Solution du challenge SSTIC 2019

Théo COMBE

22 mai 2019

Introduction

Le challenge 2019 consiste à analyser l'image mémoire du téléphone d'un individu tentant de nuire à la communauté de la sécurité informatique française, afin d'en extraire des preuves permettant de l'inculper. En effet, quelle occasion plus appropriée que le SSTIC pour empoisonner des experts en sécurité informatique ? Le challenge se compose de 4 niveaux, la résolution de chacun d'entre eux permettant d'obtenir une clé qui sert à déchiffrer l'étape suivante. A l'exception du premier niveau, l'enchaînement est linéaire, c'est-à-dire que chaque niveau est protégé par (au plus) une seule couche de chiffrement (par opposition à une organisation en oignon).

Ce document vise à présenter la démarche suivie par l'auteur lors de la résolution du challenge, il s'agit d'une méthode parmi d'autres, pas forcément la plus directe car prenant en compte les errements. Il s'organise en 5 parties, chacune correspondant à un niveau du challenge : tout d'abord une *simple power analysis (SPA)* permettant de déchiffrer l'image disque du téléphone incriminé, puis la rétro-ingéniérie d'un *secure element* matériel maison, suivis de deux épreuves de rétro-ingéniérie logicielle se présentant sous la même forme : un fichier exécutable nommé `decrypted_binary` prenant le flag en argument, et fournissant l'information `bon flag` ou `mauvais flag` sur sa sortie standard. La dernière partie est toute simple : il suffit de grep `@challenge.sstic.org` dans un fichier.

Le code produit pour la réalisation du challenge est disponible sur le dépôt : `https://github.com/tdrrdt/sstic2019/`.

Table des matières

1 Acquisition des données

La mission est décrite comme suit :

```
Bonjour,

Récemment un individu au comportement suspect nous a été signalé. Il semblerait qu'il
 s'attaque à la communauté sécurité informatique française avec notamment l'intention
 de lui nuire.

Sans preuve, il est difficile d'agir à son encontre. Ainsi, nous avons décidé de
saisir son téléphone portable afin de collecter des éléments confirmant nos hypothè
ses. Cependant son téléphone semble posséder plusieurs couches de chiffrement qui
nous empêchent d'accéder à ses données.

Dans l'incapacité de contourner ces systèmes de chiffrement, nous avons décidé de
faire appel à vous pour nous aider. Nous avons consacré du temps à rendre possible le
 démarrage du téléphone sécurisé dans un environnement virtualisé. Malheureusement le
 coffre de clef du téléphone ciblé n'a pas pu être copié. Avant de devoir restituer
le téléphone, nous avons été en mesure d'enregistrer une trace de consommation de
courant lors du démarrage du téléphone. Nous espérons que cela pourra vous être utile
.

Des instructions techniques plus précises vous seront fournies.

Bonne chance pour votre mission et nous comptons sur vous pour nous communiquer
toutes les preuves que vous pourrez trouver au cours de votre investigation à l'
adresse mail suivante : challenge2019@sstic.org.

La communauté sécurité informatique française dépend de vous !
```

Après avoir récupéré l'archive du challenge et vérifié son empreinte `sha256`, on examine son contenu :

```
[user@machine SSTIC2019]$ sha256sum challenge_SSTIC_2019-virtual_phone.tar.gz
4dcd7f3c59ca7fd3b4ef6f0be3055173b87ea3ec6dd9b0e37eb697ecca57ba1d
challenge_SSTIC_2019-virtual_phone.tar.gz
```

Le hash est correct, on peut extraire l'archive :

```
[user@machine SSTIC2019]$ tar xf challenge_SSTIC_2019-virtual_phone.tar.gz
[user@machine SSTIC2019]$ ls -l
total 44100
-rw-r--r-- 1 user user 45152817 May 20 21:05 challenge_SSTIC_2019-virtual_phone.tar.
gz
drwxr-xr-x 2 user user     4096 Mar 27 12:43 virtual_phone
[user@machine SSTIC2019]$ cd virtual_phone/
[user@machine SSTIC2019]$ ls -l
total 49468
-rw-r--r-- 1 user user 35879224 Mar 28 23:24 flash.bin
-rw-r--r-- 1 user user 14737268 Mar 28 23:24 power_consumption.npz
-rw-r--r-- 1 user user      622 Mar 28 23:24 README
-rw-r--r-- 1 user user    30897 Mar 28 23:24 rom.bin
```

Le fichier `README` nous indique qu'il s'agit d'une VM utilisant l'architecture `aarch64`, et comment la lancer. La version 3.1.0 de qemu est disponible immédiatement sous Archlinux : `pacman -S qemu-arch-extra`. Attention toutefois, la version était bonne au début mais a changé pendant la durée du challenge et il a fallu la verrouiller. Pour commencer on suit les instructions à la lettre, en activant directement la redirection de ports :

```
[user@machine virtual_phone]$ qemu-system-aarch64 -nographic -machine virt,secure=on
-cpu max -smp 1 -m 1024 -bios rom.bin -semihosting-config enable,target=native -
device loader,file=./flash.bin,addr=0x04000000 -netdev user,id=network0,hostfwd=tcp
:127.0.0.1:5555-192.168.200.200:22 -net nic,netdev=network0
########################################
#      virtual environment detected      #
#           QEMU 3.1+ is needed          #
########################################
NOTICE:  Booting SSTIC ARM Trusted Firmware
```

```
KEYSTORE: keystore doesn't exist
ERROR:   KEYSTORE: Can't read keystore, reset keystore, try to boot again
```

La VM ne trouve pas son coffre de clés, le crée, et s'arrête.

```
[user@machine virtual_phone]$ ls -l
total 49472
-rw-r--r-- 1 user user 35879224 Mar 28 23:24 flash.bin
-rw-r--r-- 1 user user     1360 May 20 21:23 keystore
-rw-r--r-- 1 user user 14737268 Mar 28 23:24 power_consumption.npz
-rw-r--r-- 1 user user      622 Mar 28 23:24 README
-rw-r--r-- 1 user user    30897 Mar 28 23:24 rom.bin
```

Au deuxième lancement, ça fonctionne mieux :

```
[user@machine virtual_phone]$ qemu-system-aarch64 -nographic -machine virt,secure=on
-cpu max -smp 1 -m 1024 -bios rom.bin -semihosting-config enable,target=native -
device loader,file=./flash.bin,addr=0x04000000 -netdev user,id=network0,hostfwd=tcp
:127.0.0.1:5555-192.168.200.200:22 -net nic,netdev=network0
########################################
#      virtual environment detected      #
#           QEMU 3.1+ is needed          #
########################################
NOTICE:  Booting SSTIC ARM Trusted Firmware
KEYSTORE: AES Key is still encrypted, need decryption
KEYSTORE: Need RSA key to decrypt
KEYSTORE: RSA private exponent is not set, please set it in the keystore or enter hex
 value :
```

2 Une clé pas si secrète

La machine demande une clé RSA pour continuer la séquence de démarrage, probablement pour en dériver une clé symétrique qui servira pour le déchiffrement effectif. S'agissant d'un déchiffrement, c'est probablement l'exposant privé `d` qui est demandé. Si on appuie sur Entrée sans avoir tapé de clé, la machine nous informe qu'elle attend une clé de longueur 0x100, soit 256 octets. Selon que cela prend en compte ou non l'encodage en hexadécimal attendu, cela correspond à une clé RSA de 1024 ou 2048 bits. On retente avec une en entrée `00` (hexadécimal pour un octet) :

```
00
KEYSTORE: Key len should be 0x100 instead of 2
```

Il s'agit donc de 0x100 caractères hexadécimaux, ce qui correspond à une clé de 128 octets, soit 1024 bits. C'est peu mais hors de portée de factorisation.

Le dernier fichier non utilisé est le fichier `power_consumption.npz`, correspondant d'après l'énoncé à la trace de consommation du téléphone lors du dernier démarrage réussi. N'ayant qu'une trace disponible, cela fait penser à une analyse de consommation simple (SPA).

```
[user@machine virtual_phone]$ file power_consumption.npz
power_consumption.npz: Zip archive data, at least v2.0 to extract
[user@machine virtual_phone]$ unzip power_consumption.npz
Archive:  power_consumption.npz
 extracting: arr_0.npy
[user@machine virtual_phone]$ file arr_0.npz
arr_0.npy: data
```

Ce fichier est une archive zip, après extraction on obtient le fichier `arr_0.npy`, qui est un fichier de données numpy. S'il s'agit d'une trace de consommation, la première idée qui vient à l'esprit est de l'afficher graphiquement.

2.1 Analyse visuelle

Affichage de la trace :

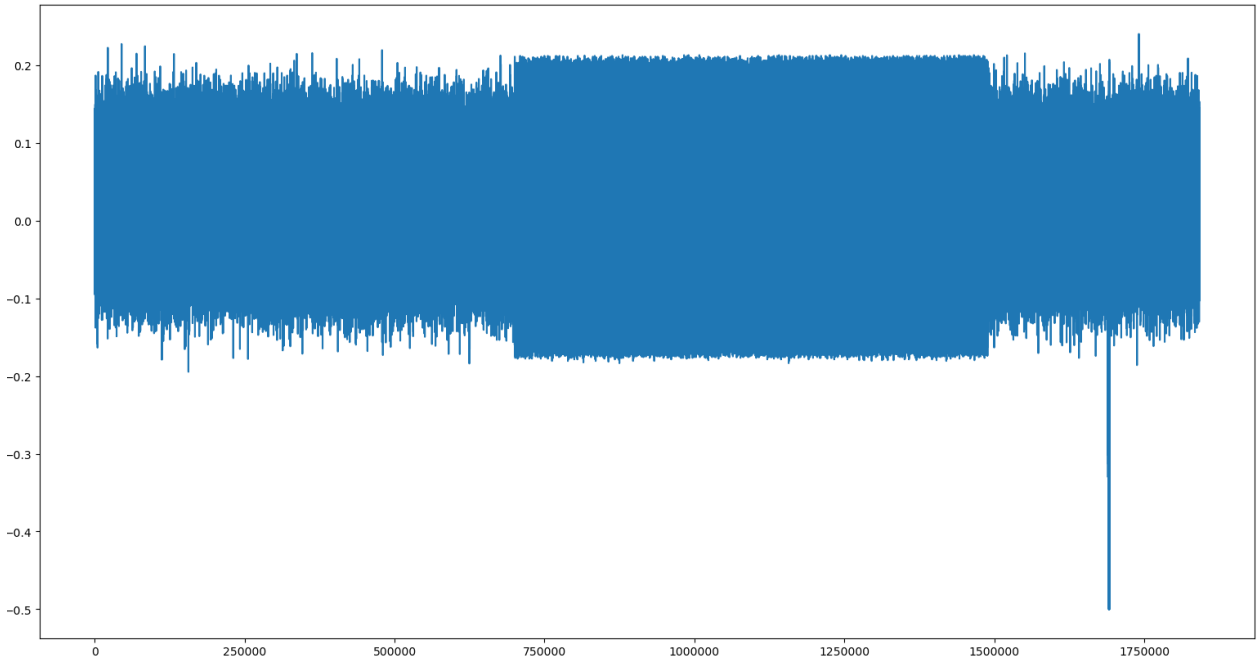

FIGURE 1 – Trace complète

Code 1 – `plot_trace.py`

```
#!/usr/bin/python3

import numpy as np
from matplotlib import pyplot as plt

arr = np.load('arr_0.npy')
print(len(arr))
plt.plot(arr)
plt.show()
```

Ce code affiche la Figure 1. On observe 3 zones distinctes : la première et la troisième semblent assez erratiques, tandis que la deuxième est très régulière. On tente d'agrandir cette deuxième zone (Figure **??**). On agrandit encore (Figure **??**). Cette fois-ci on différencie clairement un motif qui se répète avec une petite variation.

2.2 Digression : fonctionnement d'une SPA

Une SPA (*Simple Power Analysis*) est une catégorie d'attaques par canaux auxiliaires qui repose sur le fait que la consommation électrique d'un appareil vulnérable effectuant une opération cryptographique dépend directement des secrets manipulés par cet appareil. Dans le cas d'un déchiffrement RSA, il s'agit des bits de l'exposant privé `d`. Les opérations coûteuses en temps (donc visibles sur la trace) sont les multiplications et les divisions. Dans le cas d'une implémentation non protégée (ce qui est l'hypothèse faite ici, on ne sait pas si cela fonctionnera),

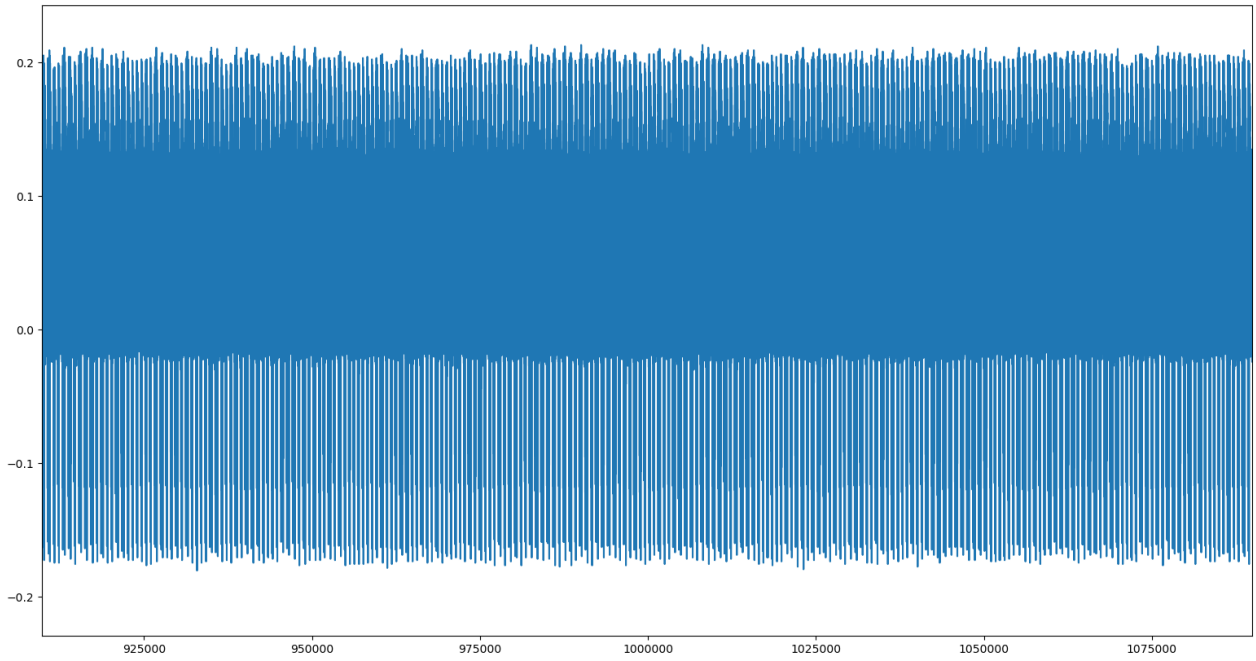

FIGURE 2 – Agrandissemnt de la zone du milieu

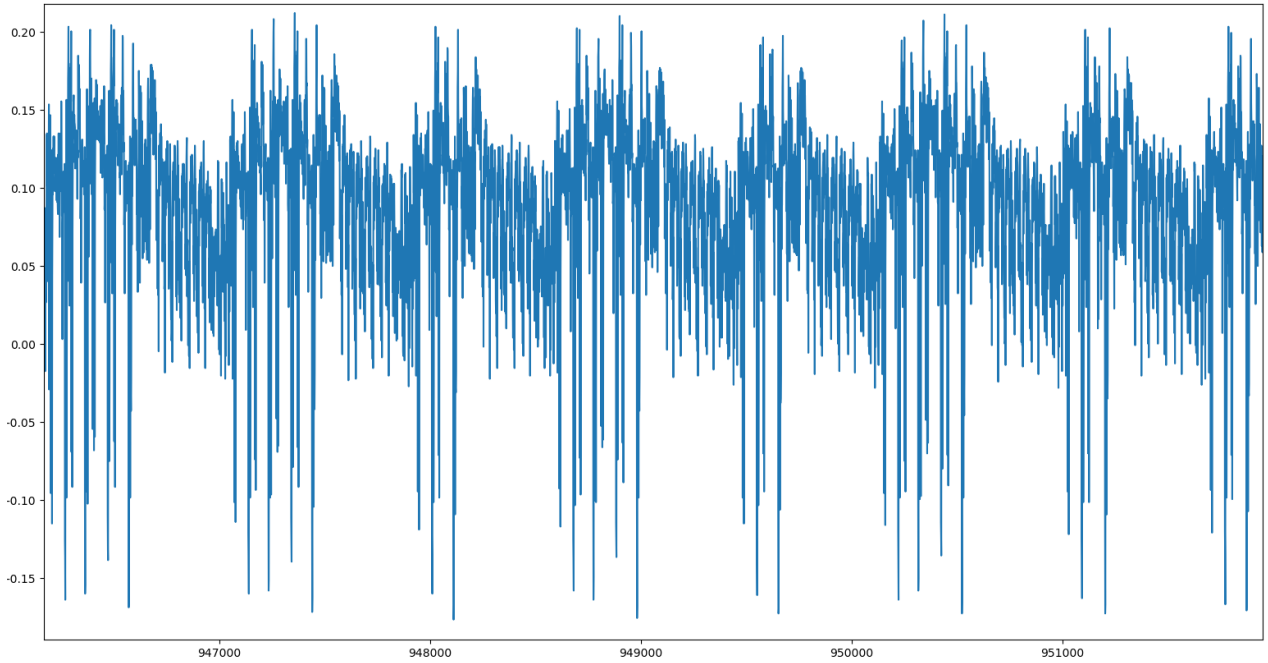

FIGURE 3 – Encore plus agrandi

l'algorithme d'exponentiation modulaire naïf est le suivant :

Data: x : entier, d : exposant, n : module
Result: $x^d \ mod \ n$
$r \leftarrow 1$;
while $d \neq 0$ **do**
  if $d \ mod \ 2 \neq 0$ **then**
    $r \leftarrow r * x \ mod \ n$;
  end
  $x \leftarrow x * x \ mod \ n$;
  $d \leftarrow d/2$;
end
return r

Algorithm 1: Exponentiation rapide modulaire

2.3 Décodage de la clé

Le flot d'exécution, donc la durée de l'exécution et donc la consommation dépendent ainsi directement de l'exposant privé, dont les bits sont manipulés un à un, successivement. On observe que quand le bit courant de `d` vaut 0 le programme effectue une multiplication et une division (le modulo) (la division de `d` par 2 est juste un décalage de bits), tandis que quand le bit courant de `d` vaut 1 le programme effectue 2 multiplications et 2 divisions.

La Figure 3 présente 2 motifs différents : un avec 3 grands pics en bas, et un avec 6 grands pics en bas. Il s'agit probablement des traces de l'exécution de l'algorithme d'exponentiation quand il passe respectivement sur un 0 et un 1 dans `d`. A partir de là, on implémente un programme en Python qui parcourt la trace en cherchant un début de pic (par exemple un échantillon inférieur à -0.05 après 50 échantillons avec une valeur supérieure à -0.05 caractérise un début de pic). Ensuite, on avance et tant qu'on n'a pas parcouru un nombre d'échantillons non négatifs consécutifs supérieur à 100 (qui caractérise la séparation entre 2 motifs), on compte le nombre d'échantillons inférieurs à -0.05 qu'on rencontre, avec un pas de 50 (un pas plus petit est inutile, car c'est approximativement la largeur d'un pic).

Quand on atteint 100 échantillons consécutifs non négatifs, on considère qu'on est passé au motif suivant, et on stocke le nombre de paquets de 50 échantillons passés sous le seuil qu'on a trouvé.

Après avoir parcouru toute la trace, on applique à nouveau un seuil (15) aux nombres ainsi obtenus : selon que ces nombres sont d'un côté ou de l'autre du seuil, le motif correspond à un 0 ou un 1 dans la clé. Il ne reste plus qu'à remettre les bits dans l'ordre (l'algorithme commence par utiliser les bits de poids faible en premier, alors que la VM nous demande les bits de poids fort en premier), puis de les afficher.

Remarque : Tous les seuils sont choisis de manière empirique, en s'aidant de la représentation visuelle de la trace.

On obtient le script suivant :

Code 2 – `decode_key.py`

```
#!/usr/bin/python3

import numpy as np

arr = np.load('arr_0.npy')
print("Number of samples :", len(arr))

# Keep only relevant samples in trace (middle of trace)
arr_2 = arr[700040:1491600]

index = 0
neg_size = 50

# count of samples below threshold in [index:index+50]
def get_negative_count(index):
    cpt = 0
    for i in range(neg_size):
        if arr_2[index+i] < -0.05:
            cpt += 1
    return cpt

# next index with at least 3 negative samples in the next 50
# ignore if less values, components power consumption is not an exact science
def get_next_negative_index():
    global index
    while index + neg_size < len(arr_2):
        if get_negative_count(index) > 3:
            index += 10
            return index
        index += 1
```

```
    return -1

last_block_index = 0
current_byte_count = 0
key_bytes = []
while get_next_negative_index() != -1:
    if index - last_block_index < 100:
        current_byte_count += 1
    else:
        # long time since no peak, it's a new block
        key_bytes.append(current_byte_count)
        current_byte_count = 0
    last_block_index = index

key_bytes[0] = 20
key_bytes = key_bytes[0:1024]

key_bytes.reverse()

# 15 is a good threshold
bits = ""
for b in key_bytes:
    bits += "1" if b > 15 else "0"

print("key =", hex(int(bits, base=2))[2:])
```

```
[user@machine virtual_phone]$ python decode_key.py
Number of samples : 1842128
key =
23d87cdf97bb95abe6273c384190c765f552ab86f6de30a8db74435c95e6e313
8f54af689812d8f9359cf0f4d453a0c11ec68ce470216c09e74c8947adaf23e9
02415d61ddf2c0ffe459cbb40f7de42bdb7cd14093100a570e8c29819765e2d8
d276f86471b52ac29aa2ce2bb72cd45006279e82bec253ae9675fe45824f6001
```

En donnant cette clé à la VM, celle-ci affiche une animation montrant un à un les bits de la clé utilisés dans l'exponentiation, puis elle affiche le flag :

```
SSTIC{a947d6980ccf7b87cb8d7c246}
```

et enfin démarre complètement.

Remarque : On a fourni la clé privée à la VM, celle-ci embarque nécessairement un chiffré ainsi que le module N quelque part. Nous le verrons plus tard.

3 Une drôle de crypto

3.1 Tour du propriétaire

On commence par explorer la machine virtuelle.

```
# uname -a
Linux sstic 5.0.3 #37 SMP PREEMPT Mon Mar 25 10:26:28 CET 2019 aarch64 GNU/Linux

# lsmod
Module                  Size  Used by    Not tainted
sstic                  16384  0

# ls -l /root
total 728
-rwxr-xr-x    1 root     root          7734 Mar 28 22:24 get_safe1_key.py
drwxr-xr-x    2 root     root            60 Mar 28 22:24 safe_01
drwxr-xr-x    2 root     root            60 Mar 28 22:24 safe_02
drwxr-xr-x    2 root     root            60 Mar 28 22:24 safe_03
-rw-r--r--    1 root     root        734017 Mar 28 22:24 schematics.png
drwxr-xr-x    2 root     root           120 May 20 22:33 tools

# ls -la /root/safe_*
/root/safe_01:
total 84
drwxr-xr-x    2 root     root            60 Mar 28 22:24 .
drwx------    6 root     root           160 Mar 28 22:24 ..
-rw-r--r--    1 root     root         84272 Mar 28 22:24 .encrypted

/root/safe_02:
total 1500
drwxr-xr-x    2 root     root            60 Mar 28 22:24 .
drwx------    6 root     root           160 Mar 28 22:24 ..
-rw-r--r--    1 root     root       1532448 Mar 28 22:24 .encrypted

/root/safe_03:
total 968
drwxr-xr-x    2 root     root            60 Mar 28 22:24 .
drwx------    6 root     root           160 Mar 28 22:24 ..
-rw-r--r--    1 root     root        989840 Mar 28 22:24 .encrypted
```

Il s'agit d'un Linux récent (`5.0.3`) sur `aarch64` ayant le module `sstic` chargé. Tous les fichiers intéressants semblent être dans le dossier `/root`, qui contient 3 dossiers `safe_0{1,2,3}`, contenant chacun un fichier `.encrypted`, qui contient une épreuve. Le dossier `/root/tools` contient des utilitaires, dont le script `add_key.py` qui sert à ajouter une clé au keystore de la machine, en lui passant un flag en argument.

L'épreuve suivante porte donc probablement sur les deux fichiers restants : `get_safe1_key.py` et `schematics.png`, leur nom aidant un peu.

3.2 Emulation de composant sécurisé

Le fichier `get_safe1_key.py` dérive la clé de `safe_01` à partir d'appels à un élément de sécurité matériel (*secure element*). Pour cela, après avoir testé le bon fonctionnement du secure element à l'initialisation, il effectue 8 étapes de vérifications, durant lesquelles il demande à l'utilisateur d'appuyer sur une combinaison des 4 boutons du secure element puis d'appuyer sur Entrée. Il effectue ensuite un certain nombre d'appels au secure element avec des paramètres différents à chaque étape de vérification, produisant une sortie finale sur 8 bits. Les sorties de chacune des 8 étapes sont ensuite concaténées pour former une clé intermédiaire de 64 bits. Son hash sha256 est comparé à une constante, et si la clé est bonne, elle est utilisée pour dériver la vraie clé du coffre 1.

Malheureusement pour nous, nous n'avons pas le secure element en question, ni un moyen pour le script de communiquer avec lui. Et malheureusement pour les auteurs du script, ils ont oublié de supprimer les documents de conception. Le fichier `schematics.png` (Figure 4)

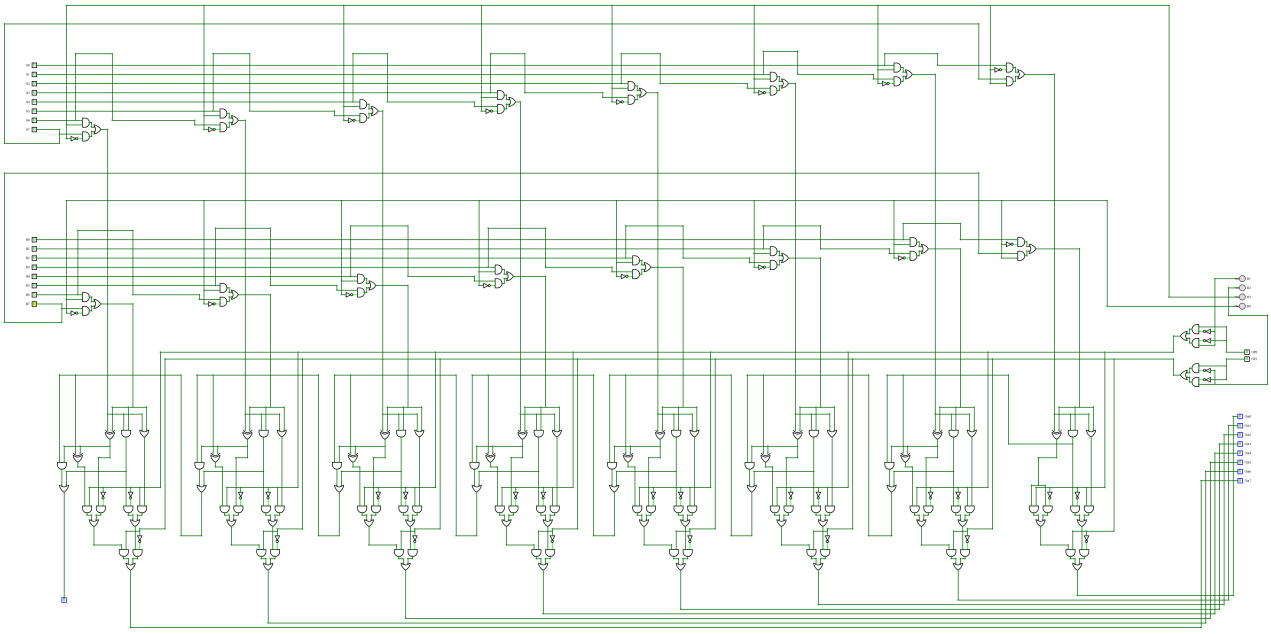

FIGURE 4 – Schéma électrique de l'élément sécurisé

contient en effet un schéma de circuit électronique, dont les entrées et sorties sont nommées comme les entrées et sorties du secure element de la fonction `secure_device()` qui l'appelle.

Partant de là, il suffit de réimplémenter logiciellement ce circuit, pour obtenir la fonction `secure_device()` manquante. Il manque cependant toujours une information : l'entrée utilisateur composée des 4 boutons (ça ne doit vraiment pas être pratique à utiliser...). Il y a $2^4 = 16$ combinaisons possibles, et chacune des 8 étapes de validation attend une combinaison différente. Il y a donc en tout $16^8 = 2^{32}$ combinaisons différentes, ce qui est à la portée d'une attaque par force brute.

Pour des raisons de performance de l'attaque, on choisit d'implémenter le secure element en C, le code des étapes de validation (`step*`) s'adaptant rapidement de Python à C, dans le fichier `code/stage2/secure_element.c`

On compile ce code sans oublier l'option `-O3` et l'édition de liens avec OpenSSL (`-lcrypto`), et au bout d'environ une heure d'exécution :

```
Found key : 8fa4dfa9d4edbbf0 (combination = 1684743006, hash = 00
c8bb35d44dcbb2712a11799d8e1316045d64404f337f4ff653c27607f436ea)
```

Le hash correspond bien, il reste à utiliser cette clé pour dériver la clé AES avec ce petit script :

Code 3 – `decrypt_aes_key.py`

```
1  #!/usr/bin/python3
2
3  import hashlib
4  import sys
5  import binascii
6
7  def info(msg):
8          print("[\033[34;1mi\033[0m] %s" % (msg))
9
10 key = bytearray(binascii.unhexlify(sys.argv[1]))
11 info("Dérivation de la clef AES safe_01")
12 aes_key = hashlib.scrypt(key,salt =b"sup3r_s3cur3_k3y_d3r1v4t10n_s4lt",n=1<<0xd,r
   =1<<3,p=1<<1,dklen=32)
13 info("aes key : %s" % aes_key.hex())
```

On obtient :

```
[user@machine safe_1]$ python decrypt_aes_key.py 8fa4dfa9d4edbbf0
[i] Dérivation de la clef AES safe_01
[i] aes key : 5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a
```

En appelant le script `/root/tools/add_key.py` dans la machine virtuelle avec cette clé en paramètre, on obtient le flag, qui est automatiquement ajouté au keystore de la machine :

```
SSTIC{5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a}
```

A ce stade, je validais la deuxième étape en première position, et pensais alors finir le challenge assez rapidement. La suite des événements m'a donné tort.

4 La Moria

4.1 Premières pistes

Le fichier obtenu est un binaire exécutable dans la machine virtuelle :

```
# cd /root/safe_01/
# ls -l
total 84
-rwxr-xr-x    1 root     root         84256 May 21 00:32 decrypted_file
# file decrypted_file
decrypted_file: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynamically
 linked, interpreter /lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0, BuildID[sha1]=5
b5be1337d13c986d0e21441d771a36e41a34d17, stripped
```

Les sections :

```
# readelf -S decrypted_file
There are 29 section headers, starting at offset 0x141d8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         00000000004001c8  000001c8
       000000000000001b  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             00000000004001e4  000001e4
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             0000000000400204  00000204
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000400228  00000228
       0000000000002420  0000000000000000  WA       5     0     8
  [ 5] .dynsym           DYNSYM           0000000000402648  00002648
       00000000000001e0  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000402828  00002828
       000000000000017b  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           00000000004029a4  000029a4
       0000000000000028  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          00000000004029d0  000029d0
       0000000000000070  0000000000000000   A       6     3     8
  [ 9] .rela.dyn         RELA             0000000000402a40  00002a40
       0000000000000030  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             0000000000402a70  00002a70
       0000000000000180  0000000000000018  AI       5    24     8
  [11] .init             PROGBITS         0000000000402bf0  00002bf0
       0000000000000014  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         0000000000402c10  00002c10
       0000000000000120  0000000000000010  AX       0     0     16
  [13] .text             PROGBITS         0000000000402d30  00002d30
       0000000000000334  0000000000000000  AX       0     0     8
  [14] .fini             PROGBITS         0000000000403064  00003064
       0000000000000010  0000000000000000  AX       0     0     4
  [15] .rodata           PROGBITS         0000000000403078  00003078
       000000000000008d  0000000000000000   A       0     0     8
  [16] .eh_frame_hdr     PROGBITS         0000000000403108  00003108
       000000000000005c  0000000000000000   A       0     0     4
  [17] .eh_frame         PROGBITS         0000000000403168  00003168
       0000000000000160  0000000000000000   A       0     0     64
  [18] .gcc_except_table PROGBITS         00000000004032cc  000032cc
       0000000000000024  0000000000000000   A       0     0     4
  [19] .init_array       INIT_ARRAY       0000000000413da0  00013da0
       0000000000000010  0000000000000008  WA       0     0     8
  [20] .fini_array       FINI_ARRAY       0000000000413db0  00013db0
       0000000000000008  0000000000000008  WA       0     0     8
  [21] .data.rel.ro      PROGBITS         0000000000413db8  00013db8
       0000000000000020  0000000000000000  WA       0     0     8
  [22] .dynamic          DYNAMIC          0000000000413dd8  00013dd8
       0000000000000200  0000000000000010  WA       6     0     8
  [23] .got              PROGBITS         0000000000413fd8  00013fd8
       0000000000000010  0000000000000008  WA       0     0     8
  [24] .got.plt          PROGBITS         0000000000413fe8  00013fe8
       0000000000000098  0000000000000008  WA       0     0     8
  [25] .data             PROGBITS         0000000000414080  00014080
       0000000000000030  0000000000000000  WA       0     0     8
  [26] .bss              NOBITS           00000000004140b0  000140b0
       0000000000000018  0000000000000000  WA       0     0     8
```

```
LOAD:0000000000400228 ; ELF GNU Hash Table
LOAD:0000000000400228 elf_gnu_hash_nbuckets DCD 3
LOAD:000000000040022C elf_gnu_hash_symbias DCD 0x11
LOAD:0000000000400230 elf_gnu_hash_bitmask_nwords DCD 1
LOAD:0000000000400234 elf_gnu_hash_shift DCD 6
LOAD:0000000000400238 elf_gnu_hash_indexes DCQ 0x100412008100000
LOAD:0000000000400240 elf_gnu_hash_bucket DCD 0, 0, 0x11
LOAD:000000000040024C elf_gnu_hash_chain DCD 0xB66B4978, 0xEC0506EE, 0x4CD54529, 0x22A8086F, 0x22080806
LOAD:000000000040024C                    DCD 0x16061206, 0x12220808, 0x8081606, 0x16061222, 0x12220808
LOAD:000000000040024C                    DCD 0x8081606, 0x28019422, 0x3150044, 0x492F0315, 0x1C240E00
LOAD:000000000040024C                    DCD 0x2A68E8E, 0x16276583, 0x2E53470E, 0x7564F161, 0x162227DC
LOAD:000000000040024C                    DCD 0x6EC6130E, 0xC69B74A8, 0x162227D9, 0x6AAED50E, 0x850B36E7
LOAD:000000000040024C                    DCD 0x28222765, 0x980E000C, 0x4030, 0x2F000000, 0xB80E7FFF
LOAD:000000000040024C                    DCD 0x4030, 0x2F000000, 0x17307FFF, 0x15051530, 0x332F05
LOAD:000000000040024C                    DCD 0x4150415, 0x1700CA2F, 0x17171317, 0x2153113, 0x1E2F0215
LOAD:000000000040024C                    DCD 0x15071500, 0xB52F07, 0x3151717, 0x31051516, 0x1C341222
LOAD:000000000040024C                    DCD 0x13FFCC28, 0x3150315, 0x34FF7A2F, 0xFF0E1217, 0xFFFFFF
```

FIGURE 5 – Données dans la section `.gnu.hash`

```
  [27] .comment          PROGBITS         0000000000000000  000140b0
       0000000000000011  0000000000000001  MS       0     0     1
  [28] .shstrtab         STRTAB           0000000000000000  000140c1
       0000000000000112  0000000000000000           0     0     1
```

On tente de l'exécuter :

```
# ./decrypted_file
Usage : ./decrypted_file <flag>
# ./decrypted_file monflag
Not good
```

Il s'agit donc d'un oracle disant si le flag entré est bon ou mauvais. En regardant les chaînes de caractères présentes dans le binaire, celui-ci semble être un programme C++ (il importe entre autres `libstdc++.so.6`), ce qui est confirmé en l'ouvrant avec IDA : les symboles `__cxa_begin_catch`, `__cxa_atexit`, `__cxa_throw` et d'autres présents correspondent à des symboles de la bibliothèque standard C++.

Il y a finalement assez peu de code dans ce binaire, un rapide parcours avec la vue hexadécimale d'IDA montre cependant un gros bloc de données dans la section `.gnu.hash` (Figure 5), qui sert à la résolution dynamique des symboles.

Le nom des fonctions étant absent, il convient de les retrouver. Pour cela on retrouve `main` en recherchant dans le code les références à la chaîne de caractère `"Usage :"`, qui est utilisée dans le parseur d'arguments.

La fonction `main` est vue comme 2 morceaux par IDA (Figure 6).

En remontant de proche en proche, on reconstitue tout le code C++ : la fonction `main` vérifie qu'il y a bien un argument, et appelle une fonction (appelée `throw_flag_exception`) dans une clause `try / catch`. Cette fonction prend en argument un pointeur sur le flag qu'on a fourni au programme, et se contente de lancer une exception avec `throw` (mot-clé traduit en `.__cxa_allocate_exception` et `.__cxa_throw`. par le compilateur). Cette dernière fonction est appelée avec un pointeur vers le flag en argument.

Le programme C++ original semble ainsi se réécrire :

Code 4 – `cpp_program.cpp`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  void throw_flag_exception(const char* flag)
6  {
7          throw flag;
8  }
9
10 int main(int argc, char* argv[])
11 {
12         if (argc != 2)
13         {
14                 printf("Usage : %s <flag>\n", argv[0]);
```

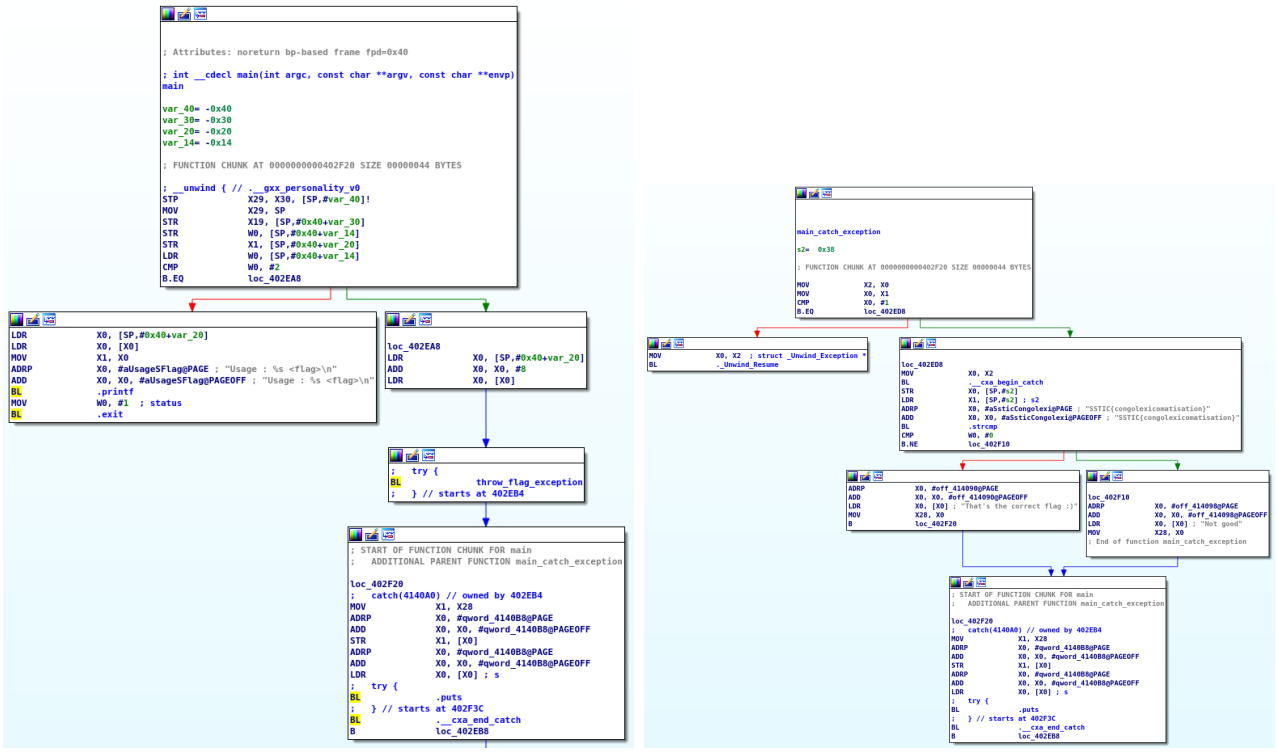

FIGURE 6 – Vue IDA de la fonction `main()`

```
15                      exit(1);
16              }
17              try
18              {
19                      throw_flag_exception(argv[1]);
20              }
21              catch (const char* msg)
22              {
23                      if (strcmp(msg, "SSTIC{congolexicomatisation}"))
24                      {
25                              puts("Not good");
26                      }
27                      else
28                      {
29                              puts("That's the correct flag :)");
30                      }
31              }
32              return 0;
33      }
```

Evidemment, ce n'est pas le bon flag... Il se passe donc quelque chose entre le `throw` et le `catch` qui modifie le flag. Par ailleurs, en posant avec `gdb` un breakpoint à l'adresse `0x402ef0` (adresse de l'appel à `strcmp()`), on s'aperçoit que ce code n'est en fait jamais exécuté.

4.2 Un nain bien caché

Après quelques recherches sur les mécanismes de gestion des exceptions en C++, on s'oriente vers les sections `.eh_frame` [1], `.eh_frame_hdr` [2] et `.gcc_except_table` qui contiennent les informations relatives au comportement du programme lorsqu'une exception est levée. En effet dans ce cas, l'exécution normale du programme est interrompue, et la fonction courante ne peut pas rendre la main correctement. Son adresse de retour est donc perdue, et il est impossible de déterminer dans quel état elle a laissé la pile. La section `.eh_frame_hdr` contient, pour chaque adresse (en fait chaque plage d'adresses) où une exception peut survenir, la liste des actions à

1. `https://refspecs.linuxfoundation.org/LSB_3.0.0/LSB-PDA/LSB-PDA/ehframechpt.html`
2. `https://refspecs.linuxfoundation.org/LSB_1.3.0/gLSB/gLSB/ehframehdr.html`

effectuer pour libérer correctement la `stack frame` : appel des destructeurs des objets locaux, restauration des registres *callee-saved*, recherche du début de la `stack frame` et retour à la fonction qui avait appelé la fonction ayant levé l'exception. Si cet appel avait lieu dans une clause `try / catch`, l'exception est rattrapée et l'exécution continue au début du `catch`. Sinon, la `stack frame` de la fonction appelante est elle aussi libérée, et on retourne à la fonction qui l'avait appelée, et ainsi de suite. Si on arrive en haut de la pile sans avoir rencontré de clause `catch`, le programme plante avec une erreur (`uncaught exception`).

Cette remontée de la pile est effectuée par une fonction appelée `personality routine`, dont la sémantique dépend de l'ABI du langage. Dans notre cas, il s'agit de `__gxx_personality_v0`. Les actions à effectuer à chaque étape sont donc lues depuis la section `.eh_frame`, qui est remplie par le compilateur. D'après la documentation, cette section contient plusieurs entrées `CIE`(*Common Information Entry*) suivies chacune de 0 ou plus `FDE`(*Frame Description Entry*). Ces entrées contiennent des informations sur les valeurs de `PC / eip` au moment où l'exception est levée, des numéros de version, des flags, et un champ `"Initial instructions"`, contenant les instructions nécessaires pour effectuer les actions décrites au paragraphe précédent. La section `.eh_frame_hdr` contient un mécanisme de lookup pour retrouver rapidement la bonne entrée CIE dans `.eh_frame` à partir de `PC / eip`.

Après avoir essayé plusieurs outils (et même avoir lu la section `.eh_frame` à la main, octet par octet !), il apparaît que Ghidra sait lire cette section (Figure 7).

4.3 Passage en dynamique

N'ayant toujours pas entendu parler de Dwarf à ce stade, les 3 instructions à la fin de chaque CIE nous apportent peu d'information, et on cherche toujours le code qui transforme le flag. On décide alors de laisser le programme nous amener jusqu'au code intéressant. Pour cela, on commence par lister toutes les bibliothèques importées, à l'aide de `strace`. Les seuls fichiers ouverts avec succès sont :

- `/usr/lib64/libstdc++.so.6`
- `/lib64/libm.so.6`
- `/lib64/libgcc_s.so.1`
- `/lib64/libc.so.6`

On ouvre ces fichiers avec IDA, afin de s'assurer qu'ils ne contiennent pas de code caché. Ce travail étant en réalité titanesque (il y a la libc et la libstdc++), on se contente de rechercher des chaînes de caractères suspectes et de rechercher les empreintes cryptographiques des fichiers sur Internet. On ne trouve pour l'instant rien de suspect dans ces bibliothèques.

L'étape suivante consiste à exécuter pas à pas le programme avec `gdb` à partir de l'appel à la fonction `.__cxa_throw`, dernière adresse connue exécutée avant le code caché. Pour cette méthode et pour toute la suite de l'étape, on désactive l'ASLR dans la VM, afin d'obtenir des résultats reproductibles lors des appels aux bibliothèques chargées dynamiquement :

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

Curieusement, on n'obtient rien non plus avec cette méthode : la trace générée passe par un certain nombre de fonctions de la libc et de la libstdc++, qui ne font rien d'anormal.

On décide alors de piéger le programme : on pose un watchpoint à l'endroit où est écrit le flag en mémoire : sur `argv[1]`. Cette adresse est chargée dans `X0` à l'adresse `0x402eb0`, on place donc tout d'abord un breakpoint juste après, à `0x402eb4`, puis on place un watchpoint (en lecture) sur `*X0`.

```
(gdb) b *0x402eb4
Breakpoint 1 at 0x402eb4
```

```
                    //
                    // .eh_frame
                    // SHT_PROGBITS  [0x403168 - 0x4032c7]
                    // ram: 00403168-004032c7
                    //

                    *************************************************************
                    * Common Information Entry                                  *
                    *************************************************************
                    cie_00403168                                    XREF[11]:    0040310c(*), 0040319c(*),
                                                                                 004031b0(*), 004031c4(*),
                                                                                 004031e8(*), 004031fc(*),
                                                                                 00403244(*), 00403264(*),
                                                                                 00403284(*), 004032b8(*),
                                                                                 _elfSectionHeaders::00000450(*)
00403168 10 00 00 00     ddw        10h                                          (CIE) Length
0040316c 00 00 00 00     ddw        0h                                           (CIE) ID
00403170 01              db         1h                                           (CIE) Version
00403171 7a 52 00        ds         "zR"                                         (CIE) Augmentation String
00403174 01              uleb128    1h                                           (CIE) Code Alignment
00403175 78              sleb128    -8h                                          (CIE) Data Alignment
00403176 1e              db         1Eh                                          (CIE) Return Address Register Co...
00403177 01              uleb128    1h                                           (CIE) Augmentation Data Length
00403178 1b              dwfenc     DW_EH_PE_sdata4 | DW_EH_PE_pcrel             (CIE Augmentation Data) FDE Enco...
00403179 0c 1f 00        db[3]
   00403179 [0]             Ch, 1Fh,  0h

                    *************************************************************
                    * Common Information Entry                                  *
                    *************************************************************
                    cie_0040317c                                    XREF[1]:     0040321c(*)
0040317c 18 00 00 00     ddw        18h                                          (CIE) Length
00403180 00 00 00 00     ddw        0h                                           (CIE) ID
00403184 01              db         1h                                           (CIE) Version
00403185 7a 50 4c        ds         "zPLR"                                       (CIE) Augmentation String
         52 00
0040318a 01              uleb128    1h                                           (CIE) Code Alignment
0040318b 78              sleb128    -8h                                          (CIE) Data Alignment
0040318c 1e              db         1Eh                                          (CIE) Return Address Register Co...
0040318d 07              uleb128    7h                                           (CIE) Augmentation Data Length
0040318e 9b              dwfenc     DW_EH_PE_sdata4 | DW_EH_PE_pcrel | DW_EH_PE_in... (CIE Augmentation Data) Personal...
0040318f 19 0f 01 00     ddw        __gxx_personality_v0                         (CIE Augmentation Data) Personal...
00403193 1b              dwfenc     DW_EH_PE_sdata4 | DW_EH_PE_pcrel             (CIE Augmentation Data) LSDA Per...
00403194 1b              dwfenc     DW_EH_PE_sdata4 | DW_EH_PE_pcrel             (CIE Augmentation Data) FDE Enco...
00403195 0c 1f 00        db[3]
   00403195 [0]             Ch, 1Fh,  0h
```

FIGURE 7 – Les deux `CIE` présentes dans la section `.eh_frame` vues par Ghidra

```
(gdb) r AAA
Starting program: /root/safe_01/decrypted_file AAA

Breakpoint 1, 0x0000000000402eb4 in ?? ()
(gdb) p/x $x0
$1 = 0xffffffffead
(gdb) x/1s 0xffffffffead
0xffffffffead: "AAA"
(gdb) rwatch *0xffffffffead
Hardware read watchpoint 2: *0xffffffffead
(gdb) c
Continuing.

Hardware read watchpoint 2: *0xffffffffead

Value = 4276545
0x0000ffffbf4b3d10 in ?? () from /lib64/libgcc_s.so.1
```

Le flag qu'on a passé en argument est donc lu par une fonction de la `libgcc_c.so.1`. Sans arrêter `gdb`, on récupère le pid du programme instrumenté, puis :

```
# cat /proc/1147/maps
...
ffffbf4a8000-ffffbf4b8000 r-xp 00000000 00:02 235    /lib/libgcc_s.so.1
...
```

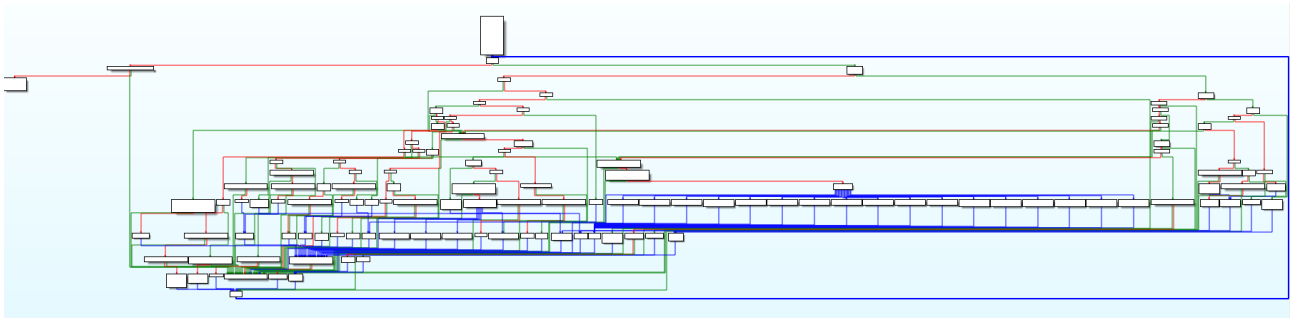

FIGURE 8 – Aspect général de la fonction

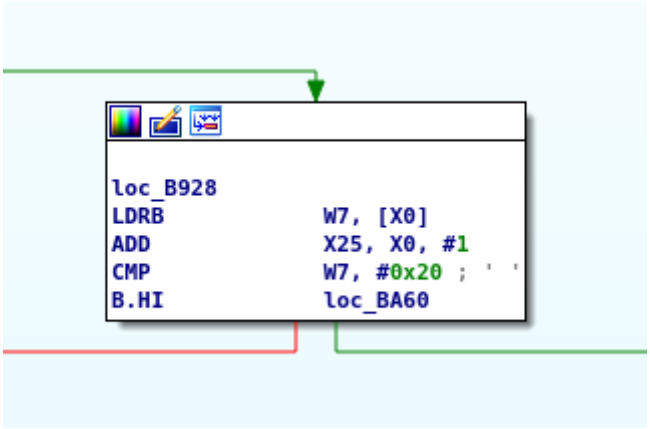

FIGURE 9 – Chargement de l'instruction

Avec l'adresse de base de la bibliothèque, on en déduit l'offset de l'instruction qu'on recherche : `0xffffbf4b3d10 - 0xffffbf4a8000 = 0xbd10`. La fonction contenant cet offset ouverte dans IDA présente un aspect assez caractéristique (Figure 8) : une boucle principale englobant toute la fonction, et qui commence par un déréférencement mémoire (Figure 9), puis une série de choix dichotomiques et de switch cases sur la valeur qu'on vient de charger, chaque cas donnant lieu à une série d'instructions différentes. La majorité de ces cas se termine par l'incrémentation de l'adresse lue en début de boucle, avant de retourner au début.

On reconnaît donc une machine virtuelle dans la libgcc, fort de ce nouveau mot-clé, une recherche nous mentionne DWARF, notamment un article nommé "Exploiting the hard-working dwarf" [3]. Quelques recherches plus tard, on trouve le code source de cette machine virtuelle dans la source de `gcc`. Y sont déclarées toutes les instructions [4], leur encodage et leur sémantique [5]. La fonction qui nous intéresse s'appelle `unwind_stack_op()`.

4.4 Digression : la machine virtuelle DWARF

Il s'agit d'une machine à pile, opérant sur des mots de 64 bits (au moins dans sa version 64 bits), la taille de la pile étant fixée à 64 mots. Ses instructions chargent leurs paramètres sur la pile, et y écrivent leur résultat. Des instructions spéciales permettent de déplacer les éléments de la pile. Les instructions sont de taille variable : la plupart font un octet, cependant certaines instructions viennent avec une valeur immédiate, mesurant de 1 à 8 octets, juste après l'instruction, faisant varier la taille des instructions de 1 à 9 octets.

3. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.651.2604&rep=rep1&type=pdf`
4. Le fichier d'en-tête contenant les définitions des opcodes se trouve en fait dans la source de la glibc, à l'emplacement `sysdeps/generic/dwarf2.h` : `https://github.com/bminor/glibc/blob/master/sysdeps/generic/dwarf2.h`
5. `https://github.com/gcc-mirror/gcc/blob/master/libgcc/unwind-dw2.c`

La machine dispose d'instructions pour lire les registres du programme l'exécutant, ainsi que sa mémoire. Lorsqu'elle termine son exécution, la valeur située au sommet de la pile est retournée au programme appelant.

4.5 Enfin du code !

L'étape suivante est de récupérer le code exécuté par la machine virtuelle DWARF. Pour cela on continue l'analyse dynamique : on place un breakpoint juste avant l'adresse où le bytecode est chargé depuis la mémoire : en `0xb8f8`. On dispose ici à la fois de l'adresse et de la valeur de l'instruction : c'est suffisant pour récupérer tout le programme. On lance le script `gdb` suivant en mode batch, qui nous donne la trace en quelques secondes :

Code 5 – `gdb_gen_trace`

```
# break à main pour que les libs soient chargées
# avant de poser les autres breakpoints
b *0x402e68
r AAA

# breakpoint à l'entrée de la VM
b *0xffffbf4b38b4
commands
        set $inst_cpt = 0

        # breakpoint à chaque instruction
        b *0xffffbf4b38f8
        commands
                # opcode
                x/1bx $x0
                # 8 octets suivants : parfois utilisés comme immédiats
                x/8bx $x0+1
                # optionnel : la pile
                #x/512bx $x26
                set $inst_cpt = $inst_cpt + 1
                continue
        end
        continue
end
continue
```

Avec les paires adresse / instruction ainsi que l'enchaînement des instructions, il est possible de reconstruire le graphe de flot de contrôle du programme (au moins des parties exécutées). Pour cela on code un programme Python qui prend en entrée cette trace et produit un fichier dot contenant le CFG découpé en *basic blocs* (ou blocs de base), les instructions, leurs mnémoniques et valeurs immédiates. On peut aussi ajouter des informations obtenues lors de l'exécution comme le nombre de fois que chaque instruction est exécutée, ou encore les différentes valeurs du pointeur de pile lors des différents passages à une adresse. On obtient la partie gauche de la Figure 10. Chaque ligne se lit ainsi :

```
Adresse : [N] : mnemonic [X,Y,Z,...]
```

où N est le nombre de fois que l'instruction est exécutée dans la trace, et X,Y,Z,... les différentes valeurs de SP (pointeur de pile de la machine DWARF) lors de ces passages.

Après analyse, ce code lit le registre `r31` du programme, y ajoute une constante (`0xa8`), le déréférence une fois, ajoute 8 à la valeur obtenue, puis utilise cette dernière valeur comme adresse source pour copier 32 octets de la mémoire du programme vers les 4 premiers mots de la pile de la VM DWARF. S'ensuit un test vérifant si le 33e octet est nul, ce qui n'est pas le cas et déclenche l'arrêt du programme. Ce morceau de code peut être la copie du flag : l'analyse dynamique confirme que la valeur initiale de `r31` pointe sur la pile (du programme hôte), et qu'en ajoutant la constante `0xa8` on obtient l'adresse de `argv` sur la pile. En ajoutant 8 on obtient donc `argv[1]`, soit le flag entré par l'utilisateur.

On relance la génération de la trace afin de prendre l'autre branche lors du test en `0x40027b` : pour cela on donne en entrée un flag de 32 octets. Cette fois la génération de la trace est

FIGURE 10 – Code DWARF exécuté

beaucoup plus longue : 15 heures environ. En lançant le programme sans instrumentation, on observe effectivement que si l'argument fait 32 octets de longueur alors l'exécution prend environ `0.3s`, contre une durée négligeable sinon.

La trace ainsi obtenue fait 750Mo (penser à rediriger la sortie de gdb à l'extérieur de la VM du challenge, il n'y a pas la place dessus).

```
[user@machine safe_2]$ ./disassembler.py --opcodes opcodes.json -i trace_32 -o
adresses_32.json -d dot_32.json
...
Processed 2939844 instructions
Found 2633 locations
```

Une fois passée dans notre désassembleur, on apprend qu'il y a 2633 adresses différentes, pour un total de presque 3000000 instructions exécutées. Le début du CFG est présenté à droite sur la Figure 10, c'est très long. (Le fichier `opcodes.json` contient un dictionnaire des instructions : mnémoniques utilisées pour le décodage).

Les adresses nous informent également que le code exécuté était bien le blob de données présent dans la section `.gnu.hash` du binaire.

Le code du désassembleur adapté à notre problème est disponible dans le fichier `code/stage3/disassem`

4.6 Simplification de code

Un premier parcours montre qu'après avoir recopié les 32 octets de l'argument, le programme entre dans jusqu'à 5 boucles imbriquées, et fait des déréférencements mémoire à au moins 3 endroits différents. Il se termine par 4 `xor` de mots de la pile avec des constantes de 64 bits, qui sont additionnées et la valeur finale est comparée à 0. Ce test final entraîne une dernière écriture sur la pile avant que le programme ne retourne : il s'agit de la valeur `0x4030b8`, qui correspond à l'adresse dans le binaire de la chaîne de caractères `"Not good"`. En regardant manuellement l'autre branche de ce test (non prise dans la trace), on voit que le programme écrit `0x403098`, soit l'adresse de la chaîne `"That's the correct flag :)"`. Mystère résolu : si le `strcmp()` dans la fonction `main()` n'est jamais exécuté, c'est parce que c'est le code DWARF qui retourne l'adresse de la chaîne à afficher. Il suffit ensuite que la `personality routine` C++ fasse reprendre l'exécution juste avant l'appel à `puts()`, et on n'a pas besoin de la congolexicomatisation.

La longueur du code et la non-limpidité des instructions DWARF (notamment le fait que c'est une machine à pile, inhabituel pour l'auteur) rendent l'analyse directe du code sans issue. Une idée serait alors de simplifier le code. Pour cela, on constate (à l'aide de notre désassembleur) qu'à l'intérieur de la boucle la plus extérieure, SP ne prend que 4 valeurs différentes, espacées de 4 chacune (par exemple 7, 11, 15, 19) pour la première instruction de la boucle. (Ce n'est pas tout à fait vrai, il existe une obscure boucle intérieure faisant 2 tours où SP n'a pas la même valeur, nous reviendrons dessus plus tard).

Pour s'en sortir, on constate qu'il n'existe aucune instruction DWARF qui permette d'accéder à des mots de la pile au-delà de SP (heureusement !). Donc lorsque le programme entre pour la première fois dans la boucle extérieure, il ne peut accéder qu'aux 7 premiers mots de la pile. Quand il entre pour la 2e fois, il a accès aux 11 premiers mots, mais on vient de voir que d'un tour de boucle à l'autre (ou presque...) les valeurs de SP sont décalées de 4. Donc le programme va toujours accéder aux 7 derniers mots de la pile, à chaque tour de boucle. Cette boucle peut donc être vue comme une fonction, itérée 4 fois, et prenant en argument 7 mots, et renvoyant 7 mots. Nous l'appellerons `big_function()`.

Les XOR finaux sont faits sur 4 des 7 mots renvoyés par le dernier appel à `big_function`, tandis que juste avant le premier appel, si le flag donné par l'utilisateur est `ABCD` où chaque lettre représente un mot de 64 bits, la pile se compose comme suit :

```
ABCD0AB
```

où '0' est le mot nul.

A l'intérieur de la fonction `big_function`, SP vaut donc une unique valeur pour chaque instruction (modulo une exception traitée plus bas). On peut donc désaliaser la pile : au lieu de prendre ses opérandes et d'écrire son résultat sur la pile, chaque instruction peut lire et écrire dans des variables locales, numérotées par la valeur de SP à cette instruction. Cela permet de se débarasser de la pile et de l'état SP, ce qui permet d'appliquer de nombreuses techniques de simplification de code, propagation de constantes, etc. Plutôt que de rajouter l'architecture DWARF à un outil existant, on peut recoder le programme en C et le recompiler avec l'option `-O3` de `gcc`, qui fera le travail pour nous. La fonction `instruction_to_c_code()` du désassembleur traduit la sémantique de chaque instruction DWARF en code C, tandis que les fonctions

`init_c_code()`, `end_c_code()`, `instructions_to_c_code()` et `reconstruct_c_code()` génèrent un fichier C qui compile :

```
[user@machine safe_2]$ ./disassembler.py --opcodes opcodes.json -i trace_32 -o
adresses_32.json -d dot_32.json -c program.c
...
Processed 2939844 instructions
Found 2633 locations
[user@machine safe_2]$ gcc -O3 -o program program.c
```

Désaliasing complet : Une boucle résiste toujours à notre méthode : il s'agit d'une boucle à 2 tours vers la fin du code : entre les 2 tours la valeurs de SP varie de 1. Il n'est donc pas possible de transformer les éléments de la pile en variables locales. On triche en utilisant le fait qu'il n'y a que 2 tours : le code C généré va décaler toutes les variables sur la pile d'un indice dans un sens au début du deuxième tour, puis redécaler dans l'autre sens à la fin du tour (lignes 166 à 174 du désassembleur). On compte ensuite sur `gcc` pour simplifier au maximum le code obtenu.

Il est également nécessaire d'inclure le fichier `gnu_hash.h` qui contient les valeurs lues par le code DWARF lors des déréférencements mémoire (tables de lookup). On obtient ce fichier en dumpant la section `.gnu.hash` du binaire avec `readelf`, puis en transformant le résultat avec le script `code/stage3/gnu_hash_dump.py`

Le code C généré produit un programme au comportement identique à l'original. On l'instrumente pour qu'il affiche les 4 mots de 64 bits qu'il utilise dans les xor avec les constantes à la fin. Il servira de référence pour les versions simplifiées, un ensemble de tests permet de s'assurer que les simplifications sont justes :

Code 6 – `test_program.sh`

```
1  #!/bin/bash
2
3  INPUTS=(BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
   CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC AZERTYUIOPMLKJHGFDSQWXCVBNHJKLMP 1234567890
   POLMKNBG6T_TVFcfgI_oKT)
4  #echo ${INPUTS[*]}
5  for INPUT in ${INPUTS[*]} ; do
6          echo "Testing input :" $INPUT
7          ./program $INPUT > output_1
8          ./simplfied_program $INPUT > output_2
9
10         diff -q output_1 output_2
11 done
```

On ouvre ensuite ce programme avec Ghidra, afin de bénéficier du décompilateur C. La fonction `big_function()` passe ainsi de 2600 instructions DWARF à environ 250 lignes de C décompilé. Celle-ci effectue des accès mémoire à seulement deux endroits, signifiant que le troisième lookup vu dans le code DWARF a été totalement résolu et simplifié, tandis que plusieurs boucles ont été déroulées.

Après quelques modifications mineures (changement de types de variables), le code généré par Ghidra (fichier `big_function.c`) compile, et produit les mêmes résultats que le code original, vérifié par le fichier de tests. La suite de l'analyse est manuelle :

— On constate notamment que presque 200 lignes du code C généré, dont une boucle, sont des affectations dépendant d'une seule variable, celle-ci ne prenant en fait initialement que 2 valeurs possibles, résultat d'un test. Comme ce résultat n'est pas connu à la compilation, le compilateur ne peut pas simplifier et laisse les lignes. On duplique alors les 200 lignes en question, pour les mettre dans chacune des 2 branches du test, puis on recompile avec `gcc` et décompile avec Ghidra : ça a marché, le code a été simplifié en

une unique affectation d'une variable dans un `if / then / else`, et le code intéressant ne fait plus que 100 lignes de C.
— `gcc` ayant du mal avec la boucle à 2 tours où on a décalé les éléments de la pile entre chaque tour, on décide de la dérouler à la main, et une passe de compilation / décompilation fait diminuer le code à 80 lignes.
— Parmi les 7 mots de 64 bits pris en argument par la fonction `big_function`, et ceux retournés, seuls 4 sont pertinents : il s'agit des données transformées. Les autres sont des variables locales n'ayant plus de sens à la sortie de la fonction, dont le compteur de la boucle externe dont cette fonction est le corps.

A ce stade, le code simplifié du programme est celui dans `code/stage3/big_function.c`
Pour la suite, on garde le nommage des variables locales fait par Ghidra.

La fonction `big_function` prend 7 arguments en paramètre (on a préféré ne pas enlever les arguments inutiles), et retourne son résultat dans un buffer, qui est ré-interprété comme des mots de 64 bits par la fonction `outer_loop` pour l'appel suivant.

4.7 Chiffrement à clé embarquée

— Le trajet des données dans `big_function` est représenté dans la Figure 11. Les transformations entre les états 1 (initial) et 3 (intermédiaire) et entre les états 3 et 5 (final) sont identiques : c'est la boucle extérieure de `big_function` à 2 tours qu'on a déroulé.
— La fonction Φ prend en argument 128 bits de données et renvoie 128 bits. La fonction Ψ prend 2 entrées de 128 bits en argument et a une sortie de 128 bits. Ce schéma fait penser à une fonction cryptographique : Φ serait une transformation appliquée sur la clé (par exemple une dérivation à partir d'une clé prinicpale), tandis que Ψ prend la clé dérivée et les données en argument, et renvoie le chiffré. On s'oriente donc vers un chiffrement par blocs, d'une taille de bloc de 128 bits, et une taille de clé de 128 bits. On note Enc cette fonction, et $Enc(P, K)$ le chiffré de P avec la clé K.

Le schéma de la Figure 11 peut alors de résumer en :

$$GH = Enc(W_1W_2, Enc(W_3W_4, W_1W_2))$$

$$CD = Enc(W_3W_4, W_1W_2)$$

Il y a donc 2 opérations de chiffrement effectuées, le chiffré de l'une étant la clé de l'autre. Si on dispose de la fonction de déchiffrement Dec (prenant comme arguments dans l'ordre le chiffré et la clé de 128 bits chacun) associée à Enc, c'est gagné car on a alors :

$$W_1W_2 = Dec(GH, CD)$$

$$W_3W_4 = Dec(CD, W_1W_2)$$

On peut donc inverser la fonction `big_function` à partir de sa sortie, et en itérant 4 fois dans `outer_loop` avec comme valeurs initiales les constantes de 64 bits sur lesquelles est fait le test final dans le code DWARF, on peut inverser tout l'algorithme.

Il reste donc juste à trouver la fonction de déchiffrement Dec. On voit que $Enc()$ effectue des opérations sur les bits de la clé et des données qui ne sont clairement pas inversibles (OU logiques entre bits de la clé et des données, lookup de variables de 64 bits dans des tables et réutilisation de ces données comme index des lookup suivants, etc).

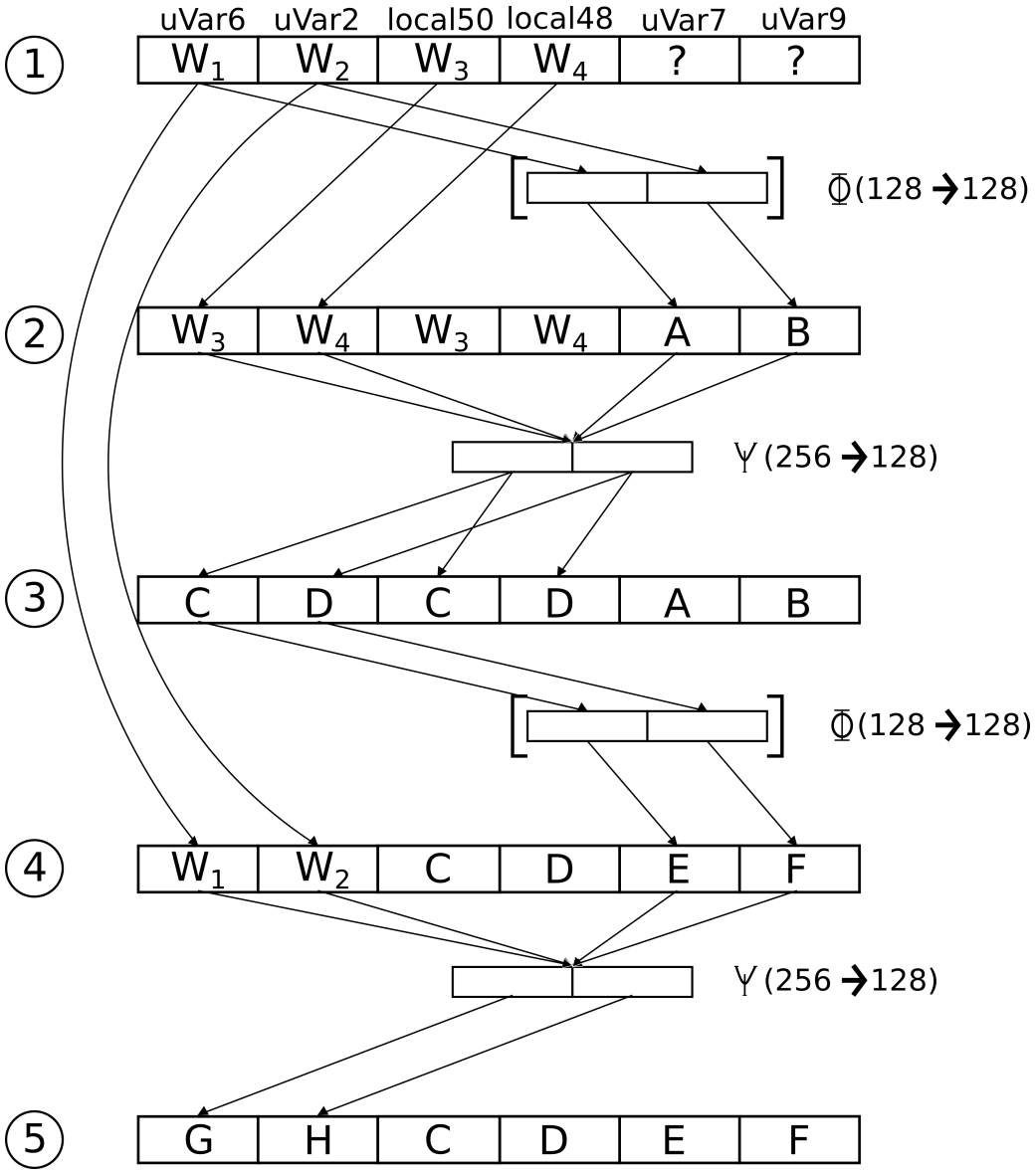

FIGURE 11 – Chemin des données dans `big_function` : Chaque case représente une variable de 64 bits. Les données passées en argument sont W_1, W_2, W_3, W_4. Les données retournées, constituant les arguments de l'appel suivant sont dans l'ordre G, H, C, D. uVar7 et uVar9 sont des variables locales.

Analyse de la fonction *Enc*

C'est la composée de Φ et Ψ. Φ transforme la clé en faisant des lookups dans une table, tandis que Ψ a une boucle extérieure de 15 tours. Dans cette boucle, elle transforme la clé qu'elle a obtenu de Φ en effectuant d'autres lookups et le fameux test dont découlent 1500 lignes de code DWARF qui obscurcissent le programme, puis utilise cette clé locale (clé de round) dans la dernière boucle (celle à 2 tours dont les tours n'ont pas la même hauteur de pile). Dans cette boucle, la clé de round est découpée en 2 mots de 64 bits, utilisés chacun dans un tour de la manière suivante : Une transformation non inversible est appliquée à la moitié du texte clair et de la clé de round ($(64, 64) \to 64$), puis ce résultat est utilisé dans une transformation inversible (combinaison de XOR et de décalages / permutations de bits), faisant intervenir ce résultat et l'autre moitié du texte clair.

La même transformation est appliquée dans le 2e tour de boucle, en utilisant la 2e moitié de la clé de round et l'autre moitié du texte clair. Le résultat est utilisé comme entrée pour le

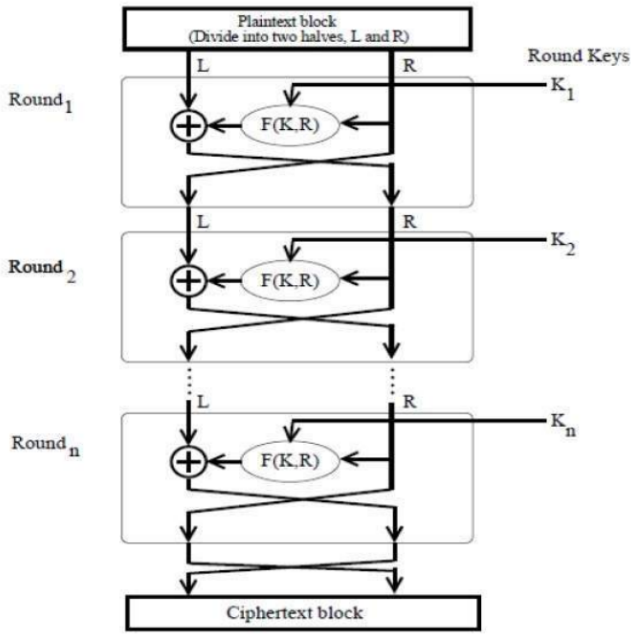

FIGURE 12 – Chiffrement de Feistel

round suivant.

Après quelques recherches sur les schémas de cryptographie symétrique, notre fonction *Enc* ressemble beaucoup à un schéma de Feistel (Figure 12), à la différence près qu'au lieu du XOR sur la gauche, nous avons une transformation différente, peu importe tant qu'elle est inversible. On l'appellera Quasi-Feistel. Le gros intérêt de ce chiffrement est qu'il n'est pas nécessaire d'inverser les lookups (fonction F sur le schéma), car son entrée est reportée dans la sortie du round.

Ainsi, chaque tour de la boucle à 15 tours dans la fonction Ψ correspond à deux rounds dans le chiffrement de Quasi-Feistel, et transforme les 128 bits des données en utilisant 128 bits de clé de round. Autre différence par rapport au schéma de Feistel : il y a 2 transformations inversibles différentes utilisées, à tour de rôle.

On remarque aussi que les clés de round peuvent être calculées à l'avance : on les met dans un tableau de $30 * 64$ bits. L'opération de déchiffrement consiste alors à remonter le flot de données à partir du chiffré, en utilisant les clés de round en sens inverse (d'où l'intérêt de toutes les générer à l'avance, et de ne pas les dériver une par une comme dans la fonction Ψ.

Tous ces éléments mis bout à bout permettent d'écrire le programme de déchiffrement du flag : `code/stage3/decrypt_flag.c`

En l'exécutant on obtient ainsi le flag :

```
SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}
```

Au moment d'ajouter le flag au keystore de la VM, le script nous donne une information utile concernant l'étape suivante :

```
[w] You must reboot in order to decrypt Secure OS
```

Et effectivement lors du redémarrage est affichée cette ligne :

```
NOTICE:  Booting Secure-OS
```

Remarque : Le test final est de la forme : $w_1 \oplus c_1 + w_2 \oplus c_2 + w_3 \oplus c_3 + w_4 \oplus c_4 = 0$ où les w sont la sortie de l'algorithme et les c des constantes. Si les opérations sont faites sans overflow, la solution est unique car la somme nulle revient à chaque xor nul, soit $w_i = c_i \forall i \in [1, 4]$, mais

dans la machine DWARF les mots font 64 bits donc pour tout triplet (w_1, w_2, w_3) il existe un unique $w_4 = ((-(w_1 \oplus c_1 + w_2 \oplus c_2 + w_3 \oplus c_3))mod 2^{64}) \oplus c_4$ qui rend le test vrai. Il y a donc 2^{192} flags valides.

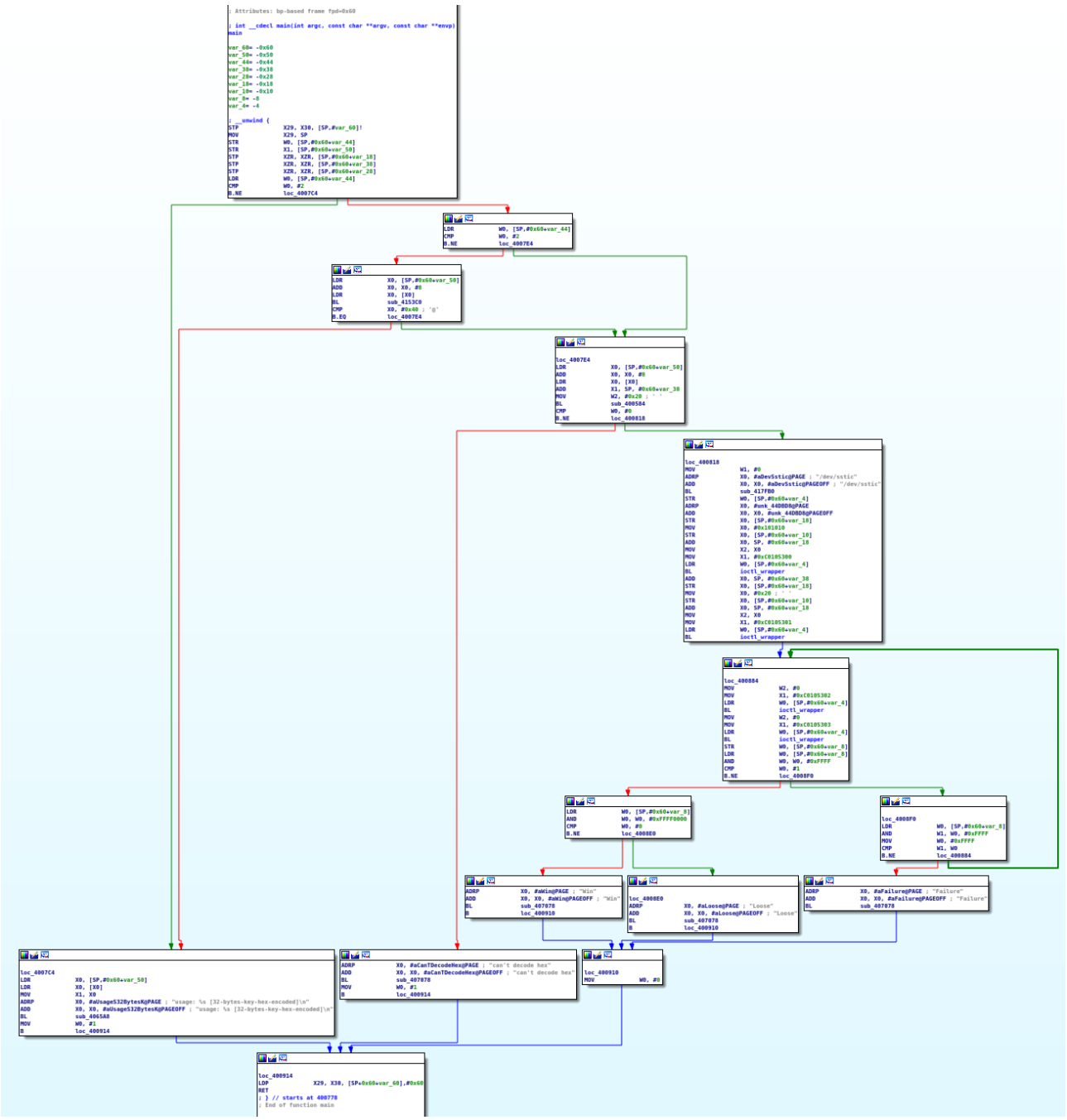

FIGURE 13 – Fonction `main()`

5 We need to go deeper

5.1 Un nouveau binaire

Le binaire se présente sous la même forme que pour l'épreuve précédente : architecture `aarch64`, il prend le flag en argument et affiche l'information `Win / Loose` sur la sortie standard en fonction de sa validité. La différence est qu'il est linké statiquement. On le charge avec IDA, et après identification de la fonction `main()` (comme précédemment en cherchant les références aux chaînes de caractères `Win` et `Loose`), on la trouve à l'adresse `0x400778` et on obtient le graphe de la Figure 13. On remarque aussi une grosse zone de données commençant à l'adresse `0x44dbd8`, et faisant environ 1Mo.

Le programme vérifie que l'argument est présent, vérifie qu'il fait bien 64 octets de long, le décode depuis un encodage hexadécimal (ce qui échoue si ce n'était pas le bon encodage). Le flag

décodé fait désormais 32 octets. Le programme entre alors dans la partie intéressante : il ouvre le périphérique de type caractères `/dev/sstic`, fait 2 appels successifs à `ioctl_wrapper()` en lui passant en argument dans `X0` un pointeur respectivement sur la zone de données commençant en `0x44dbd8` et sur le flag, avec des options différentes pour chaque appel. Un autre argument entier est aussi passé, sur la pile cette fois-ci : pour le 2e appel il vaut `0x20`, ce qui est la taille du flag, donc probablement la longueur des données passées dans le pointeur. Pour l'appel 1 il vaut `0x101010`, on ne sait pas à quoi il correspond pour l'instant. Nous appellerons ces appels 1 et 2 respectivement.

Le programme entre ensuite dans une boucle : A chaque tour de boucle il effectue 2 appels à `ioctl_wrapper()` avec de nouvelles options, mais aucune donnée relative au flag. Nous appellerons ces appels 3 et 4 respectivement. La condition d'arrêt de cette boucle porte sur les 16 bits de poids faible de la valeur de retour de l'apppel 4. En cas d'arrêt (si cette valeur est 1), l'information Valide / non valide provient des 16 bits de poids fort de la même valeur de retour : s'ils valent 0 c'est le bon flag, sinon c'st le mauvais flag. Il existe une autre sortie possible : si les 16 bits de poids faible de la valeur retournée par l'appel 4 sont `0xffff` le programme s'arrête en affichant `Failure`.

La fonction `ioctl_wrapper()` fait ce qu'on pense : elle fait l'appel système `ioctl()` (numéro `0x1d` dans `X8` selon la convention sur `aarch64`) sur le périphérique `/dev/sstic` en transmettant les arguments sans les modifier.

Les options passées en argument à l'appel système sont :

— Appel 1 : `0xC0105300`
— Appel 2 : `0xC0105301`
— Appel 3 : `0xC0105302`
— Appel 4 : `0xC0105303`

Dernière chose possible en espace utilisateur : compter les tours de boucle : on lance le programme avec strace :

```
# strace -c ./decrypted_file
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Loose
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.95    2.620567         139     18735           ioctl
  0.01    0.000329         329         1           readlinkat
  0.01    0.000295          73         4           brk
  0.01    0.000233         233         1           write
  0.01    0.000142         142         1           openat
  0.00    0.000122         122         1           execve
  0.00    0.000070          70         1           fstat
  0.00    0.000068          68         1           uname
------ ----------- ----------- --------- --------- ----------------
100.00    2.621826                 18745           total
```

Déjà, le nombre d'appels à `ioctl` est impair, étrange car notre programme ne fait qu'un nombre pair d'appels à `ioctl_wrapper()`, quelque soit le nombre de tours de boucle. En affichant les arguments des appels (strace sans l'option `-c`), on remarque que le dernier appel est différent : il sert à configurer la console avant l'affichage sur la sortie standard. (pour cela, on peut chercher tous les appels à `ioctl()` dans le binaire en faisant une recherche binaire sur les octets de l'instruction `MOV X8, #0x1D`, caractéristique de l'appel). Cet appel n'a rien à voir avec le challenge, et on constate en effet que si on relance la même commande depuis une connexion SSH, il n'a pas lieu.

Le programme fait donc 18734 appels à `ioctl()`, soit en enlevant les 2 appels d'initialisation : $(18734 - 2)/2 = 9366$ tours de boucle. Ce nombre n'a pas l'air de changer si on exécute le programme avec des entrées différentes.

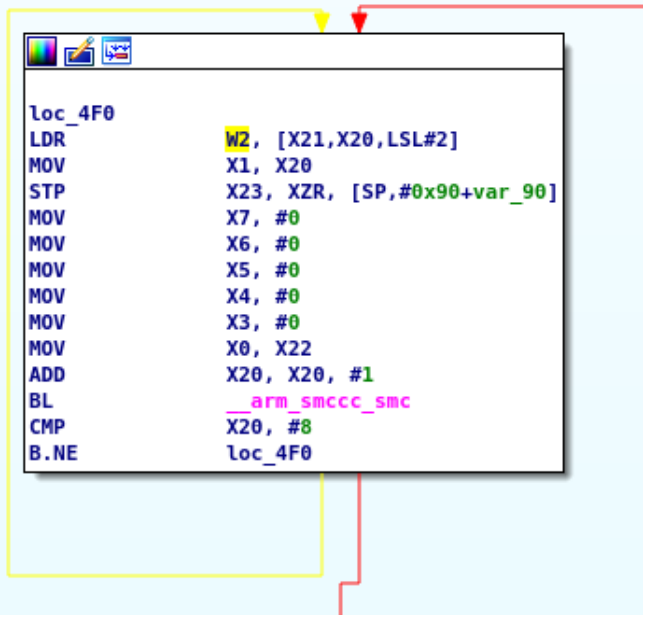

FIGURE 14 – Tranmission du flag au Secure Monitor

5.2 Un étage en dessous

La suite de l'exécution se déroule donc dans le noyau. On se rappelle alors du module `sstic` chargé dans la VM qu'on avait vu au début. On trouve le fichier à l'emplacement `/lib/sstic.ko`, et après analyse sous IDA, ce module est bien responsable de la présence du périphérique `/dev/sstic`. Cette fois-ci les symboles sont présents, et la fonction intéressante s'appelle `sstic_ioctl()`. Elle prend des branches différentes selon la valeur de l'option passée par le programme en espace utilisateur (récupéréée dans `w1` à l'entrée de la fonction). La plupart de ces branches se terminent par un ou plusieurs appels à `__arm_smccc_smc`, qui après quelques recherches est un wrapper autour de l'instruction SMC, qui est en `aarch64` l'appel permettant de descendre en mode Secure Monitor, le mode le plus privilégié du processeur.

Plus précisément, les branches correspondant aux appels 1, 3 et 4 donnent lieu à un unique appel à `__arm_smccc_smc`, tandis que la branche suivie par l'appel 2 entre dans une boucle faisant 8 tours, chaque tour faisant un appel à `__arm_smccc_smc`. En détail :

— L'appel 1 (`0xC0105300`) prend en argument un pointeur vers une section de données du binaire en espace utilisateur, et un entier, recopie les données du pointeur en espace noyau en interprétant l'entier comme une taille, puis déclenche un appel `SMC` ayant pour numéro `0x83010004`, avec en argument le pointeur vers les données en espace noyau et la même taille (`0x101010`). Il semble transmettre environ 1Mo de données au Secure Monitor.

— L'appel 2 (`0xC0105301`) prend en argument un pointeur vers le flag entré par l'utilateur sur 32 octets, recopie ce flag en espace noyau, puis fait 8 appels `SMC` ayant pour numéro `0xF2005003` dans une boucle (Figure 14). Chaque appel prend 2 arguments : le compteur de boucle dans `X1` (probablement un indice), et 4 octets du flag dans `X2`. Cela permet de transmettre les 32 octets du flag au Secure Monitor.

— L'appel 3 (`0xC0105302`) ne prend pas d'argument, et est convertit en un appel `SMC` ayant pour numéro `0xF2005001`, sans autres arguments.

— L'appel 4 (`0xC0105303`) ne prend pas d'argument, et est convertit en un appel `SMC` ayant pour numéro `0xF2005002`, sans autres arguments.

5.3 Encore un étage en dessous

TrustZone

ARM TrustZone est une fonctionnalité de certains processeurs ARM ajoutant un niveau d'exécution supplémentaire, en plus des modes utilisateur, noyau et hyperviseur. C'est le mode le plus privilégié, et il permet de séparer deux mondes : le monde sécurisé et le monde non sécurisé. Chaque monde contient un niveau d'exécution utilisateur, noyau et hyperviseur. Le Secure Monitor s'exécute lui-même en mode sécurisé. Chaque niveau dans chaque monde a son espace d'adresses. Typiquement le monde sécurisé a accès à la mémoire du monde non sécurisé, mais pas l'inverse. Les niveaux d'exécution sont nommés EL_i (Exception Level) :

— EL_0 : Espace utilisateur
— EL_1 : Espace noyau
— EL_2 : Hyperviseur
— EL_3 : Secure Monitor

Chaque niveau a accès aux registres des niveaux de privilège inférieurs dans le même monde, et possède des registres qui lui sont propres (notamment pour gérer les exceptions).

Il existe donc théoriquement 7 espaces d'adressage possibles : EL_0 non sécurisé, EL_1 non sécurisé, EL_2 non sécurisé, EL_3 (forcément sécurisé), EL_2 sécurisé, EL_1 sécurisé et EL_0 sécurisé.

Revenons à notre VM

Le processus de démarrage d'un processeur équipé de TrustZone est bien expliqué ici : `https://github.com/ARM-software/arm-trusted-firmware`. Les différentes étapes de boot, nommées `BL1` et `BL2` chargent successivement leur code et appellent l'étape suivante. `BL2` appelle `BL31` qui est l'environnement du Secure Monitor. Celui-ci lance alors dans l'ordre `BL32` : le secure OS (qui s'exécute en niveau EL_1 mode sécurisé), puis `BL33` : le système d'exploitation non sécurisé qui s'exécute en niveau EL_1 : dans le cas de notre VM il s'agit d'une application EFI qui va à son tour démarrer le noyau Linux.

On s'aide des messages affichés au démarrage de la VM pour chercher le code intéressant :

```
NOTICE:  Booting SSTIC ARM Trusted Firmware
KEYSTORE: AES Key already decrypted
NOTICE:  Loading image id=1
NOTICE:  BL1: Booting BL2
KEYSTORE: BL2 got key : key_type:0x02, key_len:0x20
HEXDUMP :
-----------------------------------------------
53 53 54 49 43 7b 61 39 34 37 64 36 39 38 30 63
63 66 37 62 38 37 63 62 38 64 37 63 32 34 36 7d
-----------------------------------------------
NOTICE:  Loading image id=3
KEYSTORE: BL2 got key : key_type:0x02, key_len:0x20
HEXDUMP :
-----------------------------------------------
53 53 54 49 43 7b 44 77 34 72 66 5f 56 4d 5f 31
73 5f 63 6f 30 6c 5f 69 73 6e 5f 74 5f 49 74 7d
-----------------------------------------------
NOTICE:  Loading image id=4
KEYSTORE: BL2 got key : key_type:0x02, key_len:0x20
HEXDUMP :
-----------------------------------------------
53 53 54 49 43 7b 61 39 34 37 64 36 39 38 30 63
63 66 37 62 38 37 63 62 38 64 37 63 32 34 36 7d
-----------------------------------------------
NOTICE:  Loading image id=5
NOTICE:  BL1: Booting BL31
NOTICE:  BL31: Initializing BL32
NOTICE:  Booting Secure-OS
UEFI firmware (version  built at 00:01:39 on Feb 25 2019)
EFI stub: Booting Linux Kernel...
```

On ouvre cette fois-ci le fichier `rom.bin` de la VM avec IDA qui contient le `BL1`, et on recherche les endroits où ces messages sont affichés. Ce fichier semble contenir à la fois des fonctions standard de `arm-trusted-firmware` et des fonctions spécifiques au challenge.

Il contient entre autres le code pour lire et écrire des clés dans le keystore, une implémentation d'AES pour déchiffrer la flash de la VM, le code du niveau 1, dont une implémentation de la fonction de déchiffrement RSA non protégée contre les attaques par canaux auxiliaires, ainsi que 2 grandes chaînes hexadécimales, qui sont probablement la clé AES chiffrée en RSA et le module de la clé publique du niveau 1.

Pour le reste de l'épreuve, nous aurons besoin de débugguer la VM en entier, à tous les niveaux d'exécution. On rajoute donc les options `-s -S` à la ligne de commande de `qemu` afin d'activer le serveur `gdb` à distance, et on s'y connecte avec `gdb`. On fait également l'hypothèse que les espaces mémoire des différents niveaux d'exécution sont distincts, permettant ainsi de rapidement repérer les changements dans une trace.

Le démarrage du Secure Monitor est notifié par le message `BL1: Booting BL31`. Cette chaîne est présente à l'adresse `0x5b16` dans la ROM, et rien n'y fait référence dans cette ROM. On place alors un breakpoint à cette addresse, qui nous amène à l'instruction à l'addresse `0x42e4`. En exécutant pas à pas depuis cette instruction, les caractères de la chaîne s'affichent un à un à l'écran, on est donc dans une fonction d'affichage (`printf` ou assimilé). Cette fonction comence en `0x42a8`. Pour gagner du temps, on place un breakpoint à la sortie de la fonction, en `0x4348`. En continuant l'exécution pas à pas à partir de là, on saute au bout d'une vingtaine d'instructions en `0xe030000`. L'adresse bien alignée fait penser au début d'une zone mémoire, c'est peut-être le Secure Monitor. On redémarre la VM en posant un breakpoint directement à cette addresse, puis on réalise un dump mémoire à partir de cette addresse, par exemple de 16Mo :

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000000000000000 in ?? ()
(gdb) b *0xe030000
Breakpoint 1 at 0xe030000
(gdb) c
Continuing.

Breakpoint 1, 0x000000000e030000 in ?? ()
(gdb) dump binary memory memory.bin 0xe030000 0xf030000
```

Cette méthode permet de contourner le chiffrement de la flash de l'étape 1, en accédant aux données déchiffrées par le bootloader.

En ouvrant le fichier obtenu avec IDA, on découvre beaucoup de code et de données. Inventaire non exhaustif :

- `0xe030000 - 0xe037000` : Beaucoup de code, contenant des instructions non reconnues par IDA et Ghidra.
- `0xe0380b0 - 0xe038850` : Des chaînes de caractères indiquant qu'on est bien dans BL31 (`0xe038131` : "BL31 : Initializinn BL31"). En posant un watchpoint dessus, on constate qu'elle est bien utilisée dans des fonctions avec des addresses en `0xe030000`
- `0xe200000 - 0xe202e80` : Encore du code, contenant encore des instructions non reconnues.
- `0xe204034` : La chaîne de caractères "Booting Secure-OS", qui est affichée quand on a démarré la VM, et uniquement après avoir ajouté le flag de l'étape précédente au keystore.
- `0xe205000` : Encore des données, mais qui semblent se répéter en étant alignées sur 4 octets et font penser à des instructions (en plus de contenir de nombreuses occurences de `1337` et `1338`. Ce fut le plus long à comprendre : ce sont en fait des instructions ARM

32 bits.

Afin de s'assurer que les instructions présentes aux addresses en `0xe200000` font bien partie du Secure-OS (donc BL32), on se souvient du message affiché lors de la validation de l'étape précédente :

```
[w] You must reboot in order to decrypt Secure OS
```

Si le secure OS est chiffré et la clé obtenue à l'étape précédente permet de le déchiffrer, la mémoire en `0xe200000` doit contenir des données chiffrées (ou rien du tout, ou tout autre chose) lorsqu'on démarre la VM sans cette clé dans le keystore. On réalise alors une image mémoire de 16Mo à partir de `0xe200000` avec et sans la clé, et on observe la différence :

```
[user@machine safe_3]$ diff --suppress-common-lines -y <(xxd memory.bin) <(xxd
memory_nokey.bin) | less
0001f750: 4b45 5900 0000 0000 0200 00 | 0001f750: 0000 0000 0000 0000 0000 00
0001f760: 5fb3 a83d 1fd9 7137 0760 19 | 0001f760: 0000 0000 0000 0000 0000 00
0001f770: 66fb 6b32 618d 162e 00cd ee | 0001f770: 0000 0000 0000 0000 0000 00
0001fa10: 40fa 040e 0000 0000 c8ff ff | 0001fa10: 0a00 0000 0000 0000 c8ff ff
0001fa20: 80fa 040e 0000 0000 80fa 04 | 0001fa20: 1c00 0000 0000 0000 1047 00
0001fa30: 40fa 040e 0000 0000 c8ff ff | 0001fa30: 50fa 040e 0000 0000 9c46 00
0001fa40: 0000 0000 0000 0000 1010 00 | 0001fa40: 40fb 040e 0000 0000 1010 00
0001fa50: 4000 0000 0000 0000 3f00 00 | 0001fa50: b0fa 040e 0000 0000 ec2d 00
0001fa60: c001 050e 0000 0000 0000 00 | 0001fa60: 3003 0000 0000 0000 c0fe 04
0001fa70: 1869 0000 0000 0000 185d 00 | 0001fa70: 90fb 040e 0000 0000 fa63 01
0001fa80: e000 050e 0000 0000 fc11 00 | 0001fa80: 0064 010e 0000 0000 fc63 01
0001fa90: 0100 0000 0000 0000 0000 00 | 0001fa90: 0000 0000 0000 0000 0000 00
0001fab0: 0000 0000 0000 0000 0000 00 | 0001fab0: 40fb 040e 0000 0000 9c17 00
0001fac0: 0000 2000 0200 0000 5353 54 | 0001fac0: 40fb 040e 0000 0000 40fb 04
0001fad0: 3437 6436 3938 3063 6366 37 | 0001fad0: 00fb 040e 0000 0000 c8ff ff
0001fae0: 3864 3763 3234 367d 0000 00 | 0001fae0: 40fb 040e 0000 0000 40fb 04
0001faf0: 0000 0000 0000 0000 0000 00 | 0001faf0: 00fb 040e 0000 0000 c8ff ff
0001fb50: 3002 050e 0000 0000 0100 00 | 0001fb50: 3002 050e 0000 0000 0000 00
0001fdb0: 4b45 5900 0000 0000 0200 00 | 0001fdb0: 0000 0000 0000 0000 0000 00
0001fdc0: 5fb3 a83d 1fd9 7137 0760 19 | 0001fdc0: 0000 0000 0000 0000 0000 00
0001fdd0: 66fb 6b32 618d 162e 00cd ee | 0001fdd0: 0000 0000 0000 0000 0000 00
0001fec0: 4b45 5900 0000 0000 0200 00 | 0001fec0: 0000 0000 0000 0000 0000 00
0001fed0: 5353 5449 437b 4477 3472 66 | 0001fed0: 0000 0000 0000 0000 0000 00
0001fee0: 735f 636f 306c 5f69 736e 5f | 0001fee0: 0000 0000 0000 0000 0000 00
00020280: a400 0000 0000 0000 8e00 00 | 00020280: db00 0000 0000 0000 5700 00
00020290: f700 0000 0000 0000 7400 00 | 00020290: 2b00 0000 0000 0000 4800 00
000202a0: 8200 0000 0000 0000 6500 00 | 000202a0: b400 0000 0000 0000 2a00 00
000202b0: 1063 010e 0000 0000 6000 00 | 000202b0: 1063 010e 0000 0000 ca00 00
000202c0: e300 0000 0000 0000 6043 01 | 000202c0: 7800 0000 0000 0000 6043 01
000202d0: 0100 0000 0000 0000 d116 01 | 000202d0: 0000 0000 0000 0000 d116 01
001d0000: 0080 0110 00c0 18d5 df3f 03 | 001d0000: a390 7636 5875 b3b9 6c65 41
001d0010: 4101 82d2 0010 38d5 0000 01 | 001d0010: d418 b5eb 8b5a 6081 3816 34
001d0020: df3f 03d5 e07e 0210 c1fe 0d | 001d0020: 31d9 17e7 237f a5bf 86f8 e5
001d0030: 0509 0094 e001 0058 0102 00 | 001d0030: 22fe 9e59 4e7b 417c 6686 3c
001d0040: 0002 0058 2102 0058 0d09 00 | 001d0040: 036c d97d 1b8c 2303 905f 42
001d0050: 7305 0094 ce05 0094 7600 00 | 001d0050: 16d5 7af0 9c8e f80b 32b6 11
001d0060: 0040 bed2 0300 00d4 0000 00 | 001d0060: c927 8cf1 02c1 a483 6639 59
001d0070: 8053 210e 0000 0000 2007 00 | 001d0070: 5fba aaf0 40da 874b c47d 04
001d0080: 00c0 210e 0000 0000 0000 00 | 001d0080: 9f7c 80a5 6b22 0165 77a4 11
001d0090: 5700 0014 4f00 0014 1a00 00 | 001d0090: 1b7c d582 32df e17f 8610 4d
001d00a0: 4500 0014 2a00 0014 2f00 00 | 001d00a0: a4d5 d338 10fe 0203 3dfa 6d
001d00b0: 0e00 0014 5700 0014 9e00 00 | 001d00b0: 6b1e 63e9 376b fb80 591c 80
...
```

A gauche avec la clé de l'étape précédente, à droite sans.

En plus des clés présentes à gauche et pas à droite (le keystore est présent en `0x1f750`), on observe effectivement une différence à partir de l'addresse `0x1d0000`, qui correspond à l'addresse `0xe200000` car les images mémoires ont été réalisées en `0xe030000` : `0xe200000 = 0xe030000 + 0x1d0000` : avec la clé on retrouve les instruction qu'on a vues sous IDA, et sans la clé c'est un blob binaire quelconque. C'est donc bien cette partie qui est déchiffrée et qui correspond au Secure-OS.

5.4 Allers-retours

Afin de déterminer le point d'entrée dans les appels du Secure Monitor, on place un breakpoint dans le noyau Linux sur juste avant l'appel à `__arm_smcc_smc`, par exemple en `0xffff00000859022c`, correspondant à l'appel à SMC de l'`ioctl()` 3 (le premier dans la boucle en espace utilisateur). On exécute ensuite pas à pas jusqu'à arriver dans le Secure Monitor. L'addresse d'arrivée est `0xe037404`.

On recommence en plaçant un breakpoint en `0xe037074` et en affichant `X0`, qui contient le numéro de l'appel SMC fait par le noyau Linux (la documentation de la convention d'appel SMC [6] aide à savoir où retrouver ce numéro, par contre les valeurs présentes ont l'air spécifiques au challenge), on peut générer une trace d'exécution complète listant tous les appels à SMC. Il y en a plus de 50000, soit beaucoup plus que le nombre d'appels à `ioctl()`. Par ailleurs l'enchaînement des valeurs n'est pas périodique et ne semble suivre aucune logique.

L'analyse dynamique ayant bien fonctionné dans l'étape précédente, on recommence. On décide cette fois de tracer toutes les instructions ARM exécutées ailleurs que dans le programme en espace utilisateur et le noyau Linux (qu'on a déjà analysés). Pour cela on replace un breakpoint juste avant l'appel à SMC dans le noyau, puis on exécute pas à pas tant qu'on n'est pas retourné dans le noyau.

Script `gdb` :

```
b *0x0
target remote localhost:1234

define print_smc_path
        p/x $pc
        s
        p/x $pc
        s
        p/x $pc
        while ($pc < 4294967296)
                # && $pc >= 236978176)
                x/1i $pc
                s
        end
        continue
end

b *0xe030000
commands
        set step-mode on
        # ioctl call_1
        b *0xffff000008590198
        # ioctl call_2
        #b *0xffff000008590518
        # ioctl call_3
        #b *0xffff00000859022c
        # ioctl call_4
        #b *0xffff000008590414
        commands
                print_smc_path
                continue
        end
        continue
end
continue
```

Quand on lance le binaire dans la VM la trace de l'appel à SMC dans l'appel `ioctl` 1 démarre. On peut alors faire la même chose pour les 3 autres appels en changeant l'addresse du breakpoint, voire une trace globale (peu utile en pratique). La genération des 4 traces prend environ 12 heures, pour un total de 42000000 instructions.

On adapte ensuite notre désassembleur pour qu'il se contente de reconstituer les blocs de base et d'afficher le CFG de la trace. Il devient un simple afficheur de CFG (fichier `code/stage4/cfg_build`

6. http ://infocenter.arm.com/help/topic/com.arm.doc.den0028b/ARM_DEN0028B_SMC_Calling_Convention.pdf

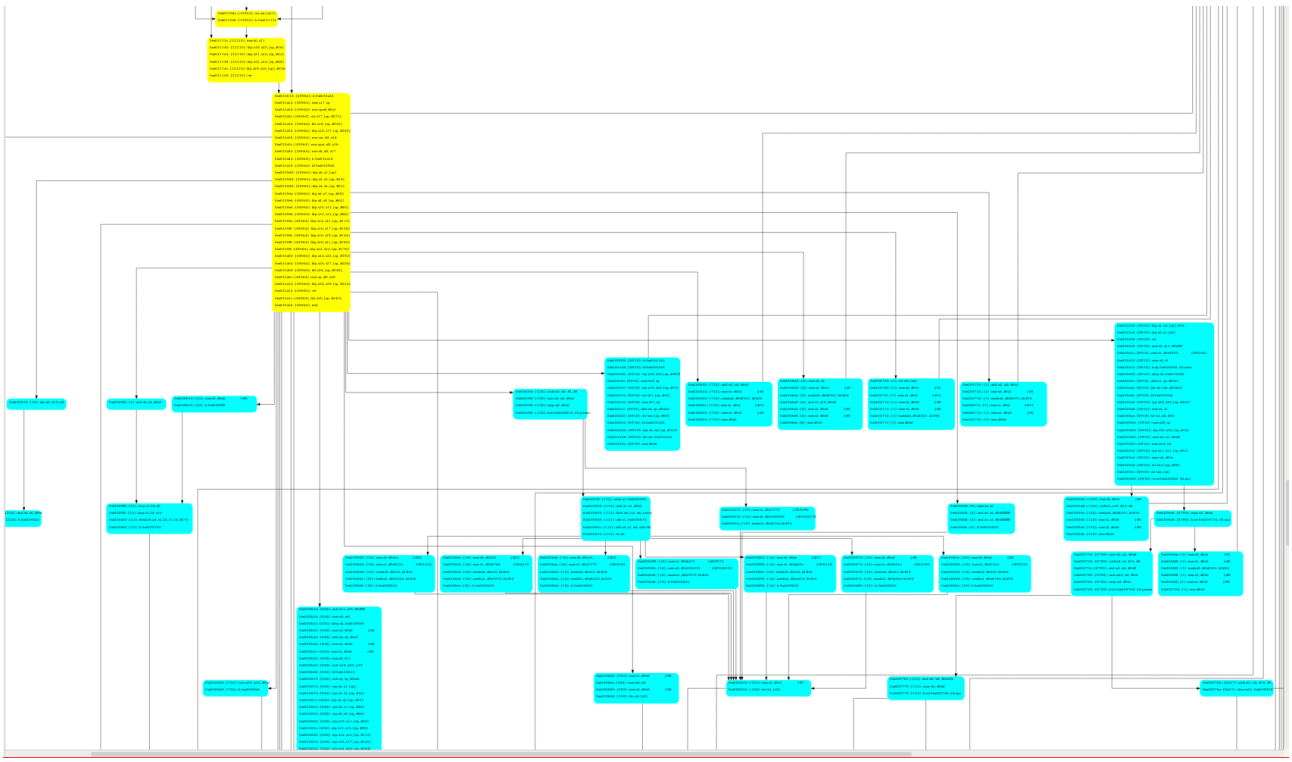

FIGURE 15 – Point d'entrée dans le Secure-OS : en jaune les instruction du Secure Monitor, en bleu celles du Secure-OS. La dernière instruction avant le saut vers le Secure-OS est un `ERET`

On ajoute aussi une couleur différente dans les blocs de base, selon que leur addresse est dans le Secure Monitor (`0xe030000`) ou dans le Secure-OS (`0xe200000`).

Le CFG affiché est assez complexe. La figure 15 en montre un tout petit extrait : le point d'entrée (il est unique) dans le Secure-OS en remontant du Secure Monitor via une instruction `ERET`).

Autre exemple : la Figure 16 montre la trace du premier appel, qui effectue un `memcpy` de `0x101010` octets provenant du noyau Linux.

On observe également des instructions intéressantes :

— Opérations sur les complexes en virgule flottante : `FCADD`, `FCMLA`

— Opérations cryptographiques : `AESE`, `AESD`, `SM4E`

5.5 Encore une VM

Après un temps relativement long et l'analyse des fonctions du Secure-OS, on tombe sur une fonction en `0xe2005a4` dont l'aspect rappelle celle de la VM DWARF (Figure 17).

Rétro-ingéniérie de la VM

L'analyse de cette fonction avec IDA est grandement facilitée par la trace qu'on a réalisée : en effet elle permet de suivre les appels à `SMC` qui sont effectués depuis le Secure-OS, et qui retournent parfois dans le monde non sécurisé, parfois dans le Secure-OS.

Cette fonction prend en arvgument une valeur dans `X1`, dont seuls les 3 octets de poids faible sont non nuls (mise à 0 des autres un peu avant dans la trace). La fonction prend alors des chemins différents selon la valeur des bits 20 à 23, ainsi que 19 et 20, d'abord par une suite de tests, puis par des switch case. On en déduit que l'opcode est encodée sur 4 ou 6 bits (les 2 derniers pouvant être des flags).

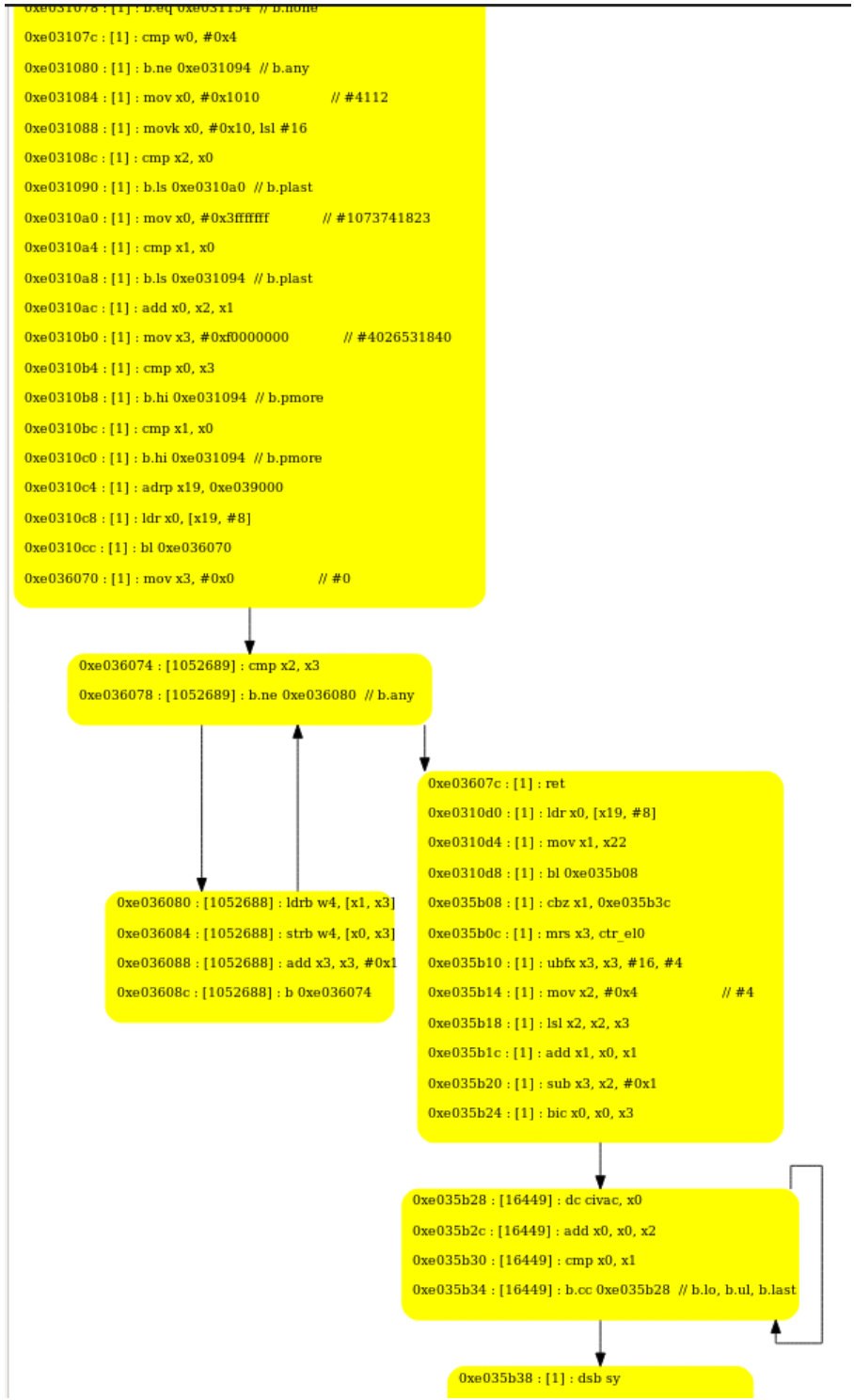

FIGURE 16 – `memcpy()` de `0x101010` octets dans la trace de l'appel 1

Le fait que c'est une VM est confirmé par l'analyse du morceau de code présenté en Figure 18. La fonction `call_32bits` met l'addresse présente en `X4` dans `ELR_EL1`, qui est le registre contenant l'adresse de retour d'une exception quand le programme retourne en `EL1`, puis fait `ERET`, retournant effectivement en `EL1`.

Le code 32 bits présent en `inc_pc` (Figure 19) effectue un appel système `1338` avec en paramètre `0x83010001` et 15. En suivant cet appel dans la trace, il charge une valeur à un offset `15*16` par rapport à une addresse de base, "déchiffre" les 16 octets à partir de cette addresse avec l'instruction `AESE` (qui effectue un tour de chiffrement AES) et une clé présente en dur dans le Secure-OS, puis prend les octets 5 à 8 (en commençant la numérotation à 0), et

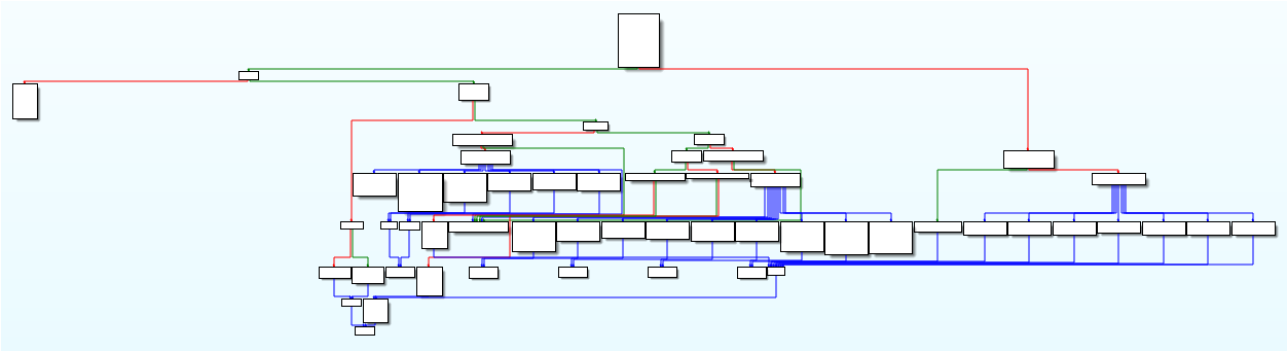

FIGURE 17 – Fonction en `0xe2005a4` : probablement une VM, condition d'arrêt sur la gauche.

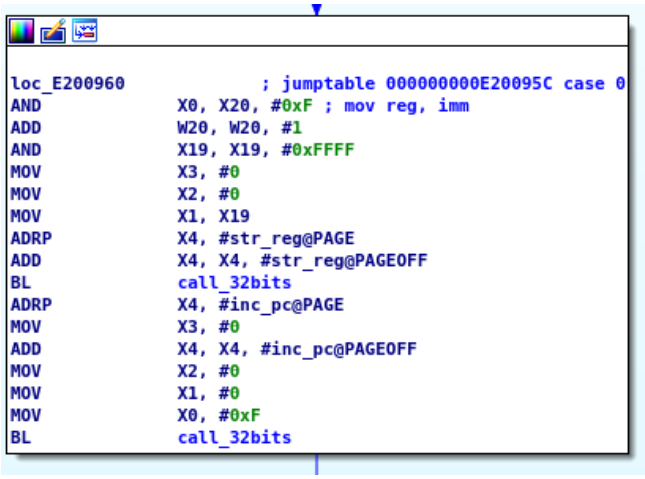

FIGURE 18 – Détail d'une branche de la fonction en `0xe2005a4`. Les symboles `str_reg` et `inc_pc` ont été rajoutés, ainsi que le commentaire `mov reg, imm`

.

```
ROM:0E205078                MOV             R8, R0
ROM:0E20507C
ROM:0E20507C loc_E20507C                            ; DATA XREF: sub_E205000+30↑r
ROM:0E20507C                MOV             R1, R0
ROM:0E205080                MOV             R0, #0x8301
ROM:0E205084                MOV             R0, R0,LSL#16
ROM:0E205088                ADD             R0, R0, #1
ROM:0E20508C                SVC             0x1338
ROM:0E205090                MOV             R1, R0
ROM:0E205094
ROM:0E205094 loc_E205094                            ; DATA XREF: ROM:0E205054↑r
ROM:0E205094                ADD             R1, R1, #3
ROM:0E205098                MOV             R0, R8
ROM:0E20509C                MOV             R2, R1
ROM:0E2050A0                MOV             R1, R0
ROM:0E2050A4                MOV             R0, #0x8301
ROM:0E2050A8                MOV             R0, R0,LSL#16
ROM:0E2050AC                ADD             R0, R0, #2
ROM:0E2050B0                SVC             0x1338
ROM:0E2050B4                SVC             0x1337
```

FIGURE 19 – Code 32 bits de `inc_pc`.

les retourne. Ce code lit en fait le registre `R15` de la VM. Les registres font 32 bits.

De la même manière, un appel SMC avec en paramètre `830010002`, un numéro entre 0 et 15 et une valeur, va écrire dans le registre en question (après l'avoir chiffré avec un tour de `AESD`).

Code 32 bits

Le code 32 bits commence en `0xe205000` et s'exécute en `EL0` sécurisé. Il communique avec le Secure-OS via des appels système (instruction `SVC`). Le Secure-OS expose ainsi 2 appels système différents, utilisés par ce code :

— 1338 : exécute l'instruction `SMC` avec les paramètres placés dans les registres `X0` à `X4`

— 1337 : rend la main au Secure-OS pour continuer l'exécution de la VM.

Opcodes de la VM

On réalise la même analyse pour toutes les branches prises par la fonction principale de la VM, et on en déduit la sémantique des instructions suivante : Les instructions sont sur 24 bits, numérotés de 0 à 23, découpés comme suit :

— Bits 20 - 23 : opcode
— Bits 18 - 19 : flags
— Bits 14 - 17 : param1
— Bits 10 - 13 : param2 (pas toujours présent)
— Bits 0 - 9 si param2 est présent (ou 0 - 13 sinon) : imm, valeur immédiate

Et la sémantique :

```
switch (opcode) :
        * 0xf : n'existe pas
        * 0xa -> halt, le programme retourne la valeur de Reg[0] : [X4] <- Reg[0]
        * 0xc -> 2e VM ?
        * 0xd -> Mem[0x9010008] <- 0
        * 0xe -> jump to imm (Reg[15] <- additional_data) if Mem[0x9010000] > 5

si flags == 0 :
        switch (opcode) :
                * 0 : mov : Reg[param1] <- Reg[param2]
                * 1 : dec : Reg[param1] <- Reg[param1] - 1
                * 2 : add : Reg[param2] <- Reg[param2] + Reg[param1] (83010012)
                * 3 : sub : Reg[param2] <- Reg[param2] - Reg[param1] (83010013)
                * 6 : xor : Reg[param2] <- Reg[param2] ^ Reg[param1] (83010016)
                * 11 : swap 2 lower bytes of Reg[param1]
                * autre : VM crash
si flags == 1 :
        si opcode == 0 : ldr : Reg[param1] <- Mem[Reg[param2]]
        sinon : VM crash

si flags == 2 :
        si opcode == 0 : str : Reg[param2] <- Mem[Reg[param1]]
        sinon : VM crash

si flags == 3 :
        switch (opcode)
                * 0 : mov : Reg[param1] <- additional_data
                * 2 : add : Reg[param1] <- Reg[param1] + additional_data
                * 3 : sub : Reg[param1] <- Reg[param1] - additional_data
                * 4 : lsl : Reg[param1] <- Reg[param1] << additional_data
                * 5 : lsr : Reg[param1] <- Reg[param1] >> additional_data
                * 7 : and : Reg[param1] <- Reg[param1] & additional_data
                * 6 : xor : Reg[param1] <- Reg[param1] ^ additional_data
                * 8 : jmp : Reg[15] <- additional_data
                * 9 : jnz : Reg[15] <- additional_data if Reg[param1] != 0
                * autre : VM crash
```

Cette sémantique est parfois implémentée dans le code 32 bits, parfois dans le Secure-OS, parfois dans le secure Monitor, parfois dans plusieurs à la fois, dépendant de l'instruction.

On note la présence de 2 instructions spéciales, qui lisent et écrivent en `0x9010000` et `0x9010008` de la machine physique (pas la VM).

Dump du bytecode

Au début de la fonction `0xe2005a4` on dispose d'une instruction en clair dans `X1`. (la trace nous dit également qu'on est dans l'appel `ioctl` 4). En remontant la trace en suivant la provenance de `X1`, on trouve l'endroit où l'instruction est chargée depuis la mémoire, qui se trouve dans l'appel `ioctl` 3. La mémoire de la VM est elle aussi chiffrée, cette fois-ci avec 4 tours de l'algorithme `SM4`[7], qui prend 2 clés de 16 octets présentes dans la mémoire (respectivement en `0xe204048` et `0xe204058`). C'est un chiffremnt par blocs sur des blocs de 16 octets : quand les 3 octets de l'instruction à charger chevauchent 2 blocs, les 2 blocs sont déchiffrés.

7. `https://en.wikipedia.org/wiki/SM4_(cipher)`

On en profite aussi pour déchiffrer complètement la mémoire de la VM, qui contient entre autres les `0x101010` octets copiés depuis le binaire en utilisateur lors de l'appel `ioctl()` 1. Le code `sm4_decrypt.c` fait cela, en reprenant l'implémentation logicielle des instructions `SM4` présente dans le code de `qemu`. Il prend en entrée le fichier `mem_first_round.bin` qui est un dump du blob de `0x101010` octets trouvé dans le binaire en espace utilisateur.

On retrouve ainsi le bytecode de la VM en entier, qui commence à l'addresse `0` de la mémoire de la VM.

A ce stade on dispose des addresses et des instructions, on peut donc reconstituer le CFG et les basic blocs du bytecode de la VM. Plutôt que d'écrire un désassembleur, on utilise le script `gdb` suivant, qui génère une trace, qu'on passe dans notre programme qui reconstruit le CFG :

```
b *0x0

target remote localhost:1234
set $current_call = 0

b *0xe030000
commands
        set step-mode on
        # sstic_open
        #b *0xffff000008590000
        # ioctl call_1
        b *0xffff000008590198
        commands
                set $current_call = 1
                continue
        end
        # ioctl call_2
        b *0xffff000008590518
        commands
                set $current_call = 2
                continue
        end
        # ioctl call_3
        b *0xffff00000859022c
        commands
                set $current_call = 3
                continue
        end
        # ioctl call_4
        b *0xffff000008590414
        commands
                set $current_call = 4
                continue
        end

        # fetch instruction
        b *0xe200ed0
        commands
                if $current_call == 3
                        p/x $w21
                end
                continue
        end

        # execute instruction
        b *0xe2005a4
        commands
                if $current_call == 4
                        p/x $w0
                end
                continue
        end
        continue
end
continue
```

5.6 Analyse du bytecode

Le bytecode se compose de 2 parties : une transformation de son entrée (qui est lue dans la mémoire de la VM à l'emplacement où l'appel `ioctl()` 2 a écrit le flag de l'utilisateur, et

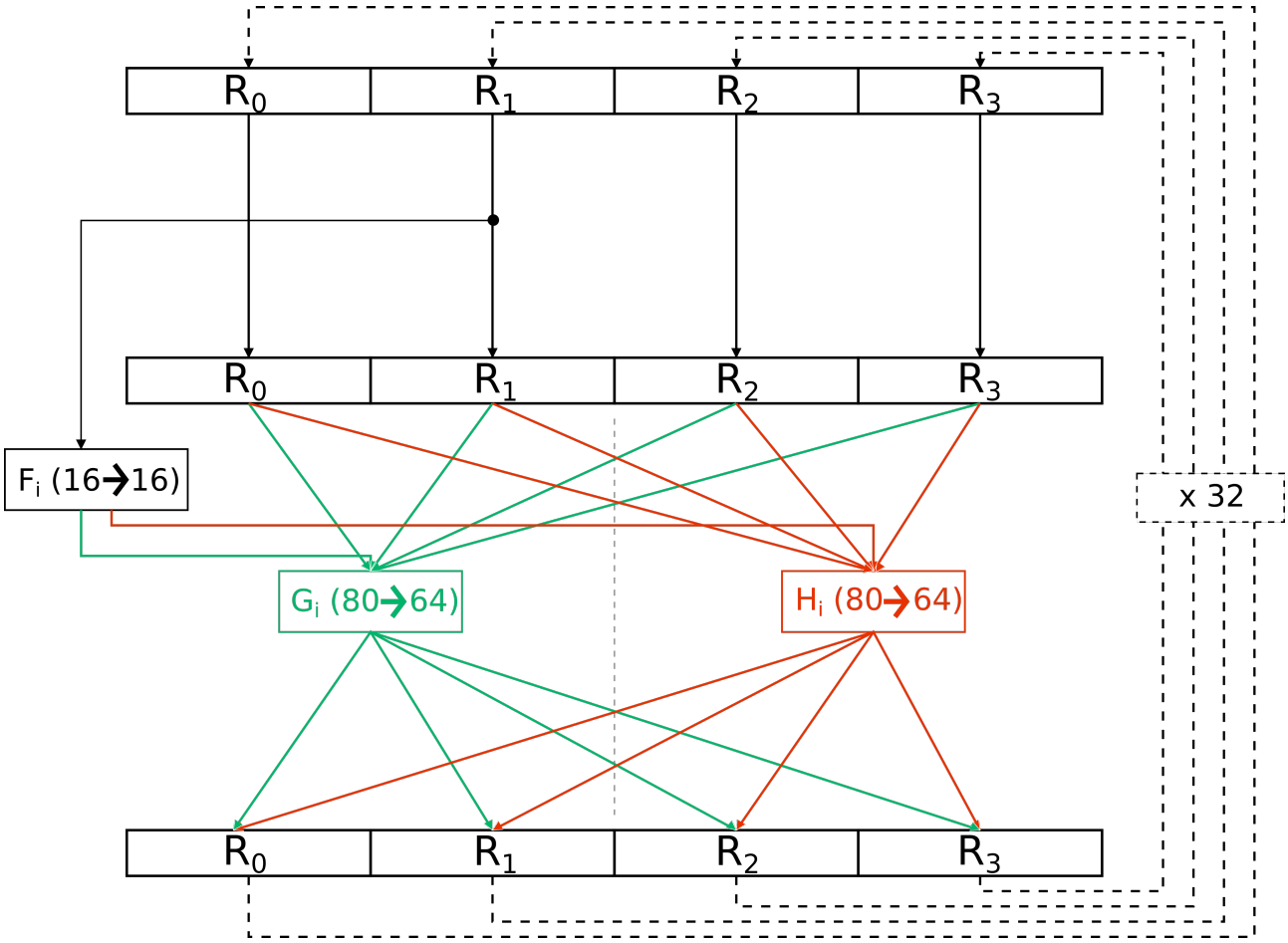

FIGURE 20 – Fonctions F_i, G_i et H_i dans la boucle à 32 tours

qui est donc ce dernier), puis une comparaison à une constante. La constante est le rot13 de la chaîne de caractères `ce.n.est.pas.si.facile@sstic.org`, constituée par des `mov` successifs de 2 octets.

La transformation (Figure 20 avant la comparaison opère sur des blocs de 8 octets. Une boucle externe parcourt les 4 blocs de 8 octets du flag de l'utilisateur. La transformation commence par charger les 8 octets dans les registres `R0` à `R4`, 2 octets par registre, les 16 bits de poids fort étant mis à 0. Puis un chronomètre est initialisé à 0 secondes, et on entre dans une boucle de 32 tours où est exécutée une fonction F_i paramétrée par le compteur de boucle, qui prend R_1 en argument. Cette fonction effectue des lookups mémoire, mais cette fois elle est inversible (pour le savoir on remplit, pour chaque valeur de i, une table des 2^{16} valeurs possibles, et on s'assure que chaque valeur de l'espace d'arrivée est atteinte. Est ensuite appliquée, au choix, une fonction F_i ou G_i, aussi paramétrée par le compteur de boucle. Ces fonctions ne sont pas inversibles car prennent R_0, R_1, R_2, R_3 et $F_i(R_1)$ en entrée (soit 80 bits) et ont 64 bits en sortie, mais leur composée avec F l'est. Les F et G effectuent des XOR entre les registres, pas besoin de calculer les tables inverses, il existe une formule.

Les fonctions G_i et H_i effectuent également des comparaison avec le timer : s'il est supérieur à 5 secondes, ce qui arrive rapidement quand on instrumente le programme pour faire une trace, le programme prend une autre branche, renvoyant un résultat faux. Pour s'en sortie, on patch le code avant de commencer à réaliser la trace, pour que le timeout soit fixé à `0xFF = 255` secondes, ce qui est suffisant. (Fichier `dump_vm_instructions_patched`).

Le fichier `reverse_algo.c` implémente les inverses de ces fonctions, de la boucle extérieure, et calcule le flag.

```
[user@machine safe_3]$ ./reverse_algo
Round 0, 0 values with no antecedent
Round 1, 0 values with no antecedent
Round 2, 0 values with no antecedent
Round 3, 0 values with no antecedent
Round 4, 0 values with no antecedent
Round 5, 0 values with no antecedent
Round 6, 0 values with no antecedent
Round 7, 0 values with no antecedent
Round 8, 0 values with no antecedent
Round 9, 0 values with no antecedent
Round 10, 0 values with no antecedent
Round 11, 0 values with no antecedent
Round 12, 0 values with no antecedent
Round 13, 0 values with no antecedent
Round 14, 0 values with no antecedent
Round 15, 0 values with no antecedent
Round 16, 0 values with no antecedent
Round 17, 0 values with no antecedent
Round 18, 0 values with no antecedent
Round 19, 0 values with no antecedent
Round 20, 0 values with no antecedent
Round 21, 0 values with no antecedent
Round 22, 0 values with no antecedent
Round 23, 0 values with no antecedent
Round 24, 0 values with no antecedent
Round 25, 0 values with no antecedent
Round 26, 0 values with no antecedent
Round 27, 0 values with no antecedent
Round 28, 0 values with no antecedent
Round 29, 0 values with no antecedent
Round 30, 0 values with no antecedent
Round 31, 0 values with no antecedent
Flag = acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e
```

6 Enfin des données en clair

Le fichier déchiffré dans `/root/safe_3` est un zip, qui semble corrompu. On extrait ce qui peut l'être avec `binwalk`, et on tombe sur une base de données de SMS, issue d'un téléphone sous Android. L'un des messages contient l'addresse de validation :

```
9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org
```

Ce qui conclut le challenge.

7 Conclusion

Ce challenge nous a fait voyager à travers de nombreux aspects de la sécurité informtique : cryptographie (cryptanalyse et attaques par canaux auxiliaires), rétro-ingéniérie logicielle, micro-architecture sécurisée, composants matériels sécurisés, et nous a fait découvrir de nouvelles manières de cacher des machines virtuelles. La communauté de la sécurité informatique française a aussi été sauvée !