écrire un script d'initialisation afin de monter automatiquement la partition LUKS au démarrage

Les commandes nécessaires au démarrage du NanoPi et du montage manuel de la partition LUKS sont illustrées dans le script **S30luks.sh** du listing suivant :

```
1 #!/bin/sh  
2 DEVICE=/dev/mmcblk0p3  
3  
4 cryptsetup --debug open --type luks $DEVICE usrfs1 --key-file=/root  
   /passphrase  
5  
6 mkdir /mnt/usrfs  
7  
8 mount /dev/mapper/usrfs1 /mnt/usrfs
```

La différence entre le montage manuel et automatique est que nous avons entrée manuellement la passphrase lors de la première étape.

Le script de montage automatique **S30luks** est stocké dans le dossier `init.d` du répertoire `etc` du NanoPI.

Le listing suivant illustre le script que nous avons réalisé permettant le formatage automatique de la carte microSD avec l'ajout de la partition LUKS supplémentaire.

```

1 #!/bin/sh
2 # These 3 variables must be modified according to user setup
3 # They are not subject to change(check SD root folder just in case)
4 # Default values according to course slides (1_NanoPi.pdf)
5 SD_ROOT_FOLDER=/dev/sdb          # SD card root directory, unmounted
6 LOCAL_MNT_POINT=/run/media/lmi  #default SD card mount point
7 HOME_DIR=/home/lmi              # user home directory
8
9 SD_ROOT_PART1=${SD_ROOT_FOLDER}1
10 SD_ROOT_PART2=${SD_ROOT_FOLDER}2
11 SD_ROOT_PART3=${SD_ROOT_FOLDER}3
12 #SD_ROOT_PART4=${SD_ROOT_FOLDER}4
13 echo "----- #umount -----"
14 umount $SD_ROOT_PART1
15 umount $SD_ROOT_PART2
16 umount $SD_ROOT_PART3
17 #umount $SD_ROOT_PART4
18 echo "----- #initialize 480MiB to 0-----"
19 #initialize 480MiB to 0
20 sudo dd if=/dev/zero of=$SD_ROOT_FOLDER count=120000
21 sync
22 echo "----- # First sector: msdos-----"
23 # First sector: msdos
24 sudo parted $SD_ROOT_FOLDER mklabel msdos
25 echo "----- #copy sunxi-spl.bin binaries-----"
26 #copy sunxi-spl.bin binaries
27 sudo dd if=${HOME_DIR}/workspace/nano/buildroot/output/images/sunxi-
   -spl.bin of=$SD_ROOT_FOLDER bs=512 seek=16
28 echo "----- #copy u-boot-----"
29 #copy u-boot
30 sudo dd if=${HOME_DIR}/workspace/nano/buildroot/output/images/u-
   -boot.itb of=$SD_ROOT_FOLDER bs=512 seek=80
31 echo "----- # 1st partition: 64MiB-----"
32 # 1st partition: 64MiB: (163840-32768)*512/1024 = 64MiB
33 sudo parted $SD_ROOT_FOLDER mkpart primary ext4 32768s 163839s
34 echo "----- # 2nd partition: 1GiB-----"
35 # 2nd partition: 1GiB: (4358144-163840)*512/1024 = 2GiB
36 sudo parted $SD_ROOT_FOLDER mkpart primary ext4 163840s 4358143s
37 sudo mkfs.ext4 $SD_ROOT_PART2 -L rootfs
38 sudo mkfs.ext4 $SD_ROOT_PART1 -L BOOT # Change line "-L" is for
   Volume Label
39 sync
40 echo "----- # 3rd partition: 2GiB-----"
41 # 3rd partition: 2GiB: (9338879-5242880)*512/1024 = 2GiB
42 sudo parted $SD_ROOT_FOLDER mkpart primary ext4 5242880s 9338879s
43 sudo mkfs.ext4 $SD_ROOT_PART3 -L LUKS
44 echo "----- #copy kernel, flattened device tree, boot.scr-----"

```



```

45 #copy kernel, flattened device tree, boot.scr
46 sudo mount $SD_ROOT_PART1 $LOCAL_MNT_POINT
47 sudo cp ${HOME_DIR}/workspace/nano/buildroot/output/images/Image
   $LOCAL_MNT_POINT
48 sudo cp ${HOME_DIR}/workspace/nano/buildroot/output/images/NanoPi-
   neo-plus2.dtb $LOCAL_MNT_POINT
49 sudo cp ${HOME_DIR}/workspace/nano/buildroot/output/images/boot.scr
   $LOCAL_MNT_POINT
50 sync
51 echo "----- #Rename 1st partition to BOOT -----"
52 #Rename 1st partition to BOOT
53 sudo umount $SD_ROOT_PART1
54 sudo e2label $SD_ROOT_PART1 BOOT
55 echo "----- #copy rootfs -----"
56 #copy rootfs
57 sudo dd if=${HOME_DIR}/workspace/nano/buildroot/output/images/
   rootfs.ext4 of=$SD_ROOT_PART2
58 echo "----- #Resize and rename 2nd partition to rootf -----"
59 # Resize and rename 2nd partition to rootfs
60 # check if the partition must be mounted
61 sudo e2fsck -f $SD_ROOT_PART2
62 sudo resize2fs $SD_ROOT_PART2
63 sudo e2label $SD_ROOT_PART2 rootfs

```

3.5 Question 4 : Initramfs

Maintenant que nous avons pu monter avec succès une partition LUKS contenant **Rootfs**, il reste encore un problème à régler. Dans la partie précédente, nous utilisons la partition EXT4 contenant Rootfs afin de décrypter et de monter la troisième partition (LUKS). Comment lancer les commande de *cryptsetup* permettant de décrypter la partition 3 alors que *cryptsetup* fait partie de la partition LUKS qui sera cryptée, et cela, sans utiliser une autre partition. Pour palier ce problème, nous devons utiliser un **initramfs**.

La seule fonction d'un initramfs est de monter le système de fichier racine d'un noyau Linux. L'initramfs est un ensemble complet de répertoires que l'on peut trouver dans un système de fichiers racine normal.

Dans notre cas, nous souhaitons réaliser un initramfs permettant d'utiliser les commandes de *cryptsetup*. Il faut donc trouver toutes les bibliothèques nécessaires à *cryptsetup* ainsi que celle permettant de réaliser les commande de base de Linux. Il est cependant nécessaire de décomposer cette étape en commençant par implémenter un initramfs simple contenant uniquement les commande de base de Linux.

Nous souhaitons donc réaliser les étapes suivantes :

1. Sur la VM, générer un **initramfs** qui exécute la commande `exec sh` afin d'afficher la console de **initramfs**
2. Initialiser le NanoPi afin de démarrer **initramfs**
3. Démarrer le NanoPi et exécuter manuellement la commande `exec switch_root`

Afin de générer l'image de *Initramfs* et afin de flasher la carte microSD il faut exécuter les deux scripts suivants :

```
1 ./initRAMfs.sh
2 ./generate_sd_initramf.sh
```

Ces commandes permettent de lancer deux scriptes que nous avons réalisées. Le premier script permet de réaliser l'image compressée d'un initramfs simple sans l'ajout des bibliothèques nécessaire à *cryptsetup*. Le second script permet de flasher la carte microSD avec le initramf simple.

Le listing suivant illustre le script `init_ramfs.sh` qui permet de créer un initramfs simple :

```
1 #!/bin/sh
2 ROOTFSLOC=ramfs
3 HOME=/home/lmi
4 echo "----- Begin -----"
5 cd $HOME
6 mkdir $ROOTFSLOC
7 mkdir -p $ROOTFSLOC/{bin,dev,etc,home,lib,lib64,newroot,proc,root,
   sbin,sys}
8 echo "----- Cpy /dev -----"
9 cd $ROOTFSLOC/dev
10 sudo mknod null c 1 3
11 sudo mknod tty c 5 0
12 sudo mknod console c 5 1
13 sudo mknod random c 1 8
14 sudo mknod urandom c 1 9
15 sudo mknod mmcblk0p b 179 0
16 sudo mknod mmcblk0p1 b 179 1
17 sudo mknod mmcblk0p2 b 179 2
18 sudo mknod mmcblk0p3 b 179 3
19 sudo mknod mmcblk0p4 b 179 4
20 sudo mknod ttyS0 c 4 64
21 sudo mknod ttyS1 c 4 65
22 sudo mknod ttyS2 c 4 66
23 sudo mknod ttyS3 c 4 67
24 echo "----- Cpy /bin -----"
25 cd ../bin
26 cp ~/workspace/nano/buildroot/output/target/bin/busybox .
27 ln -s busybox ls
28 ln -s busybox mkdir
29 ln -s busybox ln
30 ln -s busybox mknod
31 ln -s busybox mount
32 ln -s busybox umount
33 ln -s busybox sh
34 ln -s busybox sleep
35 ln -s busybox dmesg
36 cp ~/workspace/nano/buildroot/output/target/usr/bin/strace .
37 echo "----- Cpy /sbin -----"
38 cd ../sbin
39 ln -s ../bin/busybox switch_root
40 echo "----- Cpy /lib64 -----"
41 cd ../lib64
```

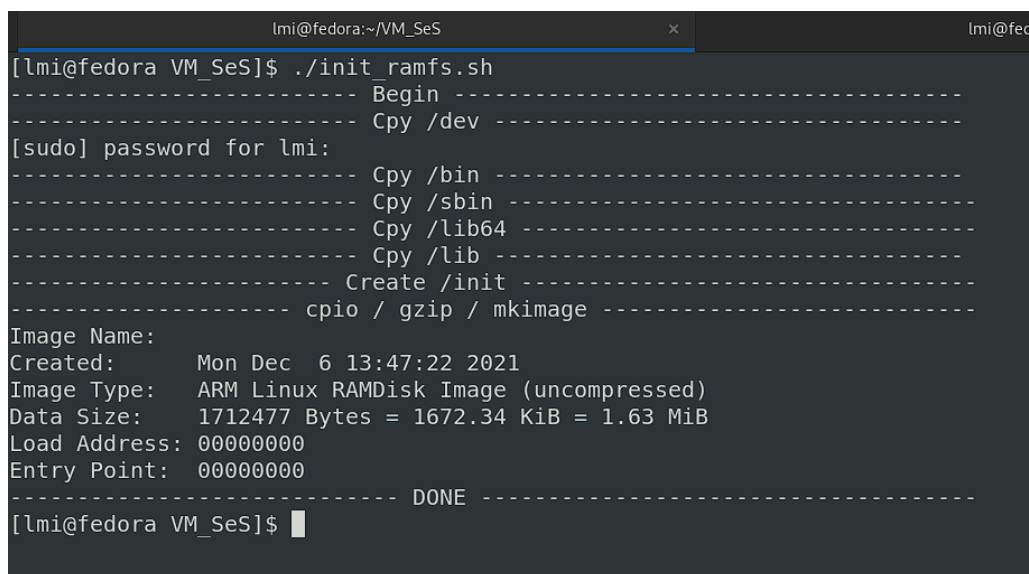


```

42 cp ~/workspace/nano/buildroot/output/target/lib64/ld-2.31.so .
43 cp ~/workspace/nano/buildroot/output/target/lib64/libresolv-2.31.so
44 cp ~/workspace/nano/buildroot/output/target/lib64/libc-2.31.so .
45 ln -s libresolv-2.31.so libresolv.so.2
46 ln -s libc-2.31.so libc.so.6
47 ln -s ../lib64/ld-2.31.so ld-linux-aarch64.so.1
48 echo "----- Cpy /lib -----"
49 cd ../lib
50 cp ~/workspace/nano/buildroot/output/target/lib64/ld-2.31.so .
51 ln -s ../lib64/ld-2.31.so ld-linux-aarch64.so.1
52 echo "----- Create /init -----"
53 cd ..
54 cat > init << endofinput
55 #!/bin/busybox sh
56 mount -t proc none /proc
57 mount -t sysfs none /sys
58 mount -t ext4 /dev/mmcblk0p2 /newroot
59 mount -n -t devtmpfs devtmpfs /newroot/dev
60 exec sh
61 #exec switch_root /newroot /sbin/init
62 endofinput
63 #####
64 chmod 755 init
65 cd ..
66 sudo chown -R 0:0 $ROOTFSLOC
67 echo "----- cpio / gzip / mkimage -----"
68 cd $ROOTFSLOC
69 find . | cpio --quiet -o -H newc > ../Initrd
70 cd ..
71 gzip -9 -c Initrd > Initrd.gz
72 mkimage -A arm -T ramdisk -C none -d Initrd.gz uInitrd
73 echo "----- DONE -----"

```

La figure 26 illustre le log de l'exécution du script `init_ramfs.sh` :



```

lmi@fedora:~/VM_SeS
[ lmi@fedora VM_SeS ]$ ./init_ramfs.sh
----- Begin -----
----- Cpy /dev -----
[sudo] password for lmi:
----- Cpy /bin -----
----- Cpy /sbin -----
----- Cpy /lib64 -----
----- Cpy /lib -----
----- Create /init -----
----- cpio / gzip / mkimage -----
Image Name:
Created:      Mon Dec  6 13:47:22 2021
Image Type:   ARM Linux RAMDisk Image (uncompressed)
Data Size:    1712477 Bytes = 1672.34 KiB = 1.63 MiB
Load Address: 00000000
Entry Point:  00000000
----- DONE -----
[ lmi@fedora VM_SeS ]$

```

FIGURE 26 – Log de l'exécution du script `init_ramfs.sh`

Il est possible visualiser la contenu du répertoire `ramfs` qui est utilisé pour créer le `initramfs` afin de contrôler les bibliothèques qui y sont présentent. La figure 27 illustre le contenu du dossier `ramfs`.

```
[lmi@fedora ~]$ cd ramfs/
[lmi@fedora ramfs]$ ls
bin dev etc home init lib lib64 newroot proc root sbin sys
[lmi@fedora ramfs]$ tree
.
├── bin
│   ├── busybox
│   ├── cryptsetup -> busybox
│   ├── dmesg -> busybox
│   ├── ln -> busybox
│   ├── ls -> busybox
│   ├── mkdir -> busybox
│   ├── mknod -> busybox
│   ├── mount -> busybox
│   ├── sh -> busybox
│   ├── sleep -> busybox
│   ├── strace
│   └── umount -> busybox
├── dev
│   ├── console
│   ├── mmcblk0p
│   ├── mmcblk0p1
│   ├── mmcblk0p2
│   ├── mmcblk0p3
│   ├── mmcblk0p4
│   ├── null
│   ├── random
│   ├── ttyS0
│   ├── ttyS1
│   ├── ttyS2
│   ├── ttyS3
│   └── urandom
├── etc
├── home
├── init
├── lib
│   ├── ld-2.31.so
│   └── ld-linux-aarch64.so.1 -> ../lib64/ld-2.31.so
├── lib64
│   ├── ld-2.31.so
│   ├── ld-linux-aarch64.so.1 -> ../lib64/ld-2.31.so
│   ├── libc-2.31.so
│   ├── libc.so.6 -> libc-2.31.so
│   ├── libresolv-2.31.so
│   └── libresolv.so.2 -> libresolv-2.31.so
├── newroot
├── proc
├── root
├── sbin
│   └── switch_root -> ../bin/busybox
└── sys

11 directories, 35 files
[lmi@fedora ramfs]$
```

FIGURE 27 – Contenu du dossier `ramfs`

La figure 28 illustre les fichiers réalisés par le script `init_ramfs.sh` soit `Initrd`, `Initrd.gz` et `uInitrd` :

```
[lmi@fedora buildroot]$ cd
[lmi@fedora ~]$ ls
Desktop Downloads Initrd.gz Music Public Templates Videos workspace
Documents Initrd minicom.log Pictures ramfs uInitrd VM SeS
```

FIGURE 28 – Illustration des fichiers créés par le script `init_ramfs.sh`

Une fois le initramfs créé, il est encore nécessaire de le flasher sur la carte microSD. Pour réaliser cela, il faut exécuter le script `generate_sd_initramf.sh`.

le listing suivant illustre le contenu du script `generate_sd_initramf.sh` qui permet de flasher la carte microSD avec le initramfs :

```

1 #!/bin/sh
2 # These 3 variables must be modified according to user setup
3 # They are not subject to change(check SD root folder just in case)
4 # Default values according to course slides (1_nanopi.pdf)
5 SD_ROOT_FOLDER=/dev/sdb          # SD card root directory, unmounted
6 LOCAL_MNT_POINT=/run/media/lmi   #default SD card mount point
7 HOME_DIR=/home/lmi              # user home directory
8
9 SD_ROOT_PART1=${SD_ROOT_FOLDER}1
10 SD_ROOT_PART2=${SD_ROOT_FOLDER}2
11 SD_ROOT_PART3=${SD_ROOT_FOLDER}3
12 #SD_ROOT_PART4=${SD_ROOT_FOLDER}4
13 echo "----- umount -----"
14 umount $SD_ROOT_PART1
15 umount $SD_ROOT_PART2
16 umount $SD_ROOT_PART3
17 #umount $SD_ROOT_PART4
18 echo "----- initialize 480MiB to 0-----"
19 #initialize 480MiB to 0
20 sudo dd if=/dev/zero of=$SD_ROOT_FOLDER count=120000
21 sync
22 echo "----- First sector: msdos-----"
23 # First sector: msdos
24 sudo parted $SD_ROOT_FOLDER mklabel msdos
25 echo "----- copy sunxi-spl.bin binaries-----"
26 #copy sunxi-spl.bin binaries
27 sudo dd if=${HOME_DIR}/workspace/nano/buildroot/output/images/sunxi-
   -spl.bin of=$SD_ROOT_FOLDER bs=512 seek=16
28 echo "----- copy u-boot-----"
29 #copy u-boot
30 sudo dd if=${HOME_DIR}/workspace/nano/buildroot/output/images/u-
   boot.itb of=$SD_ROOT_FOLDER bs=512 seek=80
31 echo "----- 1st partition: 64MiB-----"
32 # 1st partition: 64MiB: (163840-32768)*512/1024 = 64MiB
33 sudo parted $SD_ROOT_FOLDER mkpart primary ext4 32768s 163839s
34 echo "----- 2nd partition: 1GiB-----"
35 # 2nd partition: 1GiB: (4358144-163840)*512/1024 = 2GiB
36 sudo parted $SD_ROOT_FOLDER mkpart primary ext4 163840s 4358143s
37 sudo mkfs.ext4 $SD_ROOT_PART2 -L rootfs
38 sudo mkfs.ext4 $SD_ROOT_PART1 -L BOOT # Change line "-L" is for
   Volume Label
39 sync
40 echo "----- 3rd partition: 2GiB-----"
41 # 3rd partition: 2GiB: (9338879-5242880)*512/1024 = 2GiB
42 sudo parted $SD_ROOT_FOLDER mkpart primary ext4 5242880s 9338879s
43 sudo mkfs.ext4 $SD_ROOT_PART3 -L LUKS
44 echo "----- copy kernel, flattened device tree, boot.scr ----"
45 #copy kernel, flattened device tree, boot.scr
46 sudo mount $SD_ROOT_PART1 $LOCAL_MNT_POINT
47 sudo cp ${HOME_DIR}/workspace/nano/buildroot/output/images/Image

```

```

$LOCAL_MNT_POINT
48 sudo cp ${HOME_DIR}/workspace/nano/buildroot/output/images/nanopi-
    neo-plus2.dtb $LOCAL_MNT_POINT
49 sudo cp ${HOME_DIR}/workspace/nano/buildroot/output/images/boot.scr
    $LOCAL_MNT_POINT
50 sudo cp ${HOME_DIR}/uInitrd $LOCAL_MNT_POINT
51 sync
52 echo "----- Rename 1st partition to BOOT -----"
53 #Rename 1st partition to BOOT
54 sudo umount $SD_ROOT_PART1
55 sudo e2label $SD_ROOT_PART1 BOOT
56 echo "----- copy rootfs -----"
57 #copy rootfs
58 sudo dd if=${HOME_DIR}/workspace/nano/buildroot/output/images/
    rootfs.ext4 of=$SD_ROOT_PART2
59 echo "----- #Resize and rename 2nd partition to rootf -----"
60 # Resize and rename 2nd partition to rootfs
61 # check if the partition must be mounted
62 sudo e2fsck -f $SD_ROOT_PART2
63 sudo resize2fs $SD_ROOT_PART2
64 sudo e2label $SD_ROOT_PART2 rootfs
65
66 DEVICE=/dev/sdb3
67 PATH_Luks=/home/lmi/VM_SeS/luks_header
68 echo "----- Unmount device -----"
69 sudo umount $DEVICE
70 echo "----- Crypting partition -----"
71 sudo cryptsetup --debug --pbkdf pbkdf2 luksFormat $DEVICE -q --key-
    file ${PATH_Luks}/rand_key.txt
72 echo "----- First Dump -----"
73 sudo cryptsetup luksDump $DEVICE
74 echo "----- Add Key -----"
75 sudo cryptsetup luksAddKey $DEVICE --key-file ${PATH_Luks}/rand_key
    .txt
76 echo "----- Second Dump -----"
77 sudo cryptsetup luksDump $DEVICE
78 echo "----- Open -----"
79 sudo cryptsetup --debug open --type luks $DEVICE usrfs1 --key-file
    ${PATH_Luks}/rand_key.txt
80 echo "----- mkfs.ext4 LUKS -----"
81 sudo mkfs.ext4 /dev/mapper/usrfs1 -L LUKS
82 echo "----- mount /dev/mapper/usrfs1 -----"
83 sudo mount /dev/mapper/usrfs1 /mnt/usrfs
84 echo "----- dd -----"
85 sudo dd if=/home/lmi/workspace/nano/buildroot/output/images/rootfs.
    ext4 of=/dev/mapper/usrfs1 bs=4M
86 echo "----- Unmount device -----"
87 sudo umount /dev/mapper/usrfs1
88 echo "----- Close -----"
89 sudo cryptsetup close usrfs1
90 echo "----- DONE -----"

```

Une fois la carte microSD flashé. Il est possible démarrer le NanoPI et de le laisser "booter" normalement. Une fois que le initramfs démarre, il affiche une console où il est possible d'exécuter des commandes. La console de initramfs s'affiche, car dans le script de génération du initramfs nous utiliser la commande `exec sh`.

La figure 29 illustre la console d'initramfs.

```
[ 3.832248] Freeing unused kernel memory: 1200k
[ 3.832248] Run /init as init process
[ 3.875127] EXT4-fs (mmcblk0p2): mounted filesystem with ordered data mode. Opts: (null)
sh: can't access tty; job control turned off
/ # [ 10.177746] random: crng init done

/ #
/ #
/ # pwd
/ #
/ # exec switch_root /newroot /sbin/init
[ 1079.271016] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)
kernel.randomize_va_space = 2
net.ipv4.conf.lo.rp_filter = 1
net.ipv4.conf.eth0.rp_filter = 1
net.ipv4.conf.lo.accept_source_route = 0
```

FIGURE 29 – Illustration de la console de initramfs

Pour démarrer rootfs sur la troisième partition, il faut utiliser la commande `exec switch_root /newroot /sbin/init` qui permet de changer de racine (root) et ainsi de changer de partition.

Il est encore nécessaire de contrôler que le changement de partition a bien été réalisé. Pour cela nous contrôlons si le répertoire `usrfs`, qui correspond à la troisième partition, contient bien le rootfs que nous avons copié précédemment. La figure 30 illustre bien la présence du rootfs dans le `usrfs` :

```
# cd /mnt/usrfs/
# ls
bin          lib          lost+found  opt          run          tmp
dev          lib64        media       proc         sbin         usr
etc          linuxrc      mnt         root         sys         var
# exit
```

FIGURE 30 – Contrôle du contenu de usrfs (partition 3)

3.6 Partition Initramfs-LUKS

Maintenant que nous avons réussi à démarrer Linux sur une autre partition grâce à initramfs, il reste encore à compléter la bibliothèque du initramfs afin qu'il puisse utiliser les commande de *cryptsetup* afin de pouvoir décrypter une partition LUKS.

1. Depuis la console de **initramfs**, monter la partition LUKS
2. Démarrer manuellement la commande `exec switch_root` de la partition cryptée
3. Écrire un script qui démarre automatiquement la partition **rootfs** cryptée

Afin d'éviter des problèmes de dépassement de ressource mémoire, nous réalisons la partition LUKS crypté avec une seule passphrase. Pour réaliser cela nous avons suivi les étapes suivantes :

1. Ajouter les libraires pour utiliser cryptsetup (`ldd cryptsetup`) sur le NanoPi
2. Ajouter au sbin de initramfs le fichier de target `/usr/sbin`
3. Ajouter dossier `/run`

4. Changer le `/init` avec `cryptsetup open` et `/dev/mapper/urfs1`
5. Supprimer le fichier `$30luks` de la copie de la partition 2
6. Copier le keyfile à la racine de `initramfs`

Nous avons donc modifié le script `init_ramfs.sh` afin d'y ajouter toutes les bibliothèques nécessaires, l'exécution de ce script donne le dossier `ramfs` illustré par la figure 31 :

```

├── etc
├── home
├── init
├── lib
│   ├── ld-2.31.so
│   └── ld-linux-aarch64.so.1 -> ../lib64/ld-2.31.so
├── lib64
│   ├── ld-2.31.so
│   ├── ld-linux-aarch64.so.1 -> ../lib64/ld-2.31.so
│   ├── libatomic.so.1 -> ../lib64/libatomic.so.1.2.0
│   ├── libatomic.so.1.2.0
│   ├── libblkid.so.1 -> ../lib64/libblkid.so.1.1.0
│   ├── libblkid.so.1.1.0
│   ├── libc-2.31.so
│   ├── libc.so.6 -> libc-2.31.so
│   ├── libdl-2.31.so
│   ├── libdl.so.2 -> ../lib64/libdl-2.31.so
│   ├── libm-2.31.so
│   ├── libm.so.6 -> ../lib64/libm-2.31.so
│   ├── libpthread-2.31.so
│   ├── libpthread.so.0 -> ../lib64/libpthread-2.31.so
│   ├── libresolv-2.31.so
│   ├── libresolv.so.2 -> libresolv-2.31.so
│   ├── librt-2.31.so
│   ├── librt.so.1 -> ../lib64/librt-2.31.so
│   ├── libuuid.so.1 -> ../lib64/libuuid.so.1.3.0
│   └── libuuid.so.1.3.0
├── newroot
├── proc
├── root
├──/sbin
│   └── switch_root -> ../bin/busybox
├── sys
├── usr
│   └── lib64
│       ├── libargon2.so -> ../lib64/libargon2.so.1
│       ├── libargon2.so.1
│       ├── libcrypto.so -> ../lib64/libcrypto.so.1.1
│       ├── libcrypto.so.1.1
│       ├── libcryptsetup.so.12 -> ../lib64/libcryptsetup.so.12.6.0
│       ├── libcryptsetup.so.12.6.0
│       ├── libdevmapper.so -> ../lib64/libdevmapper.so.1.02
│       ├── libdevmapper.so.1.02
│       ├── libjson-c.so.5 -> ../lib64/libjson-c.so.5.1.0
│       ├── libjson-c.so.5.1.0
│       ├── libpopt.so.0 -> ../lib64/libpopt.so.0.0.1
│       ├── libpopt.so.0.0.1
│       ├── libssl.so -> ../lib64/libssl.so.1.1
│       └── libssl.so.1.1
13 directories, 63 files
[lmi@fedora ramfs]$
```

FIGURE 31 – *Ramfs* avec l'ajout des bibliothèques nécessaires à *cryptsetup*

Pour monter la partition LUKS depuis la console de *Initramfs*, il faut se connecter à minicom puis laissé le NanoPi démarrer. Une fois que le NanoPi à démarré, la console de *Initramfs* apparait, il faut ensuite utiliser la commande suivante afin de monter la partition LUKS :

```
1 exec switch_root /newroot /sbin/init
```

Pour réaliser un montage automatique de la partition LUKS, il est possible d'intègre directement la commande mentionné précédemment dans le script qui génère la partition *Initramfs*. Le script complet est le suivant :

```
1 #!/bin/sh
2 ROOTFSLOC=ramfs
3 HOME=/home/lmi
4 echo "----- Begin -----"
5 cd $HOME
6 echo "----- Remove old directory -----"
7 sudo rm -rf ramfs/
8 sudo rm -rf Output/
9 echo "----- Creat directory -----"
10 mkdir Output
11 mkdir $ROOTFSLOC
12 mkdir -p $ROOTFSLOC/{bin,dev,etc,home,lib,lib64,usr,newroot,proc,
    root,sbin,sys,run}
13 mkdir $ROOTFSLOC/usr/lib64
14 echo "----- Cpy / passphrase -----"
15 cd $ROOTFSLOC
16 cp $HOME/VM_SeS/luks_header/rand_key.txt .
17 cd ..
18 echo "----- Cpy /dev -----"
19 cd $ROOTFSLOC/dev
20 sudo mknod null c 1 3
21 #sudo mknod tty c 5 0
22 sudo mknod console c 5 1
23 sudo mknod random c 1 8
24 sudo mknod urandom c 1 9
25 sudo mknod mmcblk0p b 179 0
26 sudo mknod mmcblk0p1 b 179 1
27 sudo mknod mmcblk0p2 b 179 2
28 sudo mknod mmcblk0p3 b 179 3
29 sudo mknod mmcblk0p4 b 179 4
30 sudo mknod ttyS0 c 4 64
31 sudo mknod ttyS1 c 4 65
32 sudo mknod ttyS2 c 4 66
33 sudo mknod ttyS3 c 4 67
34 echo "----- Cpy /bin -----"
35 cd ../bin
36 cp ~/workspace/nano/buildroot/output/target/bin/busybox .
37 ln -s busybox ls
38 ln -s busybox mkdir
39 ln -s busybox ln
40 ln -s busybox mknod
41 ln -s busybox mount
42 ln -s busybox umount
43 ln -s busybox sh
44 ln -s busybox sleep
```

```
45 ln -s busybox dmesg
46 cp ~/workspace/nano/buildroot/output/target/usr/bin/strace .
47 echo "----- Cpy /sbin -----"
48 cd ../sbin
49 ln -s ../bin/busybox switch_root
50 cp ~/workspace/nano/buildroot/output/target/usr/sbin/cryptsetup .
51 echo "----- Cpy /lib64 -----"
52 cd ../lib64
53 cp ~/workspace/nano/buildroot/output/target/lib64/ld-2.31.so .
54 cp ~/workspace/nano/buildroot/output/target/lib64/libresolv-2.31.so
.
55 cp ~/workspace/nano/buildroot/output/target/lib64/libc-2.31.so .
56 ln -s libresolv-2.31.so libresolv.so.2
57 ln -s libc-2.31.so libc.so.6
58 ln -s ../lib64/ld-2.31.so ld-linux-aarch64.so.1
59 # Ajout depuis ~/target/lib64
60 cp ~/workspace/nano/buildroot/output/target/lib64/libm-2.31.so .
61 cp ~/workspace/nano/buildroot/output/target/lib64/librt-2.31.so .
62 cp ~/workspace/nano/buildroot/output/target/lib64/libdl-2.31.so .
63 cp ~/workspace/nano/buildroot/output/target/lib64/libpthread-2.31.
so .
64 cp ~/workspace/nano/buildroot/output/target/lib64/libatomic.so
.1.2.0 .
65 cp ~/workspace/nano/buildroot/output/target/lib64/libuuid.so.1.3.0
.
66 cp ~/workspace/nano/buildroot/output/target/lib64/libblkid.so.1.1.0
.
67
68 ln -s ../lib64/libm-2.31.so libm.so.6
69 ln -s ../lib64/librt-2.31.so librt.so.1
70 ln -s ../lib64/libdl-2.31.so libdl.so.2
71 ln -s ../lib64/libpthread-2.31.so libpthread.so.0
72 ln -s ../lib64/libatomic.so.1.2.0 libatomic.so.1
73 ln -s ../lib64/libuuid.so.1.3.0 libuuid.so.1
74 ln -s ../lib64/libblkid.so.1.1.0 libblkid.so.1
75 echo "----- Cpy /usr/lib64 -----"
76 cd ../usr/lib64
77
78 cp ~/workspace/nano/buildroot/output/target/usr/lib64/libjson-c.so
.5.1.0 .
79 cp ~/workspace/nano/buildroot/output/target/usr/lib64/libcrypto.so
.1.1 .
80 cp ~/workspace/nano/buildroot/output/target/usr/lib64/libssl.so.1.1
.
81 cp ~/workspace/nano/buildroot/output/target/usr/lib64/libargon2.so
.1 .
82 cp ~/workspace/nano/buildroot/output/target/usr/lib64/libdevmapper.
so.1.02 .
83 cp ~/workspace/nano/buildroot/output/target/usr/lib64/libpopt.so
.0.0.1 .
84 cp ~/workspace/nano/buildroot/output/target/usr/lib64/libcryptsetup
.so.12.6.0 .
85
86 ln -s ../lib64/libjson-c.so.5.1.0 libjson-c.so.5
87 ln -s ../lib64/libcrypto.so.1.1 libcrypto.so
88 ln -s ../lib64/libssl.so.1.1 libssl.so
```



```

89 ln -s ../lib64/libargon2.so.1 libargon2.so
90 ln -s ../lib64/libdevmapper.so.1.02 libdevmapper.so
91 ln -s ../lib64/libpopt.so.0.0.1 libpopt.so.0
92 ln -s ../lib64/libcryptsetup.so.12.6.0 libcryptsetup.so.12
93 cd ..
94 echo "----- Cpy /lib -----"
95 cd ../lib
96 cp ~/workspace/nano/buildroot/output/target/lib64/ld-2.31.so .
97 ln -s ../lib64/ld-2.31.so ld-linux-aarch64.so.1
98 echo "----- Create /init -----"
99 cd ..
100 cat > init << endofinput
101 #!/bin/busybox sh
102 mount -t proc none /proc
103 mount -t sysfs none /sys
104
105 cryptsetup --debug open --type luks /dev/mmcblk0p3 usrfs1 --key-
    file=rand_key.txt
106 echo "----- mount usrfs1 -----"
107 mount -t ext4 /dev/mapper/usrfs1 /newroot
108 mount -n -t devtmpfs devtmpfs /newroot/dev
109 echo "----- exec -----"
110 #exec sh
111 exec switch_root /newroot /sbin/init
112 endofinput
113 #####
114 chmod 755 init
115 cd ..
116 sudo chown -R 0:0 $ROOTFSLOC
117 echo "----- cpio / gzip / mkimage -----"
118 cd $ROOTFSLOC
119 find . | cpio --quiet -o -H newc > ../Output/Initrd
120 cd ../Output
121 gzip -9 -c Initrd > Initrd.gz
122 mkimage -A arm -T ramdisk -C none -d Initrd.gz uInitrd
123 echo "----- DONE -----"

```

Une fois le initramfs créé, il reste encore à le flasher sur la carte microSD. Pour ce faire, le script de génération de la carte microSD a été modifié en conséquence. Le listing suivant illustre le script final qui a été utilisé.

```

1 #!/bin/sh
2 # These 3 variables must be modified according to user setup
3 # They are not subject to change(check SD root folder just in case)
4 # Default values according to course slides (1_NanoPi.pdf)
5 SD_ROOT_FOLDER=/dev/sdb          # SD card root directory, unmounted
6 LOCAL_MNT_POINT=/run/media/lmi  #default SD card mount point
7 HOME_DIR=/home/lmi              # user home directory
8
9 SD_ROOT_PART1=${SD_ROOT_FOLDER}1
10 SD_ROOT_PART2=${SD_ROOT_FOLDER}2
11 SD_ROOT_PART3=${SD_ROOT_FOLDER}3
12 #SD_ROOT_PART4=${SD_ROOT_FOLDER}4
13 echo "----- #umount -----"
14 umount $SD_ROOT_PART1
15 umount $SD_ROOT_PART2

```

```
16 umount $SD_ROOT_PART3
17 #umount $SD_ROOT_PART4
18 echo "----- #initialize 480MiB to 0-----"
19 #initialize 480MiB to 0
20 sudo dd if=/dev/zero of=$SD_ROOT_FOLDER count=120000
21 sync
22 echo "----- # First sector: msdos-----"
23 # First sector: msdos
24 sudo parted $SD_ROOT_FOLDER mklabel msdos
25 echo "----- #copy sunxi-spl.bin binaries-----"
26 #copy sunxi-spl.bin binaries
27 sudo dd if=${HOME_DIR}/workspace/nano/buildroot/output/images/sunxi-
    spl.bin of=$SD_ROOT_FOLDER bs=512 seek=16
28 echo "----- #copy u-boot-----"
29 #copy u-boot
30 sudo dd if=${HOME_DIR}/workspace/nano/buildroot/output/images/u-
    boot.itb of=$SD_ROOT_FOLDER bs=512 seek=80
31 echo "----- # 1st partition: 64MiB-----"
32 # 1st partition: 64MiB: (163840-32768)*512/1024 = 64MiB
33 sudo parted $SD_ROOT_FOLDER mkpart primary ext4 32768s 163839s
34 echo "----- # 2nd partition: 1GiB-----"
35 # 2nd partition: 1GiB: (4358144-163840)*512/1024 = 2GiB
36 sudo parted $SD_ROOT_FOLDER mkpart primary ext4 163840s 4358143s
37 sudo mkfs.ext4 $SD_ROOT_PART2 -L rootfs
38 sudo mkfs.ext4 $SD_ROOT_PART1 -L BOOT # Change line "-L" is for
    Volume Label
39 sync
40 echo "----- # 3rd partition: 2GiB-----"
41 # 3rd partition: 2GiB: (9338879-5242880)*512/1024 = 2GiB
42 sudo parted $SD_ROOT_FOLDER mkpart primary ext4 5242880s 9338879s
43 sudo mkfs.ext4 $SD_ROOT_PART3 -L LUKS
44 echo "----- #copy kernel, flattened device tree, boot.scr-----"
45 #copy kernel, flattened device tree, boot.scr
46 sudo mount $SD_ROOT_PART1 $LOCAL_MNT_POINT
47 sudo cp ${HOME_DIR}/workspace/nano/buildroot/output/images/Image
    $LOCAL_MNT_POINT
48 sudo cp ${HOME_DIR}/workspace/nano/buildroot/output/images/nanopi-
    neo-plus2.dtb $LOCAL_MNT_POINT
49 sudo cp ${HOME_DIR}/workspace/nano/buildroot/output/images/boot.scr
    $LOCAL_MNT_POINT
50 sudo cp ${HOME_DIR}/Output/uInitrd $LOCAL_MNT_POINT
51 sync
52 echo "----- #Rename 1st partition to BOOT -----"
53 #Rename 1st partition to BOOT
54 sudo umount $SD_ROOT_PART1
55 sudo e2label $SD_ROOT_PART1 BOOT
56 echo "----- #copy rootfs -----"
57 #copy rootfs
58 sudo dd if=${HOME_DIR}/workspace/nano/buildroot/output/images/
    rootfs.ext4 of=$SD_ROOT_PART2
59 echo "----- #Resize and rename 2nd partition to rootf -----"
60 # Resize and rename 2nd partition to rootfs
61 # check if the partition must be mounted
62 sudo e2fsck -f $SD_ROOT_PART2
63 sudo resize2fs $SD_ROOT_PART2
64 sudo e2label $SD_ROOT_PART2 rootfs
```

```

65 DEVICE=/dev/sdb3
66 PATH_Luks=/home/lmi/VM_SeS/luks_header
67 echo "----- Unmount device -----"
68 sudo umount $DEVICE
69 echo "----- Crypting partition -----"
70 sudo cryptsetup --debug --pbkdf pbkdf2 luksFormat $DEVICE -q --key-
    file ${PATH_Luks}/rand_key.txt
71 echo "----- First Dump -----"
72 sudo cryptsetup luksDump $DEVICE
73 echo "----- Add Key -----"
74 sudo cryptsetup luksAddKey $DEVICE --key-file ${PATH_Luks}/rand_key
    .txt
75 echo "----- Second Dump -----"
76 sudo cryptsetup luksDump $DEVICE
77 echo "----- Open -----"
78 sudo cryptsetup --debug open --type luks $DEVICE usrfs1 --key-file
    ${PATH_Luks}/rand_key.txt
79 echo "----- mkfs.ext4 LUKS -----"
80 sudo mkfs.ext4 /dev/mapper/usrfs1 -L LUKS
81 echo "----- mount /dev/mapper/usrfs1 -----"
82 sudo mount /dev/mapper/usrfs1 /mnt/usrfs
83 echo "----- dd -----"
84 sudo dd if=/home/lmi/workspace/nano/buildroot/output/images/rootfs.
    ext4 of=/dev/mapper/usrfs1 bs=4M
85 echo "----- Unmount device -----"
86 sudo umount /dev/mapper/usrfs1
87 echo "----- Close -----"
88 sudo cryptsetup close usrfs1
89 echo "----- DONE -----"

1 # cryptsetup 2.3.4 processing "cryptsetup --debug open --type luks
    /dev/mmcblk0p3 usrfs1 --"
2 # Running comTERM handl[ 3.983043] random: cryptsetup:
    uninitialized urandom read (4 byt)
3 # Unblocking interruption on signal.
4 # Allocating context for crypt device /dev/mmcblk0p3.
5 # Trying to open and read device /dev/mmcblk0p3 with direct-io.
6 # Initialising device-mapper backend library.
7 # Trying to load any crypt type from device /dev/mmcblk0p3.
8 # Crypto backend (OpenSSL 1.1.1l 24 Aug 2021) initialized in
    cryptsetup library version 2..
9 # Detected kernel Linux 5.8.6 aarch64.
10 # Loading LUKS2 header (repair disabled).
11 # Acquiring read lock for device /dev/mmcblk0p3.
12 WARNING: Locking directory /run/cryptsetup is missing!
13 # Opening lock resource file /run/cryptsetup/L_179:3
14 # Verifying lock handle for /dev/mmcblk0p3.
15 # Device /dev/mmcblk0p3 READ lock taken.
16 # Trying to read primary LUKS2 header at offset 0x0.
17 # Opening locked device /dev/mmcblk0p3
18 # Verifying locked device handle (bdev)
19 # LUKS2 header version 2 of size 16384 bytes, checksum sha256.
20 # Checksum:6
    b58b128b9826b96bac3034eb7e5cec76909fe3ceea806c835b0170efc1f8dbc
    (on-disk)
21 # Checksum:6

```

```
        b58b128b9826b96bac3034eb7e5cec76909fe3ceea806c835b0170efc1f8dbc
        (in-memory)
22 # Trying to read secondary LUKS2 header at offset 0x4000.
23 # Reusing open ro fd on device /dev/mmcblk0p3
24 # LUKS2 header version 2 of size 16384 bytes, checksum sha256.
25 # Checksum:95
        f9223b2f50dd1179d6a4a92e512a85a13214788ba174e5a4d76c7f8fd772a8 (
        on-disk)
26 # Checksum:95
        f9223b2f50dd1179d6a4a92e512a85a13214788ba174e5a4d76c7f8fd772a8 (
        in-memory)
27 # Device size 2097152000, offset 16777216.
28 # Device /dev/mmcblk0p3 READ lock released.
29 # Not enough physical memory detected, PBKDF max memory decreased
        from 1048576kB to 242494k.
30 # PBKDF argon2i, time_ms 2000 (iterations 0), max_memory_kb 242494,
        parallel_threads 4.
31 # Activating volume usrfs1 using token -1.
32 # File descriptor passphrase entry requested.
33 # Activating volume usrfs1 [keyslot -1] using passphrase.
34 # Creating directory "/dev/mapper"
35 # Creating device /dev/mapper/control (10, 236)
36 # dm version [ opencount flush ] [16384] (*1)
37 # dm versions [ opencount flush ] [16384] (*1)
38 # Detected dm-ioct1 version 4.42.0.
39 # Detected dm-crypt version 1.21.0.
40 # Device-mapper backend running with UDEV support disabled.
41 # dm status usrfs1 [ opencount noflush ] [16384] (*1)
42 # Keyslot 0 priority 1 != 2 (required), skipped.
43 # Trying to open LUKS2 keyslot 0.
44 # Reading keyslot area [0x8000].
45 # Acquiring read lock for device /dev/mmcblk0p3.
46 # Opening lock resource file /run/cryptsetup/L_179:3
47 # Verifying lock handle for /dev/mmcblk0p3.
48 # Device /dev/mmcblk0p3 READ lock taken.
49 # Reusing open ro fd on device /dev/mmcblk0p3
50 # Device /dev/mmcblk0p3 READ lock released.
51 # Verifying key from keyslot 0, digest 0.
52 # Loading key (64 bytes, type logon) in thread keyring.
53 # dm versions [ opencount flush ] [16384] (*1)
54 # dm status usrfs1 [ opencount noflush ] [16384] (*1)
55 # Calculated device size is 4063232 sectors (RW), offset 32768.
56 # DM-UUID is CRYPT-LUKS2-529fb89618874a539f2e76ec510751d3-usrfs1
57 # dm create usrfs1 CRYPT-LUKS2-529fb89618874a539f2e76ec510751d3-
        usrfs1 [ opencount flush ] )
58 # dm reload (253:0) [ opencount flush securedata ] [16384] (*1)
59 # dm resume usrfs1 [ opencount flush securedata ] [16384] (*1)
60 # usrfs1: Stacking NODE_ADD [ 11.959201] EXT4-fs (dm-0): mounted
        filesystem with ordered )
61 (253,0) 0:0 0600
62 # usrfs1: Stacking NODE_READ_AHEAD 256 (flags=1)
63 # usrfs1: Processing NODE_ADD (253,0) 0:0 0600
64 # Created /dev/mapper/usrfs1
65 # usrfs1: Processing NODE_READ_AHEAD 256 (flags=1)
66 # usrfs1 (253:0): read ahead is 256
67 # usrfs1: retaining kernel read ahead of 256 (requested 256)
```

```
68 Key slot 0 unlocked.
69 # Releasing crypt device /dev/mmcblk0p3 context.
70 # Releasing device-mapper backend.
71 # Closing read only fd for /dev/mmcblk0p3.
72 # Unlocking memory.
73 Command successful.
74 ----- mount usrfs1-----
75 ----- exec-----
76 Starting syslogd: OK
77 Starting klogd: OK
78 Running sysctl: OK
79 Starting mdev... OK
80 Saving random seed: [ 12.972851] random: dd: uninitialized
    urandom read (512 bytes read)
81 OK
82 Starting haveged: haveged: command socket is listening at fd 3
83 OK
84 Starting dropbear sshd: [ 13.319741] random: crng init done
85 OK
86
87 Welcome to FriendlyARM Nanopi NEO Plus2
88 buildroot login: root
89 #
```

Sur le listing 3.6, on peut voir la séquence de boot à partir du lancement de initRAMSfs avec la commande cryptsetup open puis à la ligne 72,73 on voit que la commande a été correctement réalisée. Nous pouvons ensuite se connecter en root sur le NanoPi.

4 Laboratoire 7 TPM

Le laboratoire 7 sur les TPM (Trusted Platform Module) est sujet nouveau qui ne nécessite pas l'utilisation du nanoPi. Pour ce laboratoire, nous allons utiliser un logiciel afin de simuler le fonctionnement du TPM. Le logiciel est SWTPM pour software TPM est nous permet réaliser le laboratoire sans avoir de matériel supplémentaire. Ce logiciel utilise l'interface loopback pour établir une communication TCP sur un port précis. Il faut donc commencer par démarrer un serveur SWTPM pour ensuite utiliser les commandes qui seront décrites dans la suite du laboratoire.

4.1 Installation des outils TPM2

Nous avons commencé ce laboratoire par l'installation de l'outil de développement sur notre machine virtuelle. Pour ce faire, nous avons suivi la marche à suivre visible dans le listing suivant.

```

1 git clone https://github.com/stefanberger/swtpm.git
2
3 cd swtpm
4
5 sudo dnf -y install libtasn1-devel expect socat python3-twisted
   fuse-devel glib2-devel gnutls-devel gnutls-utils gnutls json-
   glib-devel
6
7 sudo dnf install libtpms-devel
8 sudo dnf install libseccomp-devel
9
10 ./autogen.sh --with-openssl --prefix=/usr
11
12 make -j4
13 make -j4 check
14
15 sudo make install
16 sudo dnf install tpm2-tss
17 sudo dnf install tpm2-tools

```

Listing 3 – Commande pour l'installation d'un virtual TPM

Puis une fois que toutes ces commandes ont bien été réalisées, comme illustré sur la figure 32, il faut encore ajouter dans le fichier `/home/lmi/.bash_profile` la commande `export TPM2TOOLS_TCTI="swtpm:port=2321"` pour activer le port tcp et le node file `/dev/tpmrm0`.

```

=====
Testsuite summary for swtpm 0.8.0
=====
# TOTAL: 69
# PASS: 56
# SKIP: 13
# XFAIL: 0
# FAIL: 0
# XPASS: 0
# ERROR: 0
=====

```

FIGURE 32 – Vérification de l'installation de swtpm

Enfin pour démarrer le serveur il faut créer le dossier suivant `/home/lmi/server/tpm/swtpm2` afin de pouvoir lancer le server swtpm dans ce dossier avec la commande

```
1 swtpm socket --tpmstate dir=/home/lmi/server/tpm/swtpm2 --tpm2 --
  server type=tcp,port=2321 --ctrl type=tcp,port=2322 --flags not-
  need-init,startup-clear
```

4.2 Question 1 - Créer des "load-save primary keys"

Une fois tous ces outils correctement installés et fonctionnels, nous sommes passés à la première question. Cette question est une introduction au principe du *Trusted Platform Module*. Cela en exécutant les principales commandes utilisées sur un TPM.

Les sous-parties de la question sont les suivantes

- Créer une clé primaire avec RSA2048bits dans la hiérarchie du propriétaire du module.
- Vérifier les clés créées dans la RAM ainsi que dans la NV-RAM. (mémoire flash)
- Effacer toutes les clés de la RAM et recréer la clé primaire du début.
- Sauvegarder la clé dans la mémoire non-volatile
- Vérifier les clés créées dans la RAM ainsi que dans la NV-RAM.

Pour ce faire nous avons écrit un petit script qui exécute tous ces commandes et affiche les résultats demandés. Le script est visible dans le listing suivant :

```
1 echo "----- Begin -----"
2 echo "----- Create primary owner key -----"
3 tpm2_createprimary -C o -G rsa2048 -c o_primary.ctx
4 echo "----- Show transient key (RAM) -----"
5 tpm2_getcap handles-transient
6 echo "----- Flush current key -----"
7 tpm2_flushcontext -t
8 echo "----- Create primary owner key -----"
9 tpm2_createprimary -C o -G rsa2048 -c o_primary.ctx
10 echo "----- Save owner key in NV-RAM -----"
11 tpm2_evictcontrol -c o_primary.ctx
12 echo "----- Show transient key (RAM) -----"
13 tpm2_getcap handles-transient
14 echo "----- Show persistent key (NV-RAM) -----"
15 tpm2_getcap handles-persistent
16 echo "----- DONE -----"
```

Listing 4 – Script de la question 1 du laboratoire TPM

Voici le résultat obtenu avec le script de la question 1. On peut observer que nous commençons par effacer toutes les clés qui ont été créées précédemment. Puis nous créons la clé primaire du propriétaire pour ensuite la stocker dans la mémoire non-volatile.

```
[lmi@fedora Question1]$ ./Q1.sh && ls -al
----- Begin -----
----- Flush previous code -----
----- Create primary owner key -----
name-alg:
  value: sha256
  raw: 0xb
attributes:
  value: fixedtpm|fixedparent|sensitivedataorigin|
  userwithauth|restricted|decrypt
  raw: 0x30072
type:
  value: rsa
  raw: 0x1
exponent: 65537
bits: 2048
scheme:
  value: null
  raw: 0x10
scheme-halg:
  value: (null)
  raw: 0x0
sym-alg:
  value: aes
  raw: 0x6
sym-mode:
  value: cfb
  raw: 0x43
sym-keybits: 128
rsa: a5f50d3a5c54a5603b5327498a7b3ebd16e25247aeb4a9baa65a215415fd34
e88916cbfa5020904a6a462b5699a4341fa1ff11ead2a751d249dfa19f9d1b9f0c5
a281c3ff2e2ae6aebca47f30dad22ffc4c1059a60cd7de4014ce9dec06195e32b8e
b8693e06cf7bd6795b249d8ce97ac8f188f9a3b9ad1142fa2e22089c542434edb53
9d637b75a7979d15a4555e6aed86a35d8b5f536f93fba8f82f0537e97cc9dfe5a56
ac255ca92d60c7f2505441006717c81f15e3dca102fc4a540a19a9f33cf0c57c98a
7210b4159d33d54d564a9ad4eb776a245922298c26cdf54eb2ba451159bcf362f50
070ac561d719429cce1fc0eac6f03656817ba5f7181d9
----- Show transient key (RAM) -----
- 0x80000000
----- Flush current key -----
----- Create primary owner key -----
----- Save owner key in NV-RAM -----
persistent-handle: 0x81000000
action: persisted
----- Show transient key (RAM) -----
- 0x80000000
- 0x80000001
----- Show persistent key (NV-RAM) -----
- 0x81000000
----- DONE -----
total 7
drwxrwxrwx. 1 lmi lmi      0 Dec 21 15:09 .
drwxrwxrwx. 1 lmi lmi 4096 Dec 28 14:27 ..
-rwxrwxrwx. 1 lmi lmi 1602 Dec 28 14:38 o_primary.ctx
-rwxrwxrwx. 1 lmi lmi  929 Dec 28 12:06 Q1.sh
```


A la fin du script, on affiche les fichiers contenus dans le dossier pour observer la copie locale de la clé.

4.3 Question 2 - Créer des *load-save child keys*

La deuxième question va plus en profondeur dans la hiérarchie des clés du TPM. Cette fois nous devons créer une deuxième couche de clé appelée "enfant" dans la hiérarchie du propriétaire du TPM. Sur la figure 33, nous pouvons observer la structure parent-enfant que nous allons créer.



FIGURE 33 – Hiérarchie parent enfant des clés dans le TPM

Dans le script suivant, nous créons à nouveau la clé primaire afin de créer la clé enfant dans la même hiérarchie. Pour finir, nous sauvegardons la clé enfant dans la mémoire non-volatile.

```

1 echo "----- Begin -----"
2 echo "----- Create primary owner key -----"
3 tpm2_createprimary -C o -G rsa2048 -c primary
4 echo "----- Create child owner key -----"
5 tpm2_create -C primary -G rsa2048 -u child_public -r child_private
6 echo "----- Show transient key (RAM) -----"
7 tpm2_getcap handles-transient
8 echo "----- Show persistent key (NV-RAM) -----"
9 tpm2_getcap handles-persistent
10 echo "----- Save owner key in NV-RAM -----"
11 tpm2_evictcontrol -c 0x80000001
12 echo "----- Show transient key (RAM) -----"
13 tpm2_getcap handles-transient
14 echo "----- Show persistent key (NV-RAM) -----"
15 tpm2_getcap handles-persistent
16 echo "----- DONE -----"

```

Listing 5 – Script de la question 2

Le résultat obtenu à l'aide du scripte décrit est le suivant. On constate tout d'abord, la partie remise à zéro des variables en mémoire. Puis la créations des deux clés et enfin la sauvegarde de la clé enfant dans la mémoire non-volatile.

```
[lmi@fedora Question2]$ ./Q2.sh && ls -al
----- Begin -----
----- Flush previous code -----
----- Create primary owner key -----
name-alg:
  value: sha256
  raw: 0xb
attributes:
  value: fixedtpm|fixedparent|sensitivedataorigin|
  userwithauth|restricted|decrypt
  raw: 0x30072
type:
  value: rsa
  raw: 0x1
exponent: 65537
bits: 2048
scheme:
  value: null
  raw: 0x10
scheme-halg:
  value: (null)
  raw: 0x0
sym-alg:
  value: aes
  raw: 0x6
sym-mode:
  value: cfb
  raw: 0x43
sym-keybits: 128
rsa: a5f50d3a5c54a5603b5327498a7b3ebd16e25247aeb4a9baa65a215415fd34
e88916cbfa5020904a6a462b5699a4341fa1ff11ead2a751d249dfa19f9d1b9f0c5
a281c3ff2e2ae6aebca47f30dad22ff03c4c1059a60cd7de4014ce9dec06195e32b
8eb8693e06cf7bd6795b249d8ce97ac8f188f9a3b9ad1142fa2e22089c542434edb
539d637b75a7979d15a4555e6aed86a35d8b5f536f93fba8f82f0537e97cc9dfe5a
56ac255ca92d60c7f2505441006717c81f15e3dca102fc4a540a19a9f33cf0c57c9
8a7210b4159d33d54d564a9ad4eb776a245922298c26cdf54eb2ba451159bcf362f
50070ac561d719429cce1fc0eac6f003656817ba5f7181d9
----- Create child owner key -----
name-alg:
  value: sha256
  raw: 0xb
attributes:
  value: fixedtpm|fixedparent|sensitivedataorigin|
  userwithauth|decrypt|sign
  raw: 0x60072
type:
  value: rsa
  raw: 0x1
exponent: 65537
bits: 2048
scheme:
  value: null
  raw: 0x10
scheme-halg:
  value: (null)
  raw: 0x0
```

```

sym-alg:
  value: null
  raw: 0x10
sym-mode:
  value: (null)
  raw: 0x0
sym-keybits: 0
rsa: a45b2cbd9c5292a9530bd3a7686581083bd25c92425505f955eafc7af0db3d
22c87c284e57f5b5b08815ddd428b62dc8c049ef9c094ec1dd1666afb02b5c1baaf
d67453e7f138d5f426f93f3bb1b3472f36fd024880e0efaf23f973a45d951dfeba1
98c393d34de5b81a6c6165befafb7b94b395fbd5cab0f565153281d1846dcf1d0cd
819ef5ca76307f036373d7631c826d292c3232fe387d29a77405496a6d73890a6c2
3b0485f046a8922a1a0ae64233f2fc6ceda68474bc22d114d6d0389ba2ec6a7b492
fec8cc233f8de90546fac21c59591ac6a3d65090912f2e1b0d9c5917c39419711de
08f84bf70315055f4338e5c5d3f76cbafdf95a76e3f97b3
----- Show transient key (RAM) -----
- 0x80000000
- 0x80000001
---- Show persistent key (NV-RAM) ----
----- Save owner key in NV-RAM -----
persistent-handle: 0x81000000
action: persisted
----- Show transient key (RAM) -----
- 0x80000000
- 0x80000001
---- Show persistent key (NV-RAM) ----
- 0x81000000
----- DONE -----
total 15
drwxrwxrwx. 1 lmi lmi 4096 Dec 28 14:28 .
drwxrwxrwx. 1 lmi lmi 4096 Dec 28 14:27 ..
-rwxrwxrwx. 1 lmi lmi 1562 Dec 28 14:30 child_key
-rwxrwxrwx. 1 lmi lmi 224 Dec 28 14:31 child_private
-rwxrwxrwx. 1 lmi lmi 280 Dec 28 14:31 child_public
-rwxrwxrwx. 1 lmi lmi 1602 Dec 28 14:31 primary
-rwxrwxrwx. 1 lmi lmi 1057 Dec 28 14:31 Q2.sh

```

Comme pour la question 1 une fois les commandes exécutées une copie des clés se trouvent dans le dossier d'où est lancé le script.

4.4 Question 3 - Décrypter avec TPM

Une fois que nous avons commencé à comprendre comment fonctionne un TPM, nous sommes passés à la question 3, qui est une application des possibilités offertes par le TPM. Soit le décryptage des fichiers codés à l'aide d'une clé public. Il nous faut donc la clé privée afin de pouvoir retrouver le fichier d'origine comme illustré sur la figure 34.

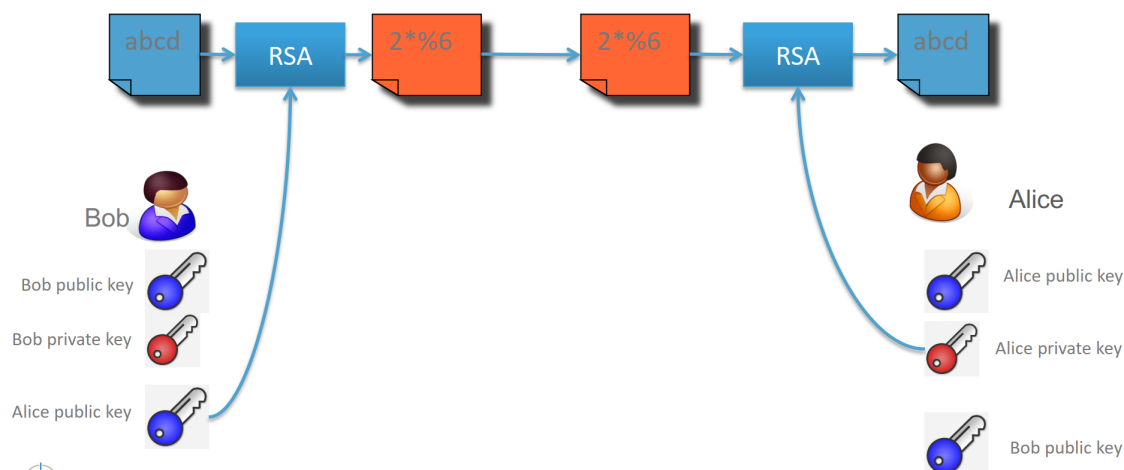


FIGURE 34 – Schéma du cryptage et du décryptage du fichier

Pour faire cela, il faut charger la clé privée dans le TPM avec la commande `tpm2_loadexternal` puis il faut décrypter le fichier avec la commande `tpm2_rsadecrypt`.

Sur le listing 4.4, on peut observer le script de réponse de la question 3.

```

1 echo "----- Begin -----"
2 echo "----- Load private key (RAM) -----"
3 tpm2_loadexternal -C n -G rsa -r rsa_key.pem -c rsa_key
4 echo "----- Show transient key (RAM) -----"
5 tpm2_getcap handles-transient
6 echo "---- Decrypt with the private key ----"
7 tpm2_rsadecrypt -c 0x80000000 -s rsaes encryptedtext -o cleartext
8 echo "----- Print decrypted text -----"
9 cat cleartext
10 echo ""
11 echo "----- DONE -----"

```

Ci-dessous se trouve le résultat de la question 3 avec le texte décrypté.

```

[lmi@fedora Question3]$ ./Q3.sh && ls -al
----- Begin -----
----- Flush previous key -----
----- Load private key (RAM) -----
name: 000b2fc470e1b4f68fc580fac5b08d8ba13e99137c29e63c1ce7517a240d4d79fdb
----- Show transient key (RAM) -----
- 0x80000000
---- Decrypt with the private key ----
----- Print decrypted text -----
Happy new year 2022

```

```
----- DONE -----
total 14
drwxrwxrwx. 1 lmi lmi 4096 Dec 28 14:36 .
drwxrwxrwx. 1 lmi lmi 4096 Dec 28 14:27 ..
-rwxrwxrwx. 1 lmi lmi 19 Dec 28 14:37 cleartext
-rwxrwxrwx. 1 lmi lmi 256 Dec 28 13:27 encryptedtext
drwxrwxrwx. 1 lmi lmi 0 Dec 28 13:43 Figures
-rwxrwxrwx. 1 lmi lmi 0 Dec 28 14:36 log3.txt
-rwxrwxrwx. 1 lmi lmi 712 Dec 28 14:24 Q3.sh
-rwxrwxrwx. 1 lmi lmi 1562 Dec 28 14:37 rsa_key
-rwxrwxrwx. 1 lmi lmi 1675 Dec 28 13:27 rsa_key.pem
```

Le text qui a été encrypté avec la clé publique est : *Happy new year 2022*

4.5 Question 4 - Politique PCR

La dernière partie du laboratoire sur les TPM est une question plus ouverte, mais surtout plus intrigante sur comment utiliser les TPM pour augmenter la sécurité d'un OS complet. Pour ce faire il nous est demandé de réfléchir comment u-boot pourrait vérifier l'intégrité du kernel Linux lors du démarrage du NanoPi. Puis dans un second temps, de réfléchir à comment modifier tout ceci lors d'une mise à jour de l'opérateur.

4.5.1 U-boot check linux

La première partie sur la mise en place d'une politique de contrôle d'un PCR se fait en 5 étapes qui vont être d'abord listées puis expliquées par un schéma et des commandes correspondantes.

- Configurer le PCR_x avec le hash du kernel Linux.
- Sauvegarder cette information dans le PCR.
- Créer une règle avec le PCR configuré.
- Sauvegarder toutes ces informations dans la mémoire non-volatile.
- Comparer la valeur du PCR_x à la valeur actuel du hash du kernel lors du démarrage.

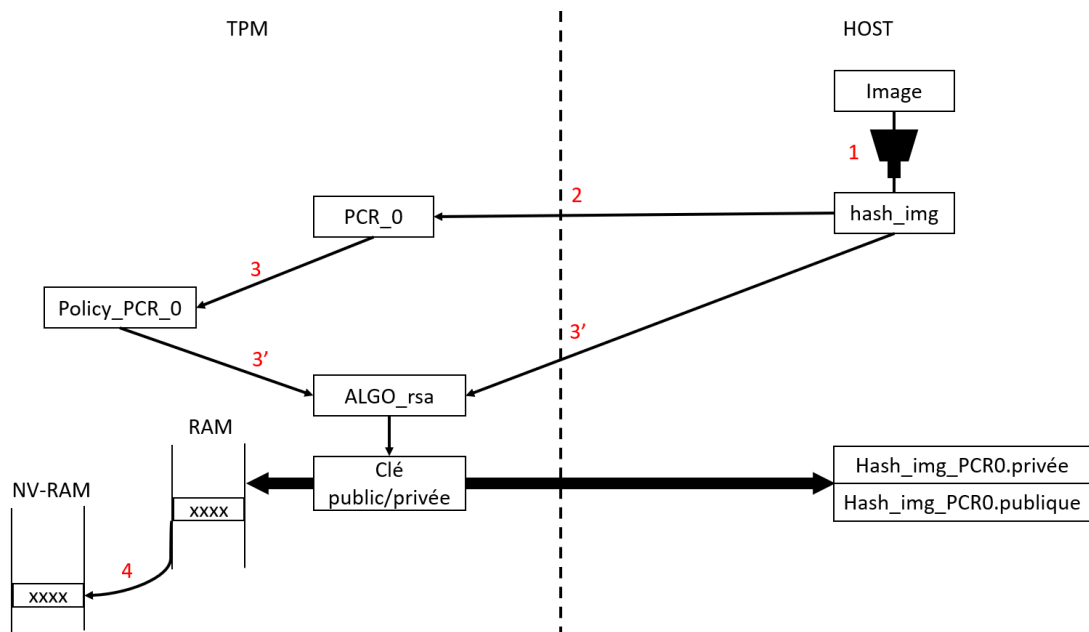


FIGURE 35 – Schéma bloc de la création d'une PCR policy

Puis lors du démarrage de l'appareil la valeur stockée dans la mémoire non-volatile du TPM est comparée avec le hash de la valeur actuel du kernel afin de vérifier s'il a été modifié. Comme le montre la figure 36.

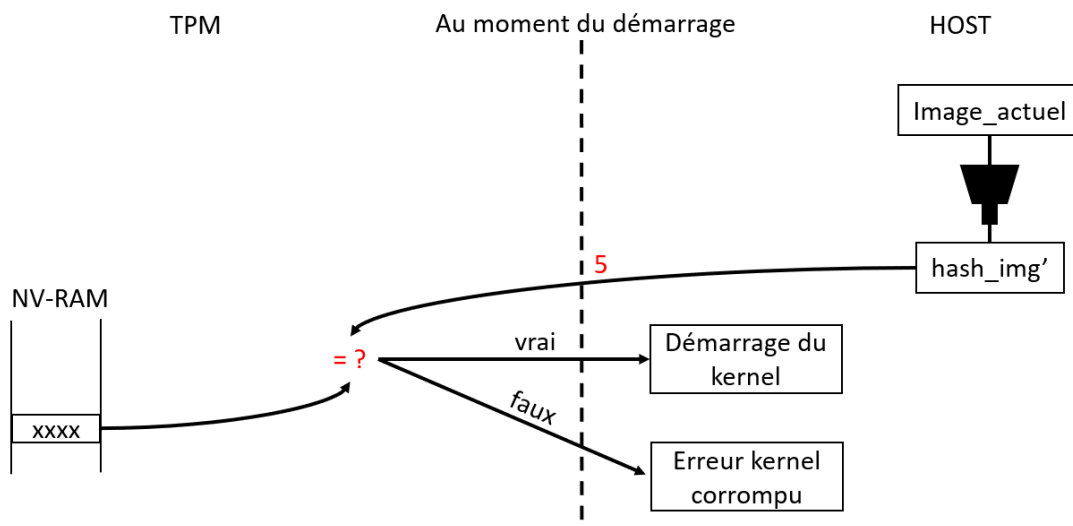


FIGURE 36 – Schéma bloc de l'utilisation d'un PCR policy

Dans le listing suivant, nous allons retrouver les principales commandes utilisées pour configurer le PCR ainsi que ce politique de contrôle.

```
1 echo "----- BEGIN -----"
2 echo "----- Calcul du hash -----"
3 sha1sum Image
4 echo "----- Effacer precedente config -----"
5 tpm2_pcrreset 0
6 echo "----- Configuration PCR 0 -----"
7 tpm2_pcrextend 0:sha1=<HASH_IMAGE>
8
9 tpm2_createprimary -C o -G rsa2048 -c primary
10 tpm2_startauthsession -S session
11 tpm2_policypcr -S session -l sha1:0 -L pcr0_policy
12 tpm2_flushcontext session
13
14 tpm2_create -C primary -g sha256 \
15 -u img_pcr0.pub -r img_pcr0.priv -i img -L pcr0_policy
16 echo "----- Sauvegarde dans la Nv-RAM -----"
17 tpm2_load -C primary -u img_pcr0.pub \
18 -r img_pcr0.priv -c img_pcr
19
20 tpm2_evictcontrol -c img_pcr0 0x81010000 -C o
21 tpm2_flushcontext session
22 echo "--- Comparaison entre PCR0 et actuel ---"
23 tpm2_startauthsession --policy-session -S session
24 tpm2_policypcr -S session -l sha1:0
25 tpm2_unseal -p session:session -c 0x81010000 > Image_OK
```

Avec cette mécanique, il serait possible de vérifier si l'image de Linux a été modifiée entre deux démarrages de l'appareil.

4.5.2 Installation d'un nouveau kernel linux

Cette deuxième partie de la question pose un dilemme sur l'utilisation même d'un tel système pour vérifier l'intégrité d'un noyau d'OS. Car lors d'une mise à jour de l'image Linux ou autre sous partie de Linux alors le hash du noyau sera forcément différent et pour ceci il faut aller modifier la politique de contrôle. Pour ce faire, nous avons identifié un moyen relativement facile si l'on connaît à l'avance le nouveau noyau et donc par la même occasion son futur hash, ce qui est souvent le cas lors d'une mise à jour. En effet, nous allons d'abord télécharger le nouvel OS avant de l'installer sur l'appareil. Nous pouvons refaire la démarche proposée dans la partie 1 en effaçant simplement l'ancienne politique. Cette version possède tout de même une faille, car si l'on laisse la possibilité de faire ce genre de manipulation, rien ne garantit qu'une personne mal intentionnée puisse faire cette manipulation pour modifier la politique avec son propre noyau modifié.

5 Conclusion

Cette série de laboratoires à permis de mettre en pratiques trois sujets différents liés à l'outil OpenSSH, le cryptage de données et les TPM.

La première partie à permis de mettre en pratique l'outil **OpenSSH** qui est un ensemble d'outils informatiques gratuit permettant des communications sécurisées par protocole **SSH**. Nous avons d'abord pu nous sensibiliser à l'importance de l'authentification des logiciels téléchargés afin d'éviter des problèmes de sécurités. Nous avons pu mettre en pratique la configuration de cet outil ainsi que son installation sur un environnement **Intel** et sur l'environnement du **NanoPI**. Nous avons aussi pu mettre en évidence les failles de sécurités que peut engendrer ce type de logiciel et comment les contrer en limitant au maximum les informations visibles et utilisable pour des attaques informatiques.

Dans la deuxième partie, nous nous sommes intéressés aux différents systèmes de fichier Linux. Nous avons pu mettre en évidence les différents types d'architecture et de comportement qui les caractérisent et ainsi pouvoir choisir au mieux le système de fichier qui convient à une application donnée. Nous avons ensuite exploré de cryptage de partition à l'aide de **LUKS**, **cryptsetup** et **dm-crypt**. Nous avons aussi réalisé la configuration d'une partition **LUKS** cryptée et contenant un **Rootfs**. Nous avons réalisé cette partie en décomposant le problème. D'abord, par la configuration d'une partition **LUKS**, puis d'une partition **LUKS** contenant un **Rootfs** puis finalement la configuration d'un **initramfs** permettant de pouvoir, décrypter et démarrer cette partition **LUKS** de manière autonome.

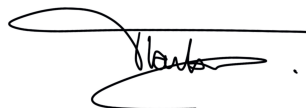
Dans dernière partie du laboratoire, nous avons pris connaissance et mis en pratique les **TPM** ("*Trusted Platform Module*"). Cela, en commençant par l'installation et la configuration de l'outil **TPM2**. nous avons ensuite créé des **clés primaires** "*primary keys*" et de **clés enfants** "*child keys*". Puis, nous avons pu comprendre le processus de décryptage à l'aide d'une TPM et finalement les politiques PCR.

6 Signatures

Lausanne le 8 avril 2022



(a) Quentin Müller



(b) Tristan Traiber

Table des figures

1	Vérification de la signature de OpenSSH	4
2	Vérification du strip après installation de OpenSSH sur intel	5
3	Arborescence du dossier d'installation pour le NanoPi	6
4	Vérification du stripage après installation de OpenSSH pour le NanoPi	6
5	Connections ssh en mode root refusée	8
6	Connections ssh en mode sshd	8
7	<i>nmap</i> de l'IP du NanoPi	9
8	<i>nmap</i> de l'IP du NanoPi sans la version de OpenSSH	9
9	Capture d'écran du contenu du fichier boot.scr	10
10	Schéma de l'utilisation des nodes files	11
11	Résultat de la commande ls /dev/root -al	12
12	Séquence d'écriture des systèmes de fichier de type B-Tree/CoW - source : <i>File Systems - Jean-Roland Schuler</i>	13
13	Séquence d'écriture des systèmes de fichier de type Log - source : <i>File Systems - Jean-Roland Schuler</i>	14
14	Création de la partition 3 à l'aide de fdisk	16
15	Création de la partition 4 à l'aide de fdisk	17
16	Installation de btrfs et f2fs	18
17	Formatage de la partition 3	19
18	Formatage de la partition 4	19
19	Histogramme des temps d'exécution du programme en fonction des systèmes de fichier	22
20	Copie d'un fichier dans la partition LUKS	25
21	Ajout d'une seconde passphrase	25
22	Copie de la partition LUKS avec la commande <i>dd</i>	26
23	Master Key 1	27
24	Master Key 2	28
25	Démarrage du NanoPi avec la partition cryptée	28
26	Log de l'exécution du script <i>init_ramfs.sh</i>	33
27	Contenu du dossier <i>ramfs</i>	34
28	Illustration des fichiers créés par le script <i>init_ramfs.sh</i>	34
29	Illustration de la console de initramfs	37
30	Contrôle du contenu de usrfs (partition 3)	37
31	<i>Ramfs</i> avec l'ajout des bibliothèques nécessaires à <i>cryptsetup</i>	38
32	Vérification de l'installation de swtpm	46
33	Hierarchie parent enfant des clés dans le TPM	49
34	Schéma du cryptage et du décryptage du fichier	52
35	Schéma bloc de la création d'une PCR policy	54
36	Schéma bloc de l'utilisation d'un PCR policy	54