

# Solution du challenge SSTIC 2019



Vincent DEHORS  
Ingénieur systèmes embarqués - Smile ECS

22 mai 2019

---

Ce document décrit une solution du **challenge** proposé à l'occasion de l'édition 2019 de la conférence annuelle de sécurité **SSTIC**. Le but de ce challenge est de retrouver une adresse mail en résolvant une série de problèmes techniques souvent par le biais de *reverse engineering*. Les quatre étapes qui composent ce challenge seront décrites en présentant l'approche, la méthode et les logiciels que j'ai utilisés pour les résoudre. A la fin de chacune des ces étapes, un *flag* intermédiaire est récupéré et permet de passer à l'étape suivante. Les différents fichiers utilisés pour la résolution du challenge sont présents dans l'archive contenant ce document.

---

# Table des matières

<b>1</b>	<b>Etape 1 : Attaque par canal auxiliaire</b>	<b>2</b>
1.1	Sujet du challenge . . . . .	2
1.2	Récupération de la clé RSA . . . . .	3
1.3	Démarrage de la VM . . . . .	7
<b>2</b>	<b>Etape 2 : Bruteforce sur 32bits</b>	<b>8</b>
2.1	Compréhension du niveau . . . . .	8
2.2	Analyse du schéma . . . . .	9
2.3	Bruteforce de la clé . . . . .	11
<b>3</b>	<b>Etape 3 : Du code caché en C++</b>	<b>14</b>
3.1	Le programme C++ . . . . .	14
3.2	Exécution dans un environnement sain . . . . .	15
3.3	Gestion des exceptions par la libgcc . . . . .	16
3.4	Méthode pour le reverse de la VM . . . . .	17
3.5	Partir de la fin et remonter . . . . .	19
3.6	Isoler les algorithmes . . . . .	20
3.7	Analyse du premier algorithme . . . . .	21
3.8	Analyse du deuxième algorithme . . . . .	22
3.9	Récupération de la clé . . . . .	24
<b>4</b>	<b>Etape 4 : De l'offuscation en EL3</b>	<b>27</b>
4.1	Analyse du binaire . . . . .	27
4.2	Analyse du driver . . . . .	28
4.3	Dump des données des deux premiers ioctls . . . . .	31
4.4	Méthodologie . . . . .	31
4.5	Stockage du gros buffer . . . . .	34
4.6	Analyse globale des SMCs TrustOS . . . . .	35
4.7	Analyse globale des SMCs OEM . . . . .	40
4.8	Autres protections . . . . .	41
4.9	Désassemblage de la VM . . . . .	42
4.10	Analyse du code de la VM . . . . .	44
4.11	Récupération de la clé . . . . .	45
<b>5</b>	<b>Etape 5 : Partition Android</b>	<b>48</b>
<b>6</b>	<b>Remerciements et conclusion</b>	<b>49</b>

# 1 Etape 1 : Attaque par canal auxiliaire

## 1.1 Sujet du challenge

Le challenge commence sur la page Web du SSTIC où un lien de téléchargement nous attend ainsi que l'énoncé suivant :

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 hypothèses. 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.

La première chose à faire est donc de récupérer l'archive en téléchargement et de l'extraire.

Listing 1.1 – Téléchargement et extraction de l'archive initiale

```
# Téléchargement
curl [url] --output virtual_phone.tar.gz
# Vérifiez le SHA
sha256sum virtual_phone.tar.gz
# Extraire l'archive
tar xzf virtual_phone.tar.gz
```

On se retrouve avec les fichiers suivants :

```
virtual_phone
├── README
├── power_consumption.npz
├── flash.bin
└── rom.bin
```

Le *README* nous apprend qu'il s'agit d'une image QEMU et nous donne la commande pour

lancer la VM :

Listing 1.2 – Lancement de la VM

```
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
```

Le premier démarrage échoue car le keystore n'existe pas (il est créé au premier lancement). Il faut donc relancer une deuxième fois la commande. Le bootloader (fichier rom.bin) nous demande alors une clé RSA.

## 1.2 Récupération de la clé RSA

En donnant une chaîne invalide au bootloader, celui nous informe que la clé encodée en hexadécimal doit faire 0x100 octets, ce qui correspond à 1024 bits. On peut ensuite tester une clé au bon format, comme avec `python -c "print 'A' * 256 .`

A ce stade, on pourrait commencer par analyser ce que fait le bootloader *SSTIC ARM Trusted Firmware*. Mais on sait déjà qu'on doit trouver une clé RSA de taille 1024 bits. L'énoncé du challenge nous apprend qu'il y a eu une mesure de consommation faite lors du démarrage. On peut supposer que le fichier de mesure nous donnera des informations sur cette clé.

Listing 1.3 – Extraction du fichier de mesure de consommation

```
mv power_consumption.npz power_consumption.zip
unzip power_consumption.zip
```

Le fichier extrait **arr\_0.npy** est un fichier numpy contenant une série de floats. J'ai donc commencé par chercher un outil pour afficher visuellement cette consommation au cours du temps. Pour le parsing du fichier, j'ai utilisé **numpy** et le module Python **matplotlib** pour l'affichage.

Listing 1.4 – Affichage des données en Python

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

def plot(y, ylabel, x, xlabel, title):
    fig, ax = plt.subplots()
    ax.plot(x, y)
    ax.set(xlabel=xlabel, ylabel=ylabel, title=title)
    ax.grid()

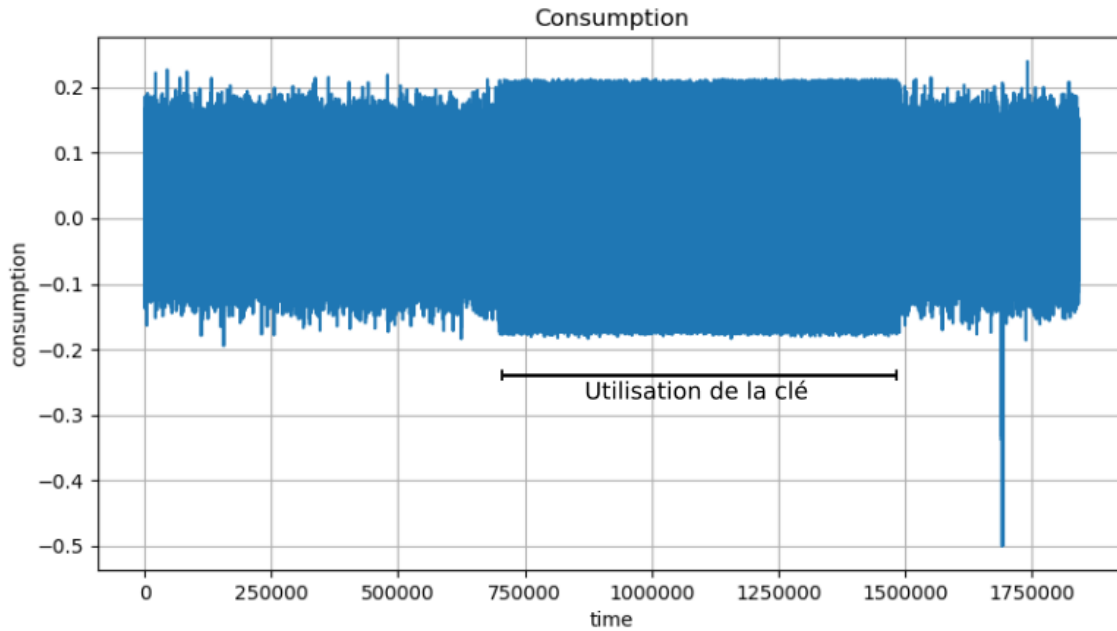
# Import measures
consumption = np.load("../files/arr_0.npy")

# Count samples for plotting
# Compute as 1Hz sample, we don't really care
```

```
t = np.arange(len(consumption))

plot(consumption, 'consumption', t, 'time', 'Consumption')
plt.show()
```

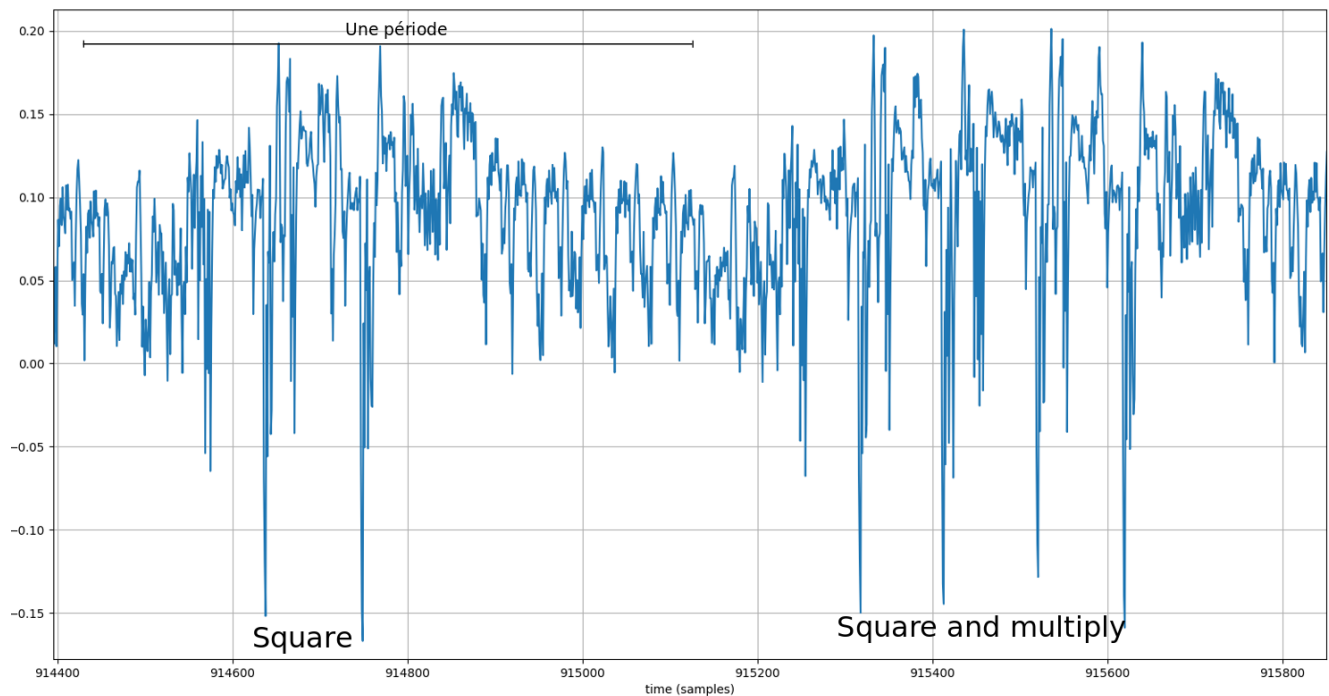
On obtient ainsi la figure suivante :



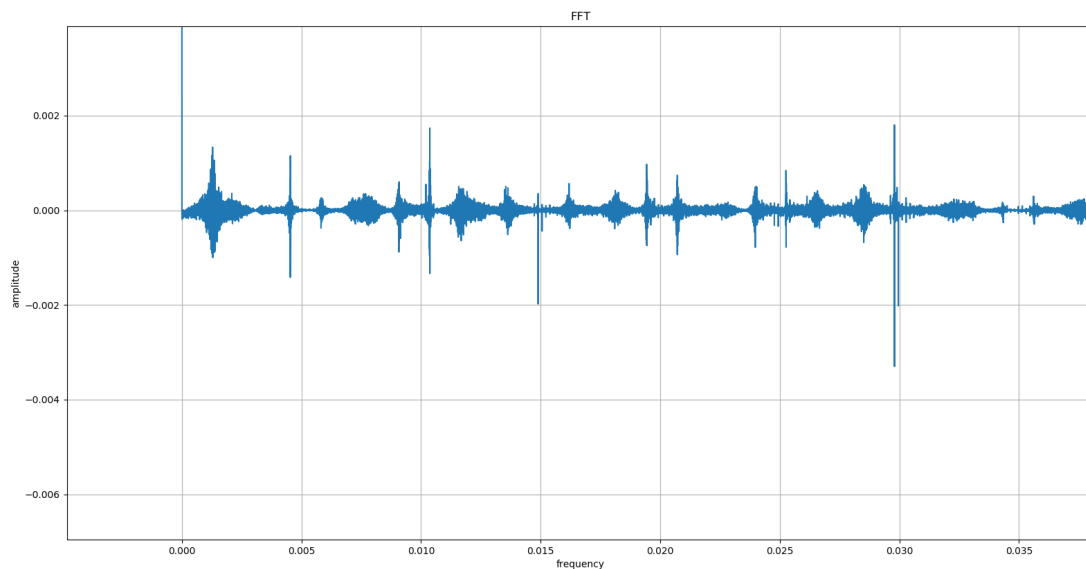
On voit nettement la zone où la consommation est symptomatique d'un algorithme long et répétitif, donc de la crypto. Si on zoom, on voit une répétition de pics de consommation.

Les algorithmes de chiffrement/signature et déchiffrement/vérification RSA sont des exponentielles modulaires. L'implémentation **logicielle** courante pour cette opération est **l'exponentiation rapide** (square and multiply). Cette implémentation logicielle sans contre-mesure est vulnérable à une simple analyse de consommation (SPA). Le document de Nicolas T. Courtois **sur les attaques par canaux auxiliaires** synthétise tout ce dont nous avons besoin de savoir pour résoudre le challenge.

L'idée pour extraire la clé est de compter les *pics* de consommation durant la phase où l'algorithme s'exécute afin de catégoriser chaque **période (itération)** : soit on fait une opération *square* (2 pics) lorsque le bit de la clé vaut 0 soit *square and multiply* (4 pics) lorsque le bit vaut 1. La capture suivante montre les deux types de périodes :



Le fichier fournit montre un signal **très** propre, ce qui n'est probablement jamais le cas en vrai. Avant de compter les pics, j'ai essayé d'améliorer la détections de ceux-ci en les amplifiant par rapport au reste du signal. Un *pic* est une variation importante d'amplitude sur un temps très court (deux ou trois échantillons). En terme de fréquence, même s'il n'est pas composé que de ça, un pic possède des hautes fréquences. Numpy permet facilement de calculer une FFT et de l'afficher, on obtient pour ce signal cette répartition fréquentielle (en proportion de la fréquence d'échantillonnage que j'ai fixée arbitrairement à 1) :



Pour rendre les pics plus visible, j'ai utilisé un filtre passe-bande pour enlever les fréquences

basses (celles qui sont inférieurs à un pourcent de la fréquence d'échantillonnage). Je n'ai pas essayé sans, mais vu qu'il y a peu de bruit et de signaux effaçant la mesure, un filtrage n'est peut-être pas nécessaire pour résoudre ce challenge. Le fait de filtrer la fréquence 0 permet également de recentrer le signal dans la mesure où l'offset constant disparaît.

Ensuite, un algorithme très simple parcourt les échantillons et détermine combien de pics sont présents durant une période. Pour identifier un pic, puisque la chute du pic est plus importante que sa montée, j'utilise un seuil minimum : si un échantillon passe en dessous, il s'agit d'un pic jusqu'à ce que le signal remonte au dessus du seuil. Pour délimiter les périodes, je compte le temps entre le pic courant et le précédent : si c'est supérieur à une durée minimum, il s'agit d'une nouvelle itération de l'algo.

Je m'attendais à devoir faire du tuning de l'algo ou à estimer la probabilité que chaque bit soit correct mais la clé obtenue avec cette méthode fut bonne du premier coup. Avant même de la vérifier, le fait d'obtenir une clé de 1024 bits est bon signe.

#### Listing 1.5 – Extraction de la clé

```
# Here we just count number of peaks during one iteration of fast exp. modul.
# Two peaks means this is just "square" -> Di = 0
# Four peaks means this is "square and multiply" -> Di = 1
#
# Note that negative peaks are more usable, so we use a negative threshold
threshold = -0.20
# Maximum time between two peak to understand that is a new iteration (D[i+1])
max_time_between_2peak = 250

D = list()
peaks = 0
below_threshold = False
samples_without_peaks = 0

for sample in consumption:
    # Increment peaks each time we see one
    if sample < threshold:
        samples_without_peaks = 0
        # New peak seen
        if not below_threshold:
            below_threshold = True
            peaks = peaks + 1
    else:
        samples_without_peaks = samples_without_peaks + 1
        below_threshold = False

# Store peaks seen between each fast exp. mod. iteration
if peaks > 0 and samples_without_peaks > max_time_between_2peak:
    if peaks == 4:
        D.append(1)
    elif peaks == 2:
        D.append(0)
    else:
        print('Error: wrong number of peaks seen')
    peaks = 0
```

Le code complet est disponible dans le fichier source **conso\_analyze.py**. Il extrait la clé à partir du fichier de consommation *arr\_0.npy* que vous devez placer à côté.

Listing 1.6 – Utilisation du script attaché pour générer le flag de l'étape 1

```
python conso_analyze.py
# Key is 23D87CDF97BB95ABE6273C384190C765F552AB86F6DE30A8DB74435C
# 95E6E3138F54AF689812D8F9359CF0F4D453A0C11EC68CE470216C09E74C894
# 7ADAF23E902415D61DDF2C0FFE459CBB40F7DE42BDB7CD14093100A570E8C29
# 819765E2D8D276F86471B52AC29AA2CE2BB72CD45006279E82BEC253AE9675F
# E45824F6001
```

Pour conclure, je pense que ce signal est bien trop propre pour avoir été réellement mesuré. Cette technique ne fonctionne que si on utilise une implémentation logicielle avec un matériel qui *leak* des informations sur ce qu'il fait vraiment. Cependant, dans le cas du challenge, on peut remarquer que le bootloader affiche des '+' et des '-' en fonction de l'opération faite. Or, l'impact sur la consommation de l'activité d'un bus externe (UART de debug par exemple) est sûrement plus visible que les instructions exécutées par un coeur ARM.

## 1.3 Démarrage de la VM

En utilisant la clé RSA précédemment extraite, on obtient le flag de validation :

```
KEYSTORE: Decrypting ...
...
Bravo, envoyez le flag SSTIC{a947d6980ccf7b87cb8d7c246} à l'adresse
challenge2019@sstic.org pour valider votre avancée
...
[ 0.000000] Booting Linux on physical CPU 0x0000000000 [0x411fd070]
[ 0.000000] Linux version 5.0.3 (david@polylaptop) #37 SMP PREEMPT Mon Mar
25 10:26:28 CET 2019
...
#####
# Welcome to SSTIC challenge #
#####

VM IP      : 192.168.200.200/24
USER       : root
PASSWORD   : sstic
```

Le bootloader démarre alors un Linux avec une version très récente (5.0.3). On se retrouve avec une console root dans la VM et celle-ci possède un serveur SSH.



## 2 Etape 2 : Bruteforce sur 32bits

### 2.1 Compréhension du niveau

Grâce à la console root, on peut explorer l'arborescence de la VM qui est un rootfs Linux classique. Un `ps` nous indique que rien ne tourne sur le système à part :

- Init : il s'agit du init de busybox (`ls -l /proc/1/exe`)
- Le logger système (syslogd et klogd)
- Le serveur SSH (dropbear)

Rien d'intéressant donc. Un `lsmod` nous indique qu'un module noyau **sstic.ko** est chargé mais celui-ci ne nous intéressera pas pour cette étape.

En revanche, le *HOME* de root contient plusieurs fichiers qui ne font pas partie d'un système classique, ils ont donc été ajoutés.

```
/root
├── tools
├── get_safe1_key.py
├── schematics.png
├── safe_01
│   └── .encrypted
├── safe_02
│   └── .encrypted
├── safe_03
│   └── .encrypted
```

On remarque le fichier python **get\_safe1\_key.py** et l'image **schematics.png** qui sont les deux fichiers à analyser pour résoudre ce deuxième challenge. On peut récupérer ces fichiers simplement en SSH :

Listing 2.1 – Récupération des fichiers de /root

```
vim ~/.ssh/config
#Host phonesstic
#       Hostname 127.0.0.1
#       User root
#       Port 5555
```

```
scp phonesttic:* .
```

La lecture du script Python nous apprend qu'il s'agit d'un outil utilisé conjointement avec des actions physiques sur des boutons du téléphone afin de générer une clé de chiffrement. On connaît le SHA256 de la clé mais on ne connaît pas les combinaisons d'appuis boutons qui ont permis de la générer. L'image à côté est le document de spécification du *secure element* avec lequel ce script Python est censé communiquer (qui a été laissé là par erreur...).

## 2.2 Analyse du schéma

L'image **schematics.png** est un schéma à base de portes logiques (AND, OR, XOR, NOT). Il implémente ce qui censé être fait par un *secure élément* qui accède à quatre boutons physique. Le logiciel définit deux entrées de 8bits ainsi qu'une opération codée sur 2 bit.

Le schéma peut sembler complexe de premier abord mais on peut le simplifier est découpant l'analyse en regardant comment certains signaux intermédiaires sont générés. J'ai choisi de distinguer trois parties :

- Le mixage des entrées

```
BM_n = (B_n-1 and BTN4) or (B_n and !BTN4)
```

Idem pour AM\_n mais avec BTN3 au lieu de BTN4

Donc  $BM = B$  ou  $BM = B \ll 1$  suivant BTN4, Idem pour AM en fonction de BTN3

Note : le décallage est cyclique (le bit de poids fort passe en poids faible)

- L'utilisation de l'OP

```
OPM0 = OP0 xor BTN1
```

```
OPM1 = OP1 xor BTN2
```

- La génération des sorties qui est un peu plus complexe

Note : "M" dans le nom signifie que c'est un signal intermédiaire (pour *Mixed*).

Concernant le mixage des sorties, j'en ai déduit le pseudo-code suivant :

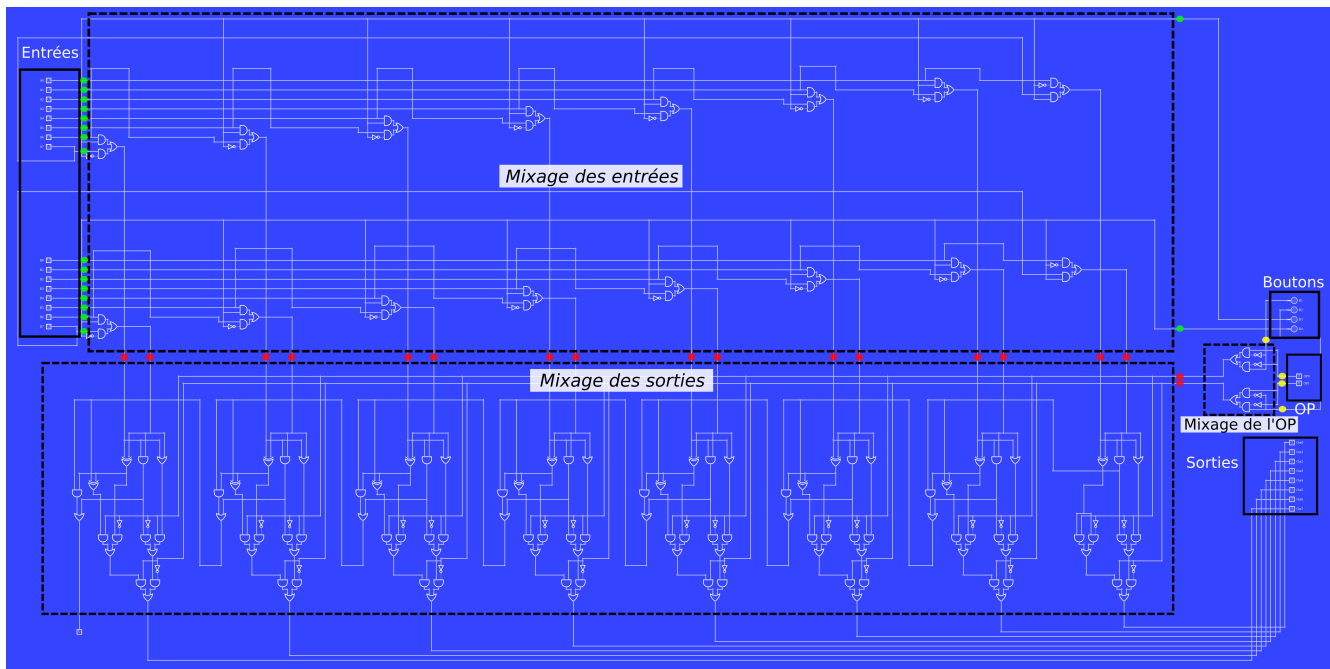
Listing 2.2 – Pseudo code pour le mixage des sorties

```
Si OPM1=1
  Si OPM0=1
    OUTn = OutTempN ^ AMn^BMn
    OutTempN = (OutTempN-1 . AMn^BMn) + AMn.BMn
    OutTemp0 = 0 (donc OUTn = AMn^BMn)
  Sinon
    OUTn = AMn^BMn
Sinon
  Si OPM0=1
    OUTn = AMn+BMn
  Sinon
    OUTn = AMn.BMn
```

Le cas qui ajoute de la complexité est la propagation des bits N-1 dans le cas où OPM0=1 et

OPM1=1.

Voici les différentes zones en question. J'ai mis des pastilles de couleurs pour montrer les **entrées** de ces trois parties.



Le code est donc assez simple, je l'ai réécrit en Python pour remplacer l'utilisation du *secure element* :

Listing 2.3 – Schéma implémenté en Python

```
# Mixage des entrées
def mix_input(inpt, btnX):
    if not btnX:
        return inpt
    else:
        return ((inpt << 1) & 0xff) | ((inpt & 0x80) >> 7)

# Mixage de l'OP
def mix_op(op, btn1, btn2):
    return ((op & 1) ^ btn1) | ((op & 2) ^ (btn2 << 1))

# Mixage des sorties
def mix_out(a, b, op):
    mask=0
    outTmpPrev=0

    if op == 0:
        return (a & b)
    if op == 1:
        return (a | b)
    if op == 2:
        return (a ^ b)
    # Otherwise op == 3
```

```

for i in range(8):
    maskn = (1 << i)
    an = (a & maskn)
    bn = (b & maskn)
    outTmpPrev = (outTmpPrev << 1)
    mask = mask | outTmpPrev
    outTmpPrev = (outTmpPrev & (an ^ bn)) | (an & bn)
return (a ^ b ^ mask)

# Réimplémentation du code du secure element
def secure_device(a,b,op):
    out = 0

    btn1 = 1 if (btn & 1) else 0
    btn2 = 1 if (btn & 2) else 0
    btn3 = 1 if (btn & 4) else 0
    btn4 = 1 if (btn & 8) else 0

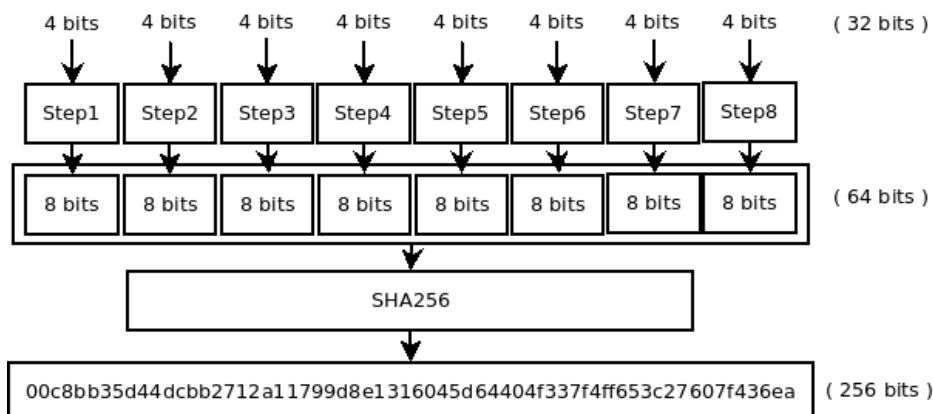
    op = mix_op(op, btn1, btn2)
    a = mix_input(a, btn3)
    b = mix_input(b, btn4)
    out = mix_out(a, b, op)

    return out

```

## 2.3 Bruteforce de la clé

La fonction `secure_device(a,b,op)` est appelée par huit fonctions `step1()` à `step8()` avec une combinaison différente des boutons pour chacune. Chaque étape effectue plusieurs appels au *secure element* avec des valeurs hard-codées pour les entrées et les deux bits d'opération. Etant donnée qu'on a précédemment implémenter le code manquant, nous sommes capable de générer la sortie (8bits) des ces fonctions à partir d'une configuration des boutons (4bits). L'ensemble d'entrée est donc 4bits secrets répétés 8 fois (car 8 étapes), soit  $\text{pow}(16, 8) = 4\,294\,967\,296$  possibilités. Pour tester qu'une de ces possibilité est correcte, il suffit de calculer son condensat SHA256 et de vérifier qu'il s'agit de celui qui est hard-codé dans le fichier Python.



J'ai effectué le bruteforce de la clé en deux étapes. Tout d'abord, puisque les fonctions `stepN()` sont indépendantes entre elles, j'ai calculé les 16 résultats possibles à partir des 16 combinaisons de boutons (4bits). Vous pouvez trouver le code qui le fait dans le fichier `generate_dict.py` attaché. J'obtiens alors les possibilités suivantes, `pX` étant les résultats possibles de `stepX()` :

Listing 2.4 – Valeurs utilisées pour le bruteforce

```
p1=(0xd7,0x19,0xdf,0x40,0xaf,0x62,0xbf,0x81,
    0x47,0x39,0xc7,0xc2,0x90,0x72,0x8f,0x89)
p2=(0x29,0xd1,0x08,0xdb,0x52,0xa4,0x10,0xb5,
    0x32,0xd9,0x90,0xda,0xf3,0xb4,0x20,0xf5)
p3=(0xed,0x8d,0xd9,0xdf,0xdb,0x1c,0xb0,0xbf,
    0xbd,0x4f,0xdc,0xdd,0x7b,0x9f,0x01,0xbb)
p4=(0xaa,0x38,0xa9,0x28,0x55,0x71,0x53,0x52,
    0x1a,0x4e,0xad,0x00,0x35,0x9c,0x5b,0x40)
p5=(0xfd,0x00,0xaf,0x47,0xff,0x01,0x60,0x8e,
    0xed,0x41,0xde,0xd4,0xcb,0x82,0xbd,0xa9)
p6=(0x31,0xb2,0xf6,0x64,0x4e,0x65,0xed,0xc8,
    0x3d,0xcf,0xee,0xe0,0x1b,0x9d,0xdd,0xc1)
p7=(0xdd,0x54,0x4d,0x4f,0xbb,0xa9,0x9a,0x9e,
    0x60,0x86,0x05,0xdf,0xca,0xe8,0x0a,0xbf)
p8=(0x83,0x54,0x78,0xfb,0x07,0xaa,0xf0,0xf3,
    0x97,0x85,0xd8,0xfc,0x4d,0x48,0xb0,0xfd)
```

Ensuite, il suffit de bruteforcer en calculant le hash de la concaténation de chaque combinaison :

Listing 2.5 – Test d'une combinaison

```
key = bytearray([s1,s2,s3,s4,s5,s6,s7,s8])
h = hashlib.sha256(key).hexdigest()
if h == '00c8bb35d44dcbb2712a11799d8e1316045d64404f337f4ff653c27607f436ea':
    print('Found key : %02x%02x%02x%02x%02x%02x%02x%02x' \
          % (s1,s2,s3,s4,s5,s6,s7,s8))
    sys.exit(0)
```

Le code est disponible dans le fichier `brute_force.py` attaché. Pour paralléliser sur les différents coeurs de mon laptop, j'ai passé le choix de l'index de `p1` (0 à 15) en argument au script.

Listing 2.6 – Récupération de la clé

```
for i in $(seq 0 15); do ./brute_force.py $i > log.$i & echo $i; done
# Au bout d'un moment (<15min sur mon laptop)
grep "Found" log.*
# log.14:Found key : 8fa4dfa9d4edbbf0
```

En conclusion, la résolution de ce challenge par brute-force a avant tout été possible car **l'espace d'entrée de l'algorithme est faible** (4 boutons répétés 8 fois) et qu'il y a un moyen de **tester** si une clé est bonne **rapidement**.

Avec cette clé 32bits, il suffit d'exécuter la partie du script Python pour dériver une clé AES, et utiliser l'outil fourni pour déchiffrer le conteneur suivant :

### Listing 2.7 – Validation de l'étape 2

```
#[i] Vous pouvez sauvegarder cette clef en utilisant /root/tools/add_key.py key
/root/tools/add_key.py 5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee
9bad427a8a
#[+] Key with key_id 00000002 ok
#[+] Key added into keystore
#[+] Envoyez le flag SSTIC{5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00
cdee9bad427a8a} à l'adresse challenge2019@sstic.org pour valider votre avancée
#[+] Container /root/safe_01/.encrypted decrypted to
/root/safe_01/decrypted_file
```

A la fin de cette étape, nous obtenons un fichier qui a été déchiffré avec notre clé : */root/safe-01/decrypted\_file*.

## 3 Etape 3 : Du code caché en C++

### 3.1 Le programme C++

Le fichier déchiffré lors du niveau précédent est un fichier ELF ARM64 lié dynamiquement et sans les symboles.

```
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]=5b5be1337d13c986d0e21441d771a36e41a34d17, stripped
```

J’ai utilisé [Ghidra](#) pour l’analyser. Le code semble très simple, si on avait les sources, ça donnerait quelque chose comme ça :

Listing 3.1 – Code C++ du programme

```
int main(int argc, char** argv){
    try {
        ExceptionStr e = argv[1];
        throw e;
    } catch(ExceptionStr e) {
        if (strcmp("SSTIC{congolexicomatisation}", e.str) == 0) {
            puts("That's the correct flag\n");
        } else {
            puts("Not good\n");
        }
    }
}
```

Ghidra calcule automatiquement l’adresse du `catch` en fonction des [Exception Frames](#) stockées dans les sections `.eh_frame` et `.eh_framehdr` de l’ELF.

A ce moment, je n’ai rien constaté d’anormal dans ce binaire. Le flag me paraît donc être logiquement “SSTIC{congolexicomatisation}” que je teste avec plein d’espoir :

```
./decrypted_file SSTIC{congolexicomatisation}
# Not good
```

Il y a donc un comportement dans l’exécution qui est anormal. Etant donné qu’il y a une

certaines formes d'obfuscation, je me suis d'abord dit qu'il ne fallait **rien** négliger : chaque constituant de la VM a pu être modifié pour impacter le fonctionnement de ce programme : le bootloader, le noyau Linux, **les bibliothèques** du système. J'ai soupçonné principalement la libstdc++ d'avoir été modifiée pour altérer la gestion des exceptions C++. De même, je n'ai pas fait confiance aux outils de debug sur la VM.

## 3.2 Exécution dans un environnement sain

On a un programme ARM64 compilé où l'analyse *offline* ne correspond pas au comportement *runtime*. J'ai donc cherché à exécuter le programme dans un environnement sain afin d'isoler l'élément qui modifie l'exécution de celui-ci. Lorsqu'on a un système basé sur Linux, il faut d'abord déterminer sa **distribution**. Pour cela, on peut se baser sur le type de services/outils présents sur la cible, sur les différentes versions des logiciels, leur système de paquets, etc. Pour les systèmes embarqués à base Linux (non Android), on retrouve très souvent Yocto et Buildroot. Le fichier `/etc/os-release` nous apprend qu'il s'agit bien d'un **Buildroot** en version 2018.11.

L'avantage de Buildroot est qu'il est très simple à prendre en main. Je décide donc de générer un **rootfs** pour l'architecture ARM 64bits (aka. AARCH64) compatible avec QEMU (machine "virt") en prenant la même version. Grâce au **menuconfig**, j'ajoute des outils de debug ainsi que le support du C++. J'ai attaché mon fichier de configuration (`cp buildroot_config buildroot/.config`) pour exemple.

```
git clone git://git.buildroot.net/buildroot
cd buildroot
git checkout 2018.11.x # J'ai réellement utilisé master
make qemu_aarch64_virt_defconfig
make menuconfig # Utiliser glibc à la place de uclibc + activer C++
# Le fichier de configuration généré est .config
make
```

À la fin du build, on a un **sysroot** dans le dossier *output/target*. QEMU fournit un mode d'exécution **user** pour AARCH64 qui est très pratique et qui fonctionne avec l'utilitaire **chroot** de ma distribution (Debian). Grâce à cette méthode, on peut exécuter un binaire dans un sysroot d'une architecture étrangère sans avoir besoin de lancer une vraie VM.

Listing 3.2 – Exécution du binaire dans un environnement sain

```
apt install qemu-user-static
cp -a buildroot/output/target chroot-aarch64
cp /usr/bin/qemu-arm-static chroot-aarch64/usr/bin
# Oui, il y a bien un binaire x86_64 dans votre chroot arm64 maintenant...
cp decrypted_file chroot-aarch64/root/
chroot chroot-aarch64 /root/decrypted_file SSTIC{congoloxicomatisation}
# Not good
```

Bonne nouvelle : on peut exécuter ce programme dans notre environnement. Mauvaise nouvelle : le programme se comporte toujours anormalement ! À ce stade, je me suis dit qu'il y avait *un truc*



que je ne voyais pas dans le binaire.

Avec un Buildroot déjà compilé, rajouter des outils de debug prend peu de temps (gdb, strace, etc...). J'ai d'abord voulu regarder quelle chaîne de caractères était comparée dans le programme. Pour cela, j'ai simplement fait une librairie qui wrap la fonction `strcmp()` avec un affichage des arguments :

Listing 3.3 – Wrap de la fonction `strcmp()`

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
#include <string.h>

static int (*real_strcmp)(const char *s1, const char *s2) = NULL;

int strcmp(const char *s1, const char *s2)
{
    printf("compare %s with %s\n", s1, s2);
    real_strcmp = dlsym(RTLD_NEXT, "strcmp");
    return real_strcmp(s1, s2);
}
```

Pour l'utiliser, il suffit de cross-compiler le fichier et de remplir `LD_PRELOAD`. Pour la cross-compilation, il suffit d'utiliser la toolchain de Buildroot.

Listing 3.4 – Tracing du `strcmp()`

```
buildroot/output/host/usr/bin/aarch64-buildroot-linux-gnu-gcc -fPIC -shared -o
    strcmp.so strcmp_wrapper.c -ldl
cp strcmp.so chroot-aarch64/root/
chroot chroot-aarch64 sh -c "LD_PRELOAD=/root/strcmp.so /root/decrypted_file
    SSTIC{congolexicomatisation}"
# compare St5ctypeIcE with NSt6locale5facetE -> Fait par une librairie
# Not good
```

La fonction `strcmp()` n'est pas appelée ! J'ai aussi fait le même genre de vérification avec GDB. Si on regarde le code, la fonction devrait être appelée si on exécute le `puts()` qui affiche "Not good". Lors de mon analyse du binaire, j'avais fait confiance à Ghidra pour déterminer l'adresse du code du `catch()` lors du déclenchement d'une exception mais je commence à remettre en doute cette information.

### 3.3 Gestion des exceptions par la libgcc

En remettant en question la manière dont les exceptions sont gérées dans le binaire, je me suis documenté sur le sujet que je ne connaissais pas du tout. Le document [Exception Handling ABI for the ARM® Architecture](#) contient beaucoup d'informations qui m'ont aidé à comprendre comment une exception C++ fonctionne.

J'ai appris qu'une partie de la gestion des exceptions est gérée par des structures et des instructions indépendantes du langage C++. Ces méta-données se trouvent dans les `.eh_frame` et sont générées par GCC lors de la compilation. Lorsqu'une exception survient, ces méta-données sont interprétées par la fonction *personality* définie. Celle-ci utilise la `libgcc` pour déterminer les actions à faire afin de retomber dans le code du `catch`. Cet [article de blog](#) décrit précisément ce qu'il se passe, pas à pas. Enfin, le langage utilisé pour décrire la gestion des exceptions est le **DWARF**.

Ok, mais je ne vois toujours rien d'anormal dans ces méta-données affichée dans Ghidra. J'ai alors chercher d'autres outils pour afficher plus d'informations. Ayman Khamouma (ak42), que je remercie, a eu la gentillesse de me pointer **readelf** et de m'expliquer comment la `libgcc` interprète les instructions en runtime. L'outil **readelf** permet d'afficher les instructions DWARF :

Listing 3.5 – Dump du code DWARF avec `readelf`

```
readelf --debug-dump=frames decrypted_file
...
00000090 0000000000000001c 00000094 FDE cie=00000000 pc=0000000000402e34
..0000000000402e68
DW_CFA_advance_loc: 1 to 0000000000402e35
DW_CFA_def_cfa_offset: 32
DW_CFA_offset: r29 (x29) at cfa-32
DW_CFA_offset: r30 (x30) at cfa-24
DW_CFA_val_expression: r28 (x28) (DW_OP_skip: -12222) <- ???
DW_CFA_nop
DW_CFA_nop
...
```

Dans la *frame* qui nous intéresse, on voit une instruction suspecte (`DW_OP_skip`) qui n'est présente que pour cette exception. D'après la documentation DWARF, il s'agit d'une instruction "unconditional branch", l'équivalent d'un **goto**. C'est à ce moment que j'ai compris que le code qui altère le comportement de notre exception se trouve au format DWARF dans le binaire et est interprété. J'ai alors trouvé quelques ressources très utiles sur Internet sur la construction de VM DWARF afin de cacher du code.

## 3.4 Méthode pour le reverse de la VM

Domage, `readelf` ne suit pas le branchement pour dumper le code qui nous interesse. J'ai localisé le code DWARF dans la section `.gnu.hash` de l'ELF. J'ai tenté d'utiliser des logiciels existants ou de modifier `readelf` pour extraire les instructions. Finalement, vu que le code est déjà interprété en runtime, je me suis dit qu'il était probablement plus facile de regarder le code de l'interpréteur. J'ai recherché ce code en particulier dans la `libgcc` : `grep -Ri "dw_op_skip" output/build/ | grep gcc`. Le paquet générant la `libgcc` est `host-gcc-final`. Même s'il s'agit d'un paquet "host-" (qui signifie architecture hôte), c'est bien lui qui construit la **libgcc.s.so** qui contient l'interpréteur DWARF. Plus précisément, il se trouve dans le fichier `libgcc/unwind-dw2.c`. Bien que l'indentation mélange tabulations et espaces, le code est plutôt simple et se résume à une seule fonction : `execute_stack_op()`.

Cette fonction nous montre l'implémentation de chaque instruction. Globalement, le langage est très simple et les instructions effectuent des opérations basiques à partir et à destination d'une stack de 64 éléments maximum.

Pour reverse le code DWARF interprété, j'avais deux options :

- Extraire le code de `execute_stack_op()` pour fabriquer un désassembleur DWARF dans un programme externe. Ainsi, j'aurais pu extraire le code de la VM à partir du binaire.
- Tracer l'exécution de la fonction en runtime pour afficher ce que fait l'interpréteur.

J'ai choisi la deuxième option. Avec du recul, je pense que la première solution aurait été plus simple dans la mesure où il y a beaucoup de boucles qui rendent l'exploitation des traces d'exécution plus compliquée. Cependant, l'intérêt de tracer ce qui est fait en runtime avec les valeurs manipulées permet de comprendre plus facilement comment les instructions fonctionnent.

Pour chaque implémentation d'instruction dans l'interpréteur, j'ai inséré un `printf()` qui affiche l'adresse et le nom de l'instruction ainsi que les données manipulées. J'ai attaché à ce pdf le code de mes modifications sur le fichier *unwind-dw2.c*.

L'exécution du programme avec la libgcc modifiée nous donne alors les informations tant attendue :

Listing 3.6 – Tracing de l'interpréteur DWARF

```
# Remplacer le fichier unwind-dw2.c
cd buildroot
make host-gcc-final-rebuild
cp output/build/host-gcc-final-7.4.0/build/aarch64-buildroot-linux-gnu/libgcc/
  libgcc_s.so [...]chroot-aarch64/lib64/libgcc_s.so
chroot chroot-aarch64 /root/decrypted_file SSTIC{congolexicomatisation}
# 0x403214 : DW_OP_skip (47) offset=-12222 [nopush]
# 0x400259 : DW_OP_regxx (111) result = 0x4000800500
# 0x40025a : DW_OP_const1u (8) result = 0xa8
# ...
```

La première instruction est bien le `DW_OP_skip` qu'affichait `readelf`. Avant d'analyser les traces, j'ai calculé le nombre d'instruction exécutées : **5.87 million** d'instructions exécutées dont 2631 différentes.

Ici commencent les choses compliquées. J'aurais sûrement dû analyser les 2631 lignes de code plutôt que d'essayer de comprendre les 5.87 millions traces d'exécution... J'ai ajouté une nouvelle information pour m'aider à comprendre l'algorithme : pour chaque instruction, j'affiche également le contenu de la stack.

En analysant les traces, j'ai d'abord vu que l'ensemble du code était exécuté deux fois par GCC, j'ai donc simplement coupé les traces en deux pour ensuite enlever la première partie (les deux exécutions sont identiques). Plus que 2.8 million...

Ensuite, j'ai effectué plusieurs tests dans le but de mieux comprendre ce qu'il se passe :

1. Analyser la fin pour comprendre comment afficher "That's the correct key" plutôt que "Not

good”.

2. Délimiter les différents algorithmes utilisés.
3. Faire varier l’entrée et comparer les traces d’exécution.

Le dernier point m’a montré que le nombre d’instructions exécutées varie légèrement en fonction de l’entrée. J’ai donc essayé de déterminer s’il y avait un *oracle* en faisant varier l’input et en tentant de minimiser ou de maximiser le nombre d’instructions exécutées. Au final, ce fut une mauvaise piste.

## 3.5 Partir de la fin et remonter

S’il y a bien une chose facile à comprendre dans les traces, c’est la fin.

Listing 3.7 – Fin de la trace

```
0x400286 : DW_OP_const8u (14) result = 0x658302a68e8e1c24
0x40028f : DW_OP_xor (39) first=0x658302a68e8e1c24, second=FINAL0, result=tmp0
0x400290 : DW_OP_swap (22) [nopush]
0x400291 : DW_OP_const8u (14) result = 0xdc7564f1612e5347
0x40029a : DW_OP_xor (39) first=0xdc7564f1612e5347, second=FINAL1, result=tmp1
0x40029b : DW_OP_plus (34) first=tmp1, second=tmp0, result=tmp2
0x40029c : DW_OP_swap (22) [nopush]
0x40029d : DW_OP_const8u (14) result = 0xd9c69b74a86ec613
0x4002a6 : DW_OP_xor (39) first=0xd9c69b74a86ec613, second=FINAL2, result=tmp3
0x4002a7 : DW_OP_plus (34) first=tmp3, second=tmp2, result=tmp4
0x4002a8 : DW_OP_swap (22) [nopush]
0x4002a9 : DW_OP_const8u (14) result = 0x65850b36e76aaed5
0x4002b2 : DW_OP_xor (39) first=0x65850b36e76aaed5, second=FINAL3, result=tmp5
0x4002b3 : DW_OP_plus (34) first=tmp5, second=tmp4, result=0xe8bbcf4700733138
0x4002b4 : DW_OP_bra (40) offset=12 cond=true newaddr=0x4002c2 [nopush]
```

Ce code est simple dès lors qu’on se familiarise avec les instructions DWARF. Pour cela, le plus rapide est de lire l’implémentation de chaque instruction et de garder à l’esprit que le résultat des instructions est empilé dans la stack de l’interpréteur. Dans cette portion du code, il y a **quatre** DW\_OP\_const8u dont chacun pousse sur la stack **un nombre 64bits hard-codés**. Il y a un XOR entre chacun de ces nombres et un élément de la stack qui varie suivant l’entrée du programme (nommé FINALx dans la trace ci-dessus).

Les valeurs FINALx générées avec mon flag de test préféré  $8*0xaa + 8*0xbb + 8*0xcc + 8*0xdd$  sont toujours les mêmes : `0x22b5ea7114e083db`, `0x0ce6d7e53aa88d1d`, `0xb0a1d6cee9c216f7`, `0x72b4c9f59ff8ef92`.

A la fin, il y a une instruction de branchement conditionnel qui déclenche le retour au code C++ au niveau du `puts("Not good")`. On comprend maintenant pourquoi notre fonction `strcmp()` n’était jamais appelée. Afin d’empêcher ce retour vers "Not good", il faut que l’ensemble des XOR, qui sont additionnés, valent 0. Pour résoudre cette étape, il faudra donc trouver la bonne entrée afin que les valeurs FINALx soit égales aux valeurs hard-codées.

Avec les valeurs FINALx, on peut alors remonter les traces, regarder comment elles sont calculées et essayer d'inverser le calcul.

## 3.6 Isoler les algorithmes

C'est probablement le point qui m'a posé le plus de problème. J'ai vite compris que l'ensemble était constitué de plusieurs algorithmes imbriqués dans des boucles mais isoler les morceaux d'algorithme n'a pas été évident. Pour cela, j'ai utilisés plusieurs méthodes conjointement :

- Identifier les **adresses** des instructions afin de voir lorsqu'elles se répètent
- Identifier les **index des boucles** visibles dans la stack
- Localiser les instructions de branchements

J'ai ainsi lancé un nombre incalculable de **grep** pour tenter d'isoler les algorithmes. En partant de la boucle la plus englobante, j'ai noté qu'il y avait quatre itérations et chacune utilise deux fois deux algorithmes à la suite avec des données différentes que j'ai nommés (pour être original...) : **shuffle** et **algo1** (les noms ne veulent rien dire du tout).

Par exemple, pour identifier la succession des deux algorithmes, je repère une instruction (début ou fin) de chaque algorithme, puis **grep** nous montre l'enchaînement ainsi que le numéro de ligne (pour avoir une idée de la taille) :

Listing 3.8 – Enchaînement des deux algorithmes

```
grep -n -e "^0x40030c" -e "^0x4003a1" dump_instr
S      40:0x40030c : DW_OP_litxx (52) result = 0x4
A1     351:0x4003a1 : DW_OP_bra (40) offset=3 cond=false newaddr=0x4003a3
S     367520:0x40030c : DW_OP_litxx (52) result = 0x4
A1    367831:0x4003a1 : DW_OP_bra (40) offset=3 cond=true newaddr=0x4003a6
S     734963:0x40030c : DW_OP_litxx (52) result = 0x4
A1    735274:0x4003a1 : DW_OP_bra (40) offset=3 cond=false newaddr=0x4003a3
...
```

Comme j'aime bien avoir une représentation de l'algorithme dans un langage que je connais, j'écris systématiquement ce que je comprends en code C. Voici ce que j'ai compris de la boucle principale :

Listing 3.9 – Boucle principale

```
int i;
uint64_t IV1;
uint64_t IV2;
uint64_t input_1 = 0xaaaaaaaaaaaaaaaa;
uint64_t input_2 = 0xbbbbbbbbbbbbbbbb;
uint64_t input_3 = 0xcccccccccccccccc;
uint64_t input_4 = 0xdddddddddddddddd;

for (i=0; i<4; i++) {
    IV1 = input_4;
```

```

    IV2 = input_3;

    shuffle(&IV1, &IV2);
    algo1(&IV1, &IV2, &input_2, &input_1);

    IV1 = input_2;
    IV2 = input_1;
    shuffle(&IV1, &IV2);
    algo1(&IV1, &IV2, &input_4, &input_3);
}

```

Bien sûr, avant de reverse les deux algorithmes (shuffle et algo1), je n'avais qu'une vague idée des entrées et des sorties de chacun. Identifier les sorties des algorithmes fut assez complexe. En effet, il faut pouvoir distinguer un élément en stack qui est le résultat d'un algo et un autre élément qui sera simplement supprimé plus tard. De plus, la stack a tendance à grossir au cours de l'exécution du programme, il y a donc un moment où les éléments en stack ne sont volontairement pas libérés.

## 3.7 Analyse du premier algorithme

Le premier sur lequel je me suis penché fut celui que j'ai appelé shuffle et qui porte très mal son nom. J'ai suivi la méthode suivante :

1. Identifier le début et la fin
2. Générer un jeu de données. Pour une trace, on a 4\*2 jeux de données qu'on peut utiliser pour tester si notre compréhension est correcte
3. Identifier les boucles internes
4. Comprendre une itération et la traduire en code

Le code est composé de plusieurs types d'instructions :

- Des opérations arithmétiques (booléennes ou non)
- Des branchements, principalement pour faire des boucles (DW\_OP\_bra, DW\_OP\_skip)
- Des **lectures mémoires** (DW\_OP\_deref, DW\_OP\_deref\_size)

Dans les deux algorithmes, des lectures mémoires sont faites pour utiliser des tableaux (lookup tables) stockés dans le binaire.

Pour extraire la première lookup table, j'ai récupéré la valeur à l'indice 0, puis j'ai dumpé les entiers situés à la suite dans le binaire :

Listing 3.10 – Extraction du premier tableau

```

cat decrypted_file | hexdump -v -e '/4 "0x%08x\n"' | grep -A 255 "0x5963b39b"
# 0x5963b39b
# 0x30f75add
# ...

```

En dehors de l'utilisation de ce tableau, l'algorithme utilise principalement des opérations arithmétiques au sein d'une boucle à quatre itérations. Il prend en entrée deux entiers de 64bits qu'il répartie sur quatre entiers de 32bits. Ce entiers sont modifiés dans la boucle à quatre itérations et le résultat est stocké dans la stack de la VM sous la forme de deux entiers de 64bits. Le code est le suivant :

Listing 3.11 – Réécriture en C du premier algo

```
uint32_t LUT[256] = { 0x5963b39b, 0x30f75add, ... };

void shuffle(uint64_t *in1, uint64_t *in2)
{
    int i=0;
    uint32_t tmp2;

    uint32_t MIXED1 = *in1 & 0xffffffff;
    uint32_t MIXED2 = (*in1 & 0xffffffff00000000) >> 32;
    uint32_t MIXED3 = *in2 & 0xffffffff;
    uint32_t MIXED4 = (*in2 & 0xffffffff00000000) >> 32;

    for (i=0; i<4; i++) {
        tmp2 = MIXED3 ^ (MIXED1 + MIXED4) & 0xffffffff;
        MIXED4 = MIXED1 & MIXED4;
        MIXED1 = (MIXED1 - tmp2) & 0xffffffff;
        MIXED3 = (tmp2 + LUT[MIXED2 & 0xff]) & 0xffffffff; // <- LUT
        MIXED2 = (MIXED2 >> 8) ^ MIXED3;
    }

    *in1 = ((uint64_t)MIXED2 << 32) | MIXED1;
    *in2 = ((uint64_t)MIXED4 << 32) | MIXED3;
}
```

A ce moment, j'ai fait l'erreur de vouloir retourner cet algorithme. L'opération `MIXED4 = MIXED1 & MIXED4` rend l'algorithme non-bijectif. Pour m'en persuader, j'ai développé le fichier `shuffle_resolve.py` utilisant Z3 qui ne résoud rien évidemment puisque pour une même sortie plusieurs entrées sont possibles.

Avec le jeu de valeurs (entrées/sorties), j'ai pu vérifier le bon fonctionnement de mon implémentation, ce qui est indispensable pour avancer sereinement. Il est également intéressant de noter que la première entrée de l'algorithme est (0xdddddddddddddddd, 0xcccccccccccccc) (la moitié de ma clé de test).

## 3.8 Analyse du deuxième algorithme

Le deuxième algorithme, que j'ai appelé **algo1** (logique), est plus compliqué car il possède plusieurs sous-algorithmes qui utilisent deux *lookup tables*.

J'ai appelé les sous-algorithme algo1\_n (n de 1 à 4). La boucle principale contient 15 itérations, ça se voit facilement en identifiant l'index dans la stack de la trace. L'algorithme prend quatres

entrées :

- Les deux premières entrées (in1 et in2) sont les résultats de l’algo shuffle décrit précédemment
- Les deux autres entrées sont modifiées et consistent également la sortie

Listing 3.12 – Boucle principale du deuxième algo.

```
void algo1(uint64_t *in1, uint64_t *in2, uint64_t *in3, uint64_t *in4)
{
    int i;
    uint64_t tmp1, tmp2, tmp4, tmp3;

    for (i=0; i<15; i++) {
        tmp1 = *in1; // Valeur initiale
        tmp2 = *in2; // Valeur initiale
        algo1_1(&tmp1, &tmp2, i);
        tmp3 = *in3;
        algo1_2(&tmp3);
        tmp4 = *in4;
        algo1_3(&tmp2, &tmp3, &tmp4);
        *in4 = tmp4;
        algo1_2(&tmp4);
        algo1_4(&tmp1, in3, &tmp4);
        *in3 = tmp4;
    }
}
```

Finalement, une fois la méthode de reverse assimilée, l’analyse de ces quatre sous-algorithmes fut rapide. Il suffit de comprendre les différentes instructions, la manière dont sont codées les boucles et comment les *lookup tables* sont utilisées. Le code de ces algorithmes est disponible dans le fichier inclut dans le PDF **lvl3.c**.

En dehors de leur taille volumineuse, ces sous-algorithmes sont tous bijectifs. L’algorithme algo1\_1 utilisait un peu de code inutile que j’ai pu facilement identifier dans les traces. La *mémoire* de la VM, en dehors des tableaux constants utilisés pour les *lookup tables*, se limite à la stack. Ainsi, si l’état de la stack revient à un même point que précédemment, le code est inutile. En l’occurrence, dans le code algo1\_1, vous pouvez voir ceci :

```
if (tmp2 & 0x80000000) {
    tmp7 = 0x60bf080f;
} else {
    tmp7 = 0x818f694a;
}
```

Or dans les traces, le code réel est beaucoup plus important mais une partie est inutile, je ne l’ai donc pas analysé.

Comme pour le premier algorithme, j’ai pu tester l’implémentation en utilisant un jeu de données préalablement extrait à coup de **grep**. Par exemple, voici le jeu de test que j’ai utilisé :

Listing 3.13 – Jeu de test pour l’algo1



```
Input de l'algo1
=====
```

IN4	IN3	IN2	IN1
0xaaaaaaaaaaaaaaaa	0xbbbbbbbbbbbbbbbb	0x00004008ac6a9cbb	0xacbde2d76f29e73e
0xcccccccccccccccc	0xdddddddddddddddd	0x00000000f2f3956a	0xf2253178b65b8b01
0x819fbfd091e71dfe	0xd69d8f4a0f478e6e	0x00400000758c4c49	0x7519950494f0d36a
0xa8680d07e25770d8	0x6a349959a6fc1fd8	0x000000003de76888	0x3d72cdc0933bab0c
0x6afa5ed098241fdc	0xe1244964050072d7	0x0000000040e219eba	0x0ed51ac02ba2016d
0xe964398c2bcc89ff	0xc59f7016e2bb9314	0x0000000082c6fd6d	0x82d2769f0bf0872c
0xbc508191b7beaf75	0xf6eda81e474b87ed	0x00000002049b4da11	0x49e9b712256e7b4e
0x681624a8c0bc77f3	0xf5e80642c004d267	0x006080002c09c7f9	0x2c68f34e60fb6a6c

```
Output de l'algo1
=====
```

0xd69d8f4a0f478e6e	0x819fbfd091e71dfe	
0x6a349959a6fc1fd8	0xa8680d07e25770d8	
0xe1244964050072d7	0x6afa5ed098241fdc	
0xc59f7016e2bb9314	0xe964398c2bcc89ff	
0xf6eda81e474b87ed	0xbc508191b7beaf75	
0xf5e80642c004d267	0x681624a8c0bc77f3	
0x22b5ea7114e083db	0x0ce6d7e53aa88d1d	<- FINAL1 et FINAL2
0xb0a1d6cee9c216f7	0x72b4c9f59ff8ef92	<- FINAL3 et FINAL4

### 3.9 Récupération de la clé

Une fois les deux algorithmes implémentés, j'ai pu assembler le tout. Ainsi, à partir de mon entrée de test, je peux vérifier et obtenir les valeurs finales qui sont comparées aux valeurs constantes.

Le fichier **lvl3.c** contient l'ensemble du code. Je l'ai ensuite modifié pour coder l'inverse de chaque composants de l'algo1, vu qu'ils sont réversibles. Pour cela, je l'ai fait *à la main* en retournant les boucles et les opérations. Au final, je me suis rendu compte que la plupart des fonctions que j'avais ainsi créées n'était pas utiles. Par exemple, dans l'algo1, les entrées 1 et 2 ne sont pas modifiées, elles ne changent pas d'une itération à l'autre. J'ai dû en revanche retourner les algo 1\_2 et 1\_3 pour qu'ils retournent l'entrée à partir de la sortie.

Le retournement de l'algo1 se traduit par :

Listing 3.14 – Retournement de l'algo1

```
void algo1_reversed(uint64_t *in1, uint64_t *in2, uint64_t *in3, uint64_t *in4)
{
    int i;
    uint64_t IV1, IV2, tmp4, tmp3, tmp5, tmp6;

    for (i=14; i>=0; i--) {
        IV1 = *in1;
        IV2 = *in2;
        algo1_1(&IV1, &IV2, i);
    }
}
```

```

    tmp4 = *in3;
    tmp5 = *in4;
    algo1_2(&tmp5);
    algo1_4_guess_in3(&IV1, in3, &tmp5, &tmp4); // Algo retourné

    tmp3 = *in3;
    tmp6 = *in4;
    algo1_2(&tmp3);
    algo1_3_reversed(&IV2, &tmp3, &tmp6); // Algo retourné
    *in4 = tmp6;
}
}

```

A partir de ce point, il ne s'agit plus que de retourner la boucle principale, en sachant qu'il n'est pas possible de retourner l'algo shuffle, il doit donc y avoir une astuce. Si on reprend l'itération de la boucle principale :

```

IV1 = input_4;
IV2 = input_3;

shuffle(&IV1, &IV2); // IVn sont modifiés
algo1(&IV1, &IV2, &input_2, &input_1); // input_n sont modifiés

IV1 = input_2;
IV2 = input_1;
shuffle(&IV1, &IV2); // IVs sont modifiés
algo1(&IV1, &IV2, &input_4, &input_3); // input_n sont modifiés

```

On remarque que le retournement du deuxième appel à `algo1()` nous donne `input_3` et `input_4`, qui sont les entrées du premier appel à `shuffle()`. Si on retourne la boucle en partant des valeurs de fin, on peut donc obtenir les entrées de l'algorithme problématique. Les entrées du deuxième appel à `shuffle()` sont quant à elles déjà fournies puisqu'il s'agit des dernières modifications sur `input_2` et `input_3`. La boucle retournée donne :

Listing 3.15 – Retournement de la boucle principale

```

for (i=0; i<4; i++) {
    IV1 = input_2;
    IV2 = input_1;
    shuffle(&IV1, &IV2);
    algo1_reversed(&IV1, &IV2, &input_4, &input_3);

    IV1 = input_4;
    IV2 = input_3;
    shuffle(&IV1, &IV2);
    algo1_reversed(&IV1, &IV2, &input_2, &input_1);
}

```

J'ai d'abord testé avec les valeurs finales obtenues avec la trace de ma clé de test. Puis, j'ai utilisé les valeurs constantes hard-codées pour obtenir de flag. Le programme **lvl3.c** génère le flag à partir des valeurs finales attendues :

### Listing 3.16 – Récupération du flag

```
gcc -o lvl3 lvl3.c
./lvl3
#...
#SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}
```

Comme pour l'étape précédente, on valide le flag avec l'utilitaire `add_key.py` :

```
# /root/tools/add_key.py SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}
[+] Key with key_id 00000003 ok
[+] Key added into keystore
[+] Envoyez le flag SSTIC{Dw4rf_VM_1s_co0l_isn_t_It} à l'adresse
    challenge2019@sstic.org pour valider votre avancée
[+] Container /root/safe_02/.encrypted decrypted to /root/safe_02/
    decrypted_file
[w] You must reboot in order to decrypt Secure OS
```

## 4 Etape 4 : De l’offuscation en EL3

### 4.1 Analyse du binaire

Le flag de l’étape précédente nous a déchiffré un nouveau fichier **decrypted\_file**, cette fois-ci dans le dossier safe02. Il s’agit également d’un binaire mais compilé en statique.

Pour commencer, je l’ai d’abord exécuté. Celui-ci requiert une clé de 32 octets encodée en hexa-décimal. Avec ma clé préférée, on obtient "Loose" :

Listing 4.1 – Première exécution du binaire

```
FLAG=AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDDD
time ./decrypted_file $FLAG
#Loose
#real      0m 1.73s
#user      0m 0.07s
#sys       0m 1.61s
```

Malheureusement, ce n’est pas la bonne clé. J’ai pris l’habitude d’utiliser la commande **time** car elle nous donne deux informations importantes. Tout d’abord, un programme qui prend 1.73s à s’exécuter effectue **beaucoup** de chose, ou possède des mécanismes d’attente (ou de synchronisation). A titre de comparaison, les 5.8M instructions DWARF de l’étape 3 ne prenaient que 240ms dans la VM. La deuxième chose qu’on apprend avec cette commande est la répartition entre le système (noyau Linux) et l’userspace. Or ici, on voit que la grande majorité du temps d’exécution se passe dans le noyau Linux et pas dans notre binaire ou dans une librairie.

J’ai pensé utiliser Ghidra pour comprendre le fonctionnement du binaire mais j’ai d’abord utilisé un approche *système* qui m’a suffi pour comprendre ce qu’il fallait faire. Plus tard, j’ai quand même ouvert le programme avec Ghidra pour vérifier que j’avais bien compris ce qu’attendait le programme.

En considérant le programme comme une boîte noire *userspace*, on peut facilement connaître les interfaces avec le noyau en traçant les appels systèmes avec l’utilitaire **strace**. J’ai utilisé l’option **-X raw** pour empêcher le programme d’essayer de trouver des informations supplémentaires (ex. le nom de l’ioctl). Le programme écrit sur la sortie d’erreur, il faut donc l’utiliser si on redirige la sortie vers un fichier.

## Listing 4.2 – Strace du programme

```
strace -X raw ./decrypted_file $FLAG 2>&1 | tee strace.log
#execve("./decrypted_file", ["/decrypted_file", "AAAAAAAAAAAAAAAABBBBBB"...
#openat(-100, "/dev/sstic", 0) = 3
#ioctl(3, 0xc0105300, 0xffffedcc8c18) = 0
#ioctl(3, 0xc0105301, 0xffffedcc8c18) = 0
#ioctl(3, 0xc0105302, 0) = 0
#ioctl(3, 0xc0105303, 0) = 0
#ioctl(3, 0xc0105302, 0) = 0
# ...
#ioctl(3, 0xc0105303, 0) = 65537
#write(1, "Loose\n", 6Loose
```

Le strace nous apprend que le binaire utilise un *special device* **/dev/sstic** et effectue une série de `ioctl()` qui retournent tous 0 sauf le dernier. Sous Linux, une grande partie des drivers ont une API basée sur ces *special devices* qui peuvent être de type "block" ou "character". Ils permettent d'implémenter des *syscalls* génériques sous Linux (open, read, write, close, ...) en utilisant comme nom de fichier un chemin référençant ce fichier spécial. Pour des **utilisations variées**, le syscall `ioctl` est souvent utilisé et il n'y a pas de convention quant à l'opération réellement effectuée : elle dépend de chaque driver. Pour que ce syscall soit utilisable dans une grande variété d'utilisation, les arguments sont assez génériques. Un `man ioctl` nous décrit ces arguments :

- Le descripteur de fichier vers le fichier spécial. Dans le cas de la trace, la valeur 3 correspond au descripteur retourné lors du `open()` de `/dev/sstic`.
- Un numéro spécial *request* qui indique au driver quelle opération il souhaite faire. Il n'y a pas spécialement de convention mais les numéros sont aujourd'hui générés via la macro `IOC()` qui peut donner quelques indications sur les arguments, notamment la taille et la direction (lecture ou écriture).
- Un argument décrivant la requête dont le type est généralement un pointeur et qui peut ne pas être utilisé.

En analysant l'enregistrement du strace, on distingue donc :

- Une requête `0xc0105300` qu'on abrégera en `ioctl n°1` et qui passe un pointeur.
- Une requête `0xc0105301` qu'on abrégera en `ioctl n°2` et qui passe également un pointeur.
- Puis un enchainement en boucle de (**9366** itérations) :
  - La requête `0xc0105302` qu'on abrégera en `ioctl n°3`.
  - La requête `0xc0105303` qu'on abrégera en `ioctl n°4`.
- La fin de la boucle finit lorsque l'`ioctl n°4` retourne autre chose que 0. Dans mon cas il retourne 65537.

## 4.2 Analyse du driver

Pour localiser un driver, il faut souvent faire de l'archéologie. Mais dans ce cas précis, j'avais déjà remarqué un driver **sstic** compilé en module et chargé à l'init (grâce à `lsmod`). Un `mount` nous apprend que le `/dev` est un système de fichier de type **devtmpfs**. Il s'agit d'un système de fichier

géré par le noyau qui contient les fichiers spéciaux créés par les drivers (contrairement à avant où c'était statique ou géré par udev). Ainsi, si c'est le module sstic qui a créé /dev/sstic et qu'on le décharge, le fichier devrait disparaître sauf si le driver n'appelle pas la fonction pour supprimer le device lorsqu'il est déchargé. Dans ce dernier cas, l'utilisation du device devrait alors faire planter le noyau puisqu'il n'y a pas de driver derrière /dev/sstic au moment du syscall. Pour faire de genre de test, j'utilise souvent **cat** qui se contente de faire un `open()` et un `read()`. Même si le `read()` n'est pas implémenté, le `open()` l'est toujours puisque c'est lui qui fournit le descripteur de fichier à utiliser pour les autres syscalls.

Listing 4.3 – Vérification du driver derrière /dev/sstic

```
cat /dev/sstic
#cat: read error: Invalid argument
rmmod sstic
cat /dev/sstic
#[ 505.471672] Unable to handle kernel paging request at ...
```

C'est donc bien ce module qui est responsable de la gestion du fichier /dev/sstic. L'outil générique pour obtenir des information sur un driver est *modinfo*. Il parse simplement les métadonnées d'un module, on peut utiliser celui de notre machine :

Listing 4.4 – Information sur le module

```
scp phonesstic:/lib/sstic.ko .
sudo modinfo sstic.ko
#filename:          sstic.ko
#alias:             platform:sstic
#license:           GPL
#description:       Driver for SSTIC Challenge : Communication between normal
                    world and Secure world
#author:            David BERARD
```

Vu qu'il est sous licence GPL, j'ai hésité à demander le code... On apprend également d'après la description qu'il communique avec le *secure world*.

Pour analyser ce driver, j'ai utilisé Ghidra en focalisant mon analyse sur la callback enregistrée pour gérer les ioctls. Le code n'est pas du tout offusqué et est très lisible. L'analyse fut très rapide car le driver fait peu de chose. J'ai mis le code du handler d'ioctl dans le fichier joint **ioctl\_driver.c**. Il s'agit simplement d'un driver passe-plat vers un OS en EL3. Précisément, le code gère les ioctls comme ceci :

- Ioctl1 : un buffer provenant du programme *userspace* de taille 0x101010 octets est copié en espace noyau, puis un SMC 0x83010004 est fait.
- Ioctl2 : un buffer de 32 octets est reçu de *userspace* et est passé en huit fois par un SMC 0xf2005003.
- Ioctl3 : un SMC 0xf2005001 est déclenché (sans argument)
- Ioctl4 : un SMC 0xf2005002 est déclenché (sans argument), il retourne une valeur en fonction du retour du SMC.

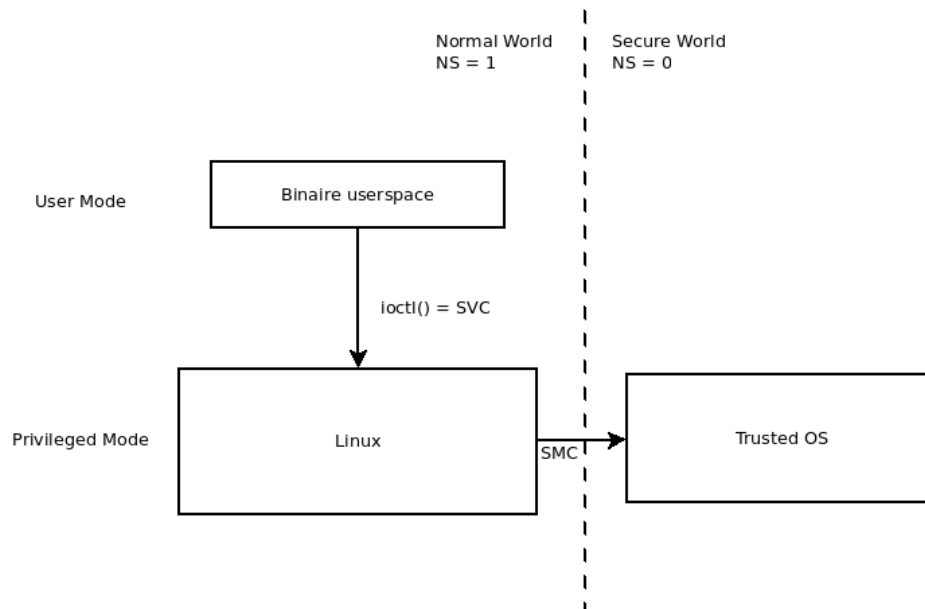
Les SMC sont faits depuis Linux grâce à la fonction `__arm_smccc_smc()` du noyau. Sur certaines

architectures ARM, et de base sur les récentes (armv8), il existe plusieurs niveaux d'exception pour la gestion du logiciel :

1. Niveau 0 (EL0) : Le mode user où tournent les applications *userspace*
2. Niveau 1 (EL1) : Le mode dans lequel tourne le système d'exploitation, ici Linux
3. Niveau 2 (EL2) : Le mode fait pour les hyperviseurs (absent dans notre cas)
4. Niveau 3 (EL3) : Le mode du **secure world**, aussi appelée TrustZone pour ARM ou TEE de manière générale (Trusted Execution Environment).

Le passage d'un niveau à l'autre se fait par des instructions spéciales. Les applications appellent le noyau grâce à l'instruction **SVC**. Ici, le noyau Linux appelle l'OS en EL3 grâce à l'instruction **SMC**. Lorsqu'on passe en mode *sécurisé*, le bit NS (Non Secure) passe alors à 0, ce qui donne des droits supplémentaires sur le hardware de la même manière que l'OS peut avoir des droits supplémentaires par rapport aux applications.

Pour résumer :



Pour bien comprendre le passage d'argument, j'ai lu le document d'ARM **SMC Calling Convention** qui explique tout ce qu'il faut savoir. Dans le cas du driver `sstic.ko`, seulement les deux premiers `ioctls` transmettent des données en TrustZone :

- `Ioctl1` : X1 contient un pointeur vers les données du gros buffer et X2 contient la taille.
- `Ioctl2` : X1 contient le numéro du découpage (0 à 7) et X2 contient 4 octets de données.

Un SMC peut retourner des informations en mettant à jour X0, X1, X2 ou X3.

## 4.3 Dump des données des deux premiers ioctls

Vu que l'application envoie des données au driver qui les transmet ensuite en TrustZone, j'ai voulu extraire celles-ci. Pour cela, j'ai patché le programme strace grâce au Buildroot que j'avais mis en place lors de l'étape 3. Mes modifications sont faites sur le fichier `ioctl.c` qu'il suffit de placer dans le dossier de Buildroot `output/build/strace-4.26` puis de reconstruire le paquet et de l'utiliser :

Listing 4.5 – Extraction des données de l'application avec strace modifié

```
# Recompilation de strace et envoi sur la VM
make strace-rebuild
scp output/build/strace-4.26/strace phonesstic:
# Dump des données
ssh phonesstic
./strace -X raw safe_02/decrypted_file $FLAG 2>&1 | tee dump
# Beaucoup de données sont extraites
```

On obtient ainsi les données des deux ioctls. Le premier passe un gros buffer de 0x101010 octets. Je suppose que les données sont stockées dans le binaire. Le deuxième ioctl envoie simplement le flag au format binaire (d'où les 32 octets).

## 4.4 Méthodologie

Pour résumer ce que l'on sait, l'application envoie un gros buffer en TrustZone puis envoie le flag qu'on lui fournit en argument. Ensuite, un enchainement de deux SMCs est fait jusqu'à ce que l'un retourne autre chose que 0.

Ainsi, pour avancer sur cette étape, il va falloir analyser ce qu'il se passe dans le code en EL3. Pour cela, je suis d'abord parti sur le fait d'instrumenter QEMU. Après tout, dans la vraie vie, la TrustZone n'est pas facilement accessible mais, dans notre cas, nous l'émulons avec QEMU. Pour récupérer le code de celui-ci :

Listing 4.6 – Récupération des sources de QEMU

```
git clone https://github.com/qemu/qemu.git
git checkout v3.1.0
patch -p1 < [patch]
```

J'ai inclut au PDF sous formes de patches les modifications que j'ai faites. Globalement, ça m'a servi à :

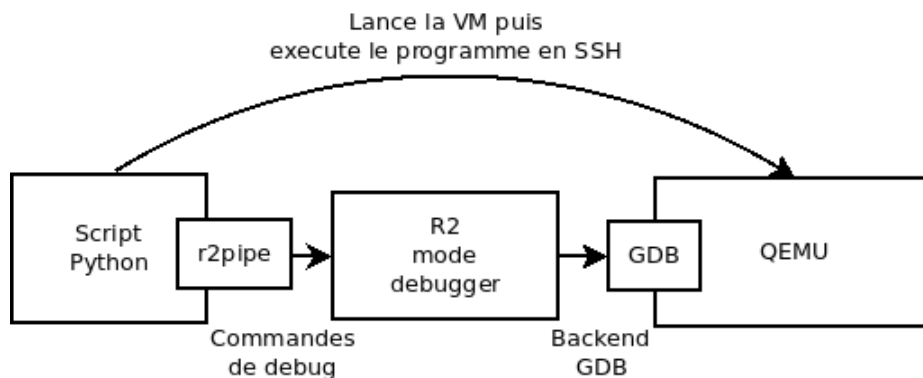
- Dumper tous les SMCs avec les arguments et l'adresse l'ayant déclenchée.
- Dumper le code en TrustZone.
- Tracer l'exécution après les SMC.

Bien que les deux premiers points m'ont énormément servi, j'ai eu pas mal de difficultés à



tracer l'exécution depuis QEMU. La raison principale à cela est que QEMU *optimise* la traduction du code ARM en créant une sorte de cache de code pré-traduit appelé *tcg* et dont le code se trouve dans `accel/tcg`. Ainsi, l'émulateur `armv8` n'est pas forcément utilisé pour chaque instruction à exécuter si le *tcg* associé est déjà calculé et toujours valide.

J'ai alors choisi de changer de méthode et d'utiliser un debugger. En lisant le code de QEMU, j'ai vu qu'il permettait de debugger tout le code émulé, y compris celui en EL3. QEMU possède un backend pour GDB et radare2 possède un backend pour GDB. Donc j'ai utilisé le montage *simple* suivant :



Tout le code Python que j'ai utilisé pour ce challenge se trouve dans le script **r2debug.py** inclut dans le PDF. Le script lance la VM QEMU avec l'option pour activer le debug et attendre une connexion de debugger `-s -S`. Puis le script lance **radare2** en mode debugger en lui demandant de se connecter sur `gdb://127.0.0.1:1234` (le port est celui par défaut). Ensuite, pour chaque test, je crée deux fonctions : une pour poser des breakpoints et une autre qui exécute du code de debug lorsqu'un breakpoint se déclenche. Dans la suite de ce document, je présenterai quelques unes des ces fonctions qui m'ont permis d'avancer dans cette étape. Pour tester une de ces fonctions, il suffit de modifier la ligne suivante à la fin du script :

Listing 4.7 – Changer de test dans **r2debug.py**

```
br_handle = fonction_du_test(r)
```

Evidemment, pour debugger sur des breakpoints, encore faut-il avoir les adresses. La méthode que j'ai utilisée est :

- Reverse *offline* avec Ghidra le code extrait avec QEMU
- Utiliser radare2 pour m'aider à comprendre l'exécution *runtime*

Utiliser un debugger pour cette étape est indispensable car il y a plusieurs portions de code dont l'exécution est particulière et que nous ne verrions pas avec du simple reverse hors-ligne.

Enfin, complètement par hasard, pour mon travail, j'ai cloné un projet OpenSource nommé **ATF** (ARM Trusted Firmware) à la même période où je faisais l'étape 4. Ce code est l'implémentation de référence d'un OS en TrustZone. Je me suis dit "et si...", et oui :) Le fait de connaître l'origine du code en TrustZone m'a été un peu utile car cela m'a permis d'identifier facilement les fonctions de l'OS et ainsi nommer correctement les fonctions dans Ghidra (`printf`, `panic`, `memcpy`, ...).

Le fichier inclut dans le PDF **ATF\_lvl4.gzf** contient l'export projet Ghidra. Il contient deux dumps mémoire en EL3 effectués avec QEMU qui sont les zones exécutées lors des SMCs qui nous intéressent.

Le code en EL3 est coupé en deux types de SMCs. D'une part, il y a les SMC du type 0xf200500x (TrustOS calls) et d'autre part les SMC 0x830100xx (OEM service calls). Les handlers de ces SMC se trouvent à deux endroits différents en mémoire :

- 0x0e031034 = Handler OEM (nom dans mon projet Ghidra : SMC\_8301000x\_handler)
- 0x0e200c08 = Handler TrustOS (nom dans mon projet Ghidra : smc\_f2xx\_statemachine1)

Pour vérifier que j'avais bien compris la succession des SMCs faits par le driver, j'ai commencé par tracer tous les SMCs d'abord avec QEMU, puis avec mon script radare2.

Listing 4.8 – Dump de tous les SMCs et leurs arguments

```
def breakpoint_handle_smc_dump(r):
    registers = r.cmdj('drj')
    if registers['pc'] == 0x0e037400: # Handler SMC
        print "%08x - %08x %08x %08x %08x %08x %08x" % (registers['x0'],
            registers['x1'], registers['x2'], registers['x3'], registers['x4'],
            registers['x5'], registers['x6'])

def smc_dump(r):
    set_breakpoint(r, '0e037400')
    return breakpoint_handle_smc_dump
```

Le script génère une longue liste de SMCs :

Listing 4.9 – Exemple de SMC listés

```
EXCP_SMC 0000000083010004 [000000007d800000, 0000000000101010, ... ]
EXCP_SMC 00000000f2005003 [0000000000000000, 00000000aaaaaaaa, ... ]
EXCP_SMC 00000000f2001000 [0000000000000000, 00000000aaaaaaaa, ... ] <- ???
EXCP_SMC 0000000083010002 [000000000000000f, 0000000000000000, ... ] <- ???
...
```

Pour une exécution du programme, on obtient un peu moins de 90000 SMC. Or il y a  $(1 + 8 + (9366 * 2))$  SMCs faits par le driver, ce qui est beaucoup moins. Comme le confirme l'adresse de déclenchement de certains SMCs, le code en TrustZone fait également lui-même des SMCs.

Dans la suite, je parlerai d'initialisation pour les deux premiers ioctls et de runtime pour les  $(9366 * 2)$  ioctls restant (ioctl 3 et 4 répétés en boucle).

En analysant l'ordres des SMCs, on constate le flux suivant en dehors de l'initialisation :

- 0xf2005001 (ioctl3 - déclenché par le driver)  
     0x0f2001000  
     **0x083010001 (arg=0xf)**  
     0x0f2005001
- 0xf2005002 (ioctl4 - déclenché par le driver)

0x0f2001000

**Un nombre variable de SMCs OEM**

0x0f2005002

Je n'ai toujours pas compris à quoi servent les SMCs 0x0f2001000 et le deuxième SMC identique à celui qui est fait par le driver (peut-être pour revenir proprement au kernel Linux). Leur compréhension n'est pas utile pour la résolution de l'étape.

Pour comprendre ce qui est fait par le driver, il y a donc six choses à analyser :

- L'implémentation du 0x83010004 (envoi du gros buffer en TZ)
- L'implémentation du 0xf2005003 (envoi du flag en TZ)
- L'implémentation du 0xf2005001 (ioctl 3)
- L'implémentation du 0xf2005002 (ioctl 4)
- L'implémentation des autres SMCs OEM 0x0830100xx utilisés en interne dans la TrustZone

## 4.5 Stockage du gros buffer

J'ai commencé par analyser la manière dont la TrustZone utilise le gros buffer de l'ioctl 1. Dans ce cas, ça semble simple, il n'y a qu'un seul SMC.

L'analyse est très simple puisqu'il s'agit de deux lignes dont j'ai le nom des fonctions (ATF) :

Listing 4.10 – Stockage du gros buffer

```
data_ptr = 0x0e05300;  
memcpy(data_ptr, gros_buffer, size);  
flush_cache(data_ptr, size);
```

On copie simplement le buffer à l'adresse 0x0e05300, puis on revient au driver Linux. J'ai donc logiquement cherché où ce buffer était utilisé. Pour cela, j'ai fait plusieurs tests avec radare2 en mettant des *watchpoints* (en utilisant deux méthodes différentes) mais je n'ai trouvé aucune utilisation. J'ai même fait du pas à pas automatisé et en s'arrêtant dès qu'un registre contient une adresse pointant vers ce buffer : seul le stockage semble utiliser cette adresse.

J'en ai conclu les options suivantes :

1. Ma méthode de debug n'est pas bonne ou
2. Le buffer n'est jamais utilisé ou
3. Il y a un mécanisme qui cache l'utilisation de ce buffer

Encore une fois, Ayman Khamouma m'a donné une piste : il y a des mécanismes d'obfuscation dans le code. Notamment, la configuration MMU est plusieurs fois utilisée pour complexifier la compréhension du programme. En réalité, c'est bien ce qui définit cette étape, il faut **trouver** les différents mécanismes d'obfuscation qui ont été ajoutés. C'est en comprenant ceci que j'ai pu avancer dans la mesure où je me suis dit que si un comportement est anormal alors il s'agit sûrement

d'un nouveau mécanisme d'offuscation. Cela m'a également poussé à vérifier régulièrement, avec mon script `r2debug.py`, que je comprenais correctement ce qu'il se passait.

Pour le gros buffer, il s'agit d'un mapping MMU qui fait pointer deux intervalles d'adresses vers la même zone mémoire (la deuxième utilisation étant l'adresse physique directement). Dans le code d'ATF, il y a la fonction `mmap_add_region_ctx()` qui est également présente en EL3.

Grâce à la fonction `mmap_dump_handle()` de mon script, j'extrait le mapping qui est fait depuis le TrustOS :

Listing 4.11 – Extraction du mapping mémoire

```
python r2debug.py
New mapping [000000000e053000] -> [0000000000413000] size 4096 attr 00000008
New mapping [000000000e055000] -> [0000000000414000] size 1048576 attr 00000008
New mapping [000000000e200000] -> [000000000e200000] size 114688 attr 0000000a
New mapping [000000000e200000] -> [000000000e200000] size 16384 attr 00000002
New mapping [000000000e204000] -> [000000000e204000] size 4096 attr 00000022
New mapping [000000000e21c000] -> [000000000e21c000] size 0 attr 00000008
```

On voit donc que l'adresse `0x0e053000` pointe vers `0x0413000` et il y a bien du code qui l'utilise (que je note dans un coin).

## 4.6 Analyse globale des SMCs TrustOS

J'ai commencé par regarder les SMCs TrustOS (`0x2f...`) car ce sont eux qui appellent les SMCs OEM. Il y en a deux qui nous intéressent en runtime : le 5001 et le 5002 qui sont appelés successivement. J'ai commencé par le 5001.

Pour le SMC `0xf2005001`, le code semble simple, il fait :

1. SMC `0x083010001` `arg=0xf`
2. Le retour du SMC est stockée dans `X0`
3. `ldr X0, [X0] // ????`
4. `ldr X0, [variable_globale]`

Deux choses sont surprenantes après le SMC. En regardant `X0` au retour de ce SMCs `0x083010001`, on se retrouve avec des valeurs assez faibles qui ne correspondent pas à des adresses. En faisant du pas à pas, je me suis retrouvé dans du code (qui correspond à celui que j'avais noté précédemment utilisant le gros buffer) entre les points 3 et 4. L'instruction `ldr X0, [X0]` sert simplement à déclencher un pagefault qui permet de traiter le retour du SMC précédent et de fournir un résultat en fonction de celui-ci. Ce résultat est ensuite stocké dans une variable globale que j'ai appelé `next_5002_cmd` car elle est également utilisée par l'ioctl 4 (SMC 5002).

Le handler de pagefault est déclenché lorsque le processeur accède à une adresse invalide. Il appelle la fonction en `0x0e200e84` que j'ai nommée initialement `store_result()` mais elle porte mal son nom puisqu'elle permet de récupérer une valeur à partir d'une autre.

Cette fonction est assez complexe, elle utilise des instructions NEONs **SM4** pour déchiffrer le gros buffer en mémoire. Voici le pseudo-code C que j'ai imaginé à partir de cette fonction. Le code assembleur est bien plus lisible que le code décompilé dans Ghidra lorsqu'il y a des instructions NEONs en jeu. Pour comprendre cette fonction, j'ai extrait des valeurs à plusieurs endroits, la fonction `store_result_init` du script Python montre les entrées et sorties durant toute l'exécution du programme et permet de bien comprendre le fonctionnement de cette méthode d'offuscation.

Listing 4.12 – Code du handler de pagefault

```
//Const :
TAB1[16] // <- R0 dans le code
TAB2[16] // <- R0 dans le code
BIGBUF[1M] // <- Passé par le premier ioctl

//Input : valeur ("offset") qui a fault
IN4 (aligned -> & fffff0)

char[16] derivate_key(input32b) {
    tmp16 = inversion_des_octets(TAB1);
    tmp16_2 = TAB2 ^ [input32b, input32b, input32b, input32b];

    key = SM4EKEY(tmp16, tmp16_2);
    key = inversion_des_mots(key);
    return key;
}

char[16] encode(key, val) {
    res = SM4E(tmp16_3, key);
    res = inversion_des_mots(inversion_des_octets(tmp16));
    return res;
}

key1 = derivate_key(IN4)
res1 = encode(key1, inversion_des_octets(BIGBUF[IN4]))

if (IN4 & 0xf > 0xc) {
    key2 = derivate_key(IN4 + 0x10)
    res2 = encode(key2, inversion_des_octets(BIGBUF[IN4+0x10]))
}

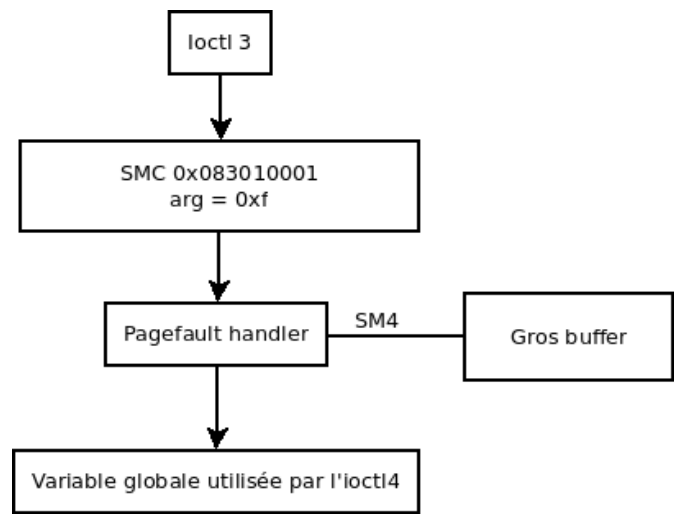
if (IN_X2 == 1) { // Stockage d'information
    res1[IN4 & 0xf] = IN_X1;

    res3 = encode(key1, res1)
    BIGBUF[IN4] = res3; // 16 octets

    if (IN4 & 0xf > 0xc) {
        /* Jamais fait */
        res4 = encode(key2, res2);
        BIGBUF[IN4+0x10] = res4; // 16 octets
    }
} else { // Lecture d'information
    NEW_X0 = res1[IN4 & 0xf]; // Retour du pagefault
}
```

La compréhension du SM4 n'est au final pas important, il s'agit simplement d'une nouvelle technique d'offuscation. Tout ce qu'il faut retenir de cette fonction, c'est qu'elle permet de lire une valeur décodée du gros buffer à partir d'un index. On passe celui-ci par l'adresse qui provoque le pagefault. Enfin, il y a une autre utilisation possible de cette fonction pour stockée une information en mémoire (pas utilisé pour l'ioctl3).

Pour résumer :



Il faut voir ça comme un processus de translation d'une valeur vers une autre. Cette translation ne dépend pas de la clé et est toujours le même. Il y a toujours quelque chose que nous ne connaissons pas dans ce que fait l'ioctl 3 : le fonctionnement du SMC OEM 0x083010001 (arg=0xf) que nous verrons plus tard.

Avant de regarder ce que font les SMCs OEM, j'ai analysé le SMCs 5002 fait par l'ioctl 4. Celui-ci possède une fonction dédiée à l'adrsse 0x0e2005a4 (j'ai nommé la fonction F2xx.5002 dans Ghidra).

La fonction utilise la variable globale écrite lors de l'ioctl 3. Cette variable est une sorte de **commande** car elle détermine ce que fait le SMC 0xf2005002. Cette commande est codée sur 3 octets (24bits) et est encodée de deux manières différentes :

Listing 4.13 – Encodage de la commande pour le SMC 5002

Encodage 1 :	OP1	OP2	VAL1	VAL2	
Encodage 2 :	OP1	OP2	VAL1	VAL4	
Bits	24	20	18	14	0

Suivant les valeurs de OP1 et OP2, des SMCs différents sont faits avec les arguments contenus dans VAL1 et VAL2. La fonction **debug\_states\_5002** de mon script permet de générer un fichier qui contient la liste des commandes ainsi que les SMCs OEM qui sont fait par la suite.

Listing 4.14 – Portion de la trace générée avec la fonction **debug\_states\_5002**

```

>>>> SMC f2005002 - 00000000 00000000
x0 002d0002 cmd (w2) 02
  
```

Dans cet exemple, la commande **0x2d0002** provoque la génération du SMC OEM 0x83010022 avec `arg1=4` et `arg2=2`. Le reverse de la fonction qui gère les commandes comporte deux types de déclenchement des SMCs OEM. Le premier est un appel normal avec l'instruction **SMC**. Le deuxième type est complexe et est basé sur une nouvelle technique d'offuscation. En effet, lors de certaines commandes, au lieu de faire un SMC OEM classique, une fonction est appelée et déclenche un SMC indirectement. Pour comprendre ceci, il faut regarder la fonction en question que j'ai nommée **smc\_trampoline**.

J'ai eu beaucoup de difficulté à tracer ce qui était réellement fait avec ces trampolines. La fonction écrit le registre **ELR\_EL1** qui modifie l'endroit où on retourne lors d'un ERET. Le registre **SPSR\_EL1** est également modifié pour **désactiver le debug** et **utiliser le mode 32bits**. J'ai eu deux problèmes avec radare2 : je n'arrive pas à lire les registres de coprocesseur (ceux qui ne sont pas PC ou X0 à X30) et je n'ai pas réussi à passer le debugger en 32bits dynamiquement. Pour debugger, j'ai extrait le code pointé par ELR\_EL1 qui commence en 0x0e205000. Dans mon projet Ghidra, j'ai nommé ces portions de code 32bits **chaine0** à **chaine8** (il y a autant de commandes qui utilisent des trampolines différents). Il suffit alors de recharger cette zone mémoire extraite dans Ghidra mais en mode ARM32. J'ai mis le projet Ghidra pour ces trampolines dans le PDF (fichier **trampoline\_32bits.gzf**). Chaque chaîne écrit les informations des SMCs sur lesquels sauter dans les registres puis effectue un **SWI** dont le handler est en 0x0e203600 (fonction **trampoline\_handler**). Si la chaîne fait un SWI 0x1338, le handler revient sur la chaîne et si la chaîne fait un SWI 0x1337, le handler revient dans notre fonction `F2xx_5002()`. Une fois ce mécanisme compris, j'ai pu comprendre ce que font ces chaînes :

Listing 4.15 – Description des trampolines

```
chaine0:
    X4 = X7^X8
    SMC 83010001 arg1=f -> ret
    SMC 83010002 arg1=f , arg2=ret+3

chaine1:
    SMC 83010001 arg1 = x0 -> ret1
    SMC 83010002 arg1 = x0 , arg2 = ret1+3

chaine2:
    input x1 X9 = *(09010000) (32bits)
    SMC 83010001 arg1=f -> ret
    if (x9 <= 5) {
        arg2 = ret + 3
    } else {
        arg2 = x1
    }
    SMC 83010002 arg1=f, arg2

chaine3:
    SMC 83010001 arg1=x0 -> ret
    SMC 83010002 arg1=x0, arg2=(ret << x1)

chaine4 :
    SMC 83010001 arg1=x0 -> ret
```

```

SMC 83010002 arg1=x0, arg2=(ret >> x1)

chaîne5 :
    SMC 83010001 arg1=x0 -> ret
    SMC 83010002 arg1=x1, arg2=ret

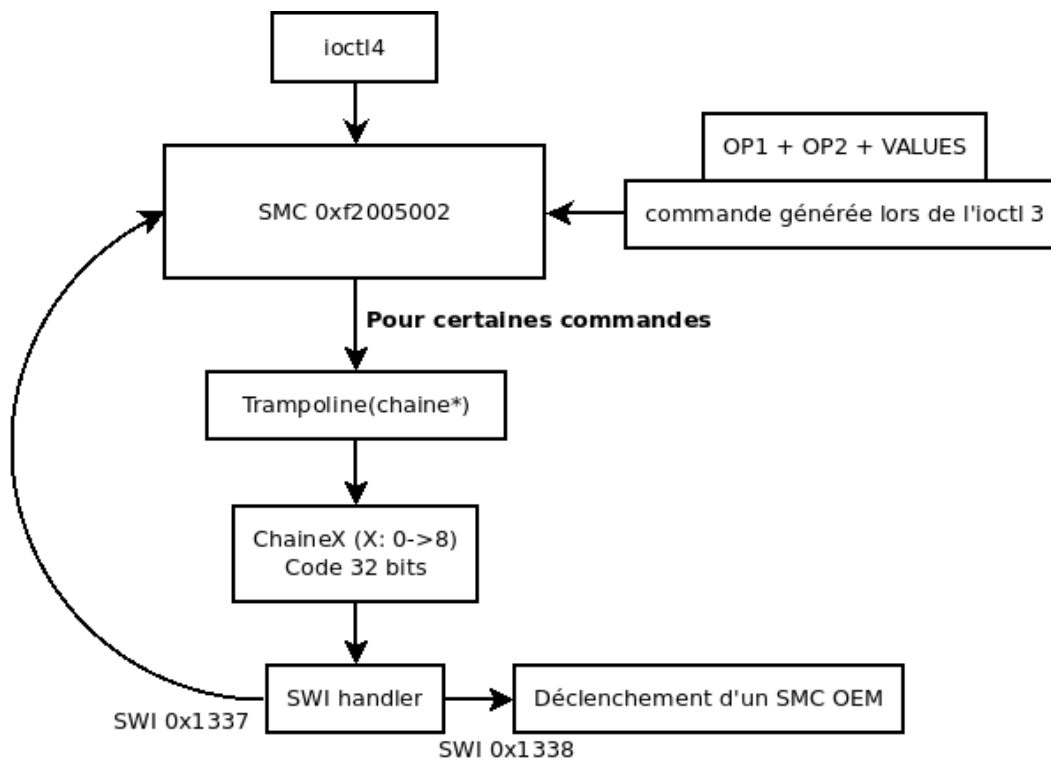
chaîne6 :
    SMC 83010001 arg1=x0 -> ret
    x2 = ((ret & 0xff) << 8) | (ret >> 8) [32bits]
    SMC 83010002 arg1=x0, arg2=x2

chaîne7 :
    SMC 83010002 arg1=x0, arg2=x1

chaîne8 :
    *(09010008) = x0
    SMC 83010001 arg1=f -> ret
    SMC 83010002 arg1=f, arg2=ret+3

```

Pour résumer, lorsque le trampoline est utilisé :



Une state-machine, des pagefaults, le SM4, les trampolines... Il faut s'attendre à encore plus d'offuscation.

Si on doit résumer, le SMC 5002 est un gros aiguillage avec un peu de logique qui effectuent des SMC OEM, soit avec un trampoline, soit directement. On remarque également que le premier argument des SMCs OEM est toujours une valeur de 0x0 à 0xf. D'un point de vue global, il y a une grande quantité de SMC 83010001 (qui retourne une valeur) et de SMC 83010002. Lorsqu'on a arg1=0xf, on voit souvent un +3...



Il y a néanmoins quatre commandes particulières qui ont un comportement différent :

- OP1 = 0xa : Il s'agit de la dernière commande qui permet au SMC de retourner autre chose que 0. La valeur retournée dépend du retour d'un SMC OEM (83010001 arg=0).
- OP1 = 0xc : On utilise le principe des pagefaults pour **stocker** une valeur hard-codée dans la fonction : 0x612e7270, 0x6766722e, 0x666e632e, 0x2e76662e, 0x76706e73, 0x66407279, 0x70766766, 0x7465622e.
- OP2 = 0x2 : On utilise le principe des pagefaults pour **stocker** une valeur issue d'une SMC OEM à un endroit donné (après le gros buffer).
- OP2 = 0x1 : On utilise le principe des pagefaults pour **récupérer** une information (pour un index VAL1 donné).

Si on fait le bilan de l'API basée sur les pagefaults, on remarque que les adresses utilisées dessinent le mapping suivant :

- 00000000 : Commandes (ioctl3)
- 00001000 : Lookup table (ioctl4)
- 00100000 : Lecture/Stockage de données

Ainsi, la seule chose qui nous manque est de comprendre ce que font les SMCs OEM.

## 4.7 Analyse globale des SMCs OEM

Le handler des SMC OEM (0x0e031034) contient également de l'offuscation. Il y a un énorme *switch-case* qui contient pour chaque cas des instructions AES. Si on regarde de plus près, on se rend compte que les instructions de chiffrement (1 round) avec une clé fixe sont faites lors du stockage d'une valeur dans un tableau de taille 16 (0xf). Lorsqu'on lit ce tableau, on déchiffre (1 round) avec la même clé fixe. Cette clé correspond à une partie de notre flag d'entrée mais ce n'est pas vraiment important. La conclusion sur l'utilisation de l'AES est que ça ne sert à rien, il suffit de comprendre qu'on stock et qu'on lit un tableau de 16 entrées.

A ce moment critique, je comprend le principe du challenge : les SMC TrustOS et OEM implémentent une VM. Le tableau manipulé dans les SMC OEM constitue **les registres** de cette VM. J'ai donc analysé l'ensemble des SMCs OEM utilisés lors de l'exécution du programme (je n'ai pas reverse les quelques SMCs qui n'étaient pas utilisés).

Le reverse fut assez rapide mais il y a encore quelques méthodes offuscations en utilisant du NEON, du SVE ou encore des instructions manipulant des nombres complexes pour implémenter des opérations de bases (addition par exemple!).

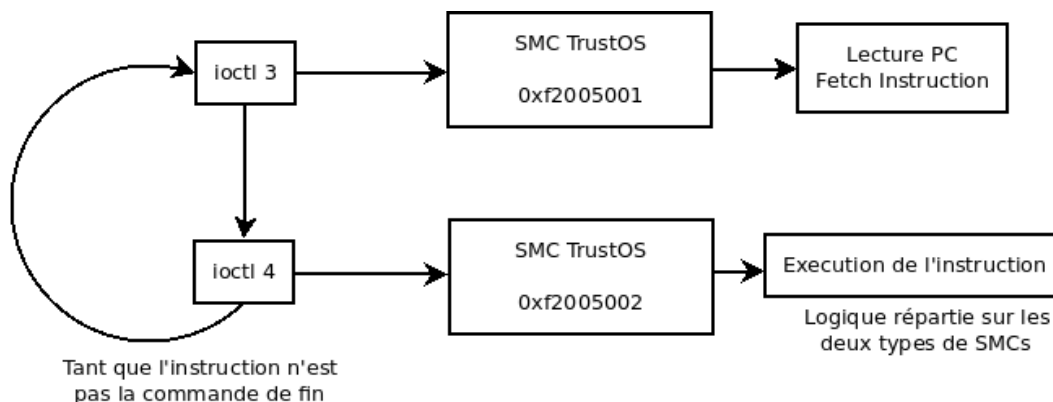
Voici le résultat de cette phase d'analyse :

- 0x83010003 : store AES key BBBB CCCC DDDD
- 0x83010004 : store VM commandes
- 0x83010001 : `ret = register[arg1]`
- 0x83010002 : `register[arg1] = arg2`

- 0x83010011 : `register[arg1] = register[arg1] - 1; register[0xf] += 3`
- 0x83010012 : `register[arg1] += register[arg2]; register[0xf] += 3`
- 0x83010013 : `register[arg1] = abs(register[arg1] - register[arg2]); register[0xf] += 3`
- 0x83010016 : `register[arg1] = register[arg1] ^ register[arg2]; register[0xf] += 3`
- 0x83010022 : `register[arg1] = arg2 + register[arg1]; register[0xf] += 3`
- 0x83010023 : `register[arg1] += x0; register[0xf] += 3`
- 0x83010027 : `register[arg1] = register[arg1] & arg2; register[0xf] += 3`
- 0x83010028 : `register[0xf] = arg2`

On note que ces opérations sont simples. Le SMC 0x83010001 permet de lire un registre et le 0x83010002 permet d'écrire. Le reste fait simplement des opérations basique (arithmétique et logique) entre des registres. Le registre 0xf est toujours incrémenté de 3, il s'agit de l'équivalent du PC puisqu'il est utilisé pour récupérer l'**instruction** à exécuter lors de l'ioctl 3.

Pour résumer, on a encore une VM dont l'implémentation globale est répartie sur le programme userspace, le driver Linux et les deux types de SMCs :



Les instructions peuvent manipuler des registres mais également lire et écrire dans une mémoire (gérée par le système des pagefaults). Dans cette mémoire, il y a le code de la VM (assez court), un grand tableau de données constantes (probablement une lookup table) et une zone à la fin utilisée pour stocker des données (et les lire plus tard).

## 4.8 Autres protections

Il y a deux autres protections anti-debug dont je n'ai pas encore parlé. La première concerne les adresses 0x09010000 et 0x09010008 utilisées dans la chaîne2 et la chaîne8 du trampoline. Elles sont utilisées si OP1 vaut 0xd, 0xe ou 0xf. Durant toutes les traces que j'ai faites, le flux d'exécution semblait assez constant. Or ces variables altèrent légèrement celui-ci. Si on compare l'enchaînement des SMCs (c'est à dire ce que fait la VM), on remarque :

- D'un lancement du programme à l'autre, l'enchaînement des SMCs varie légèrement lorsque je les dump avec radare2.

- La trace est toujours la même lorsque je la génère avec QEMU.

La différence dans le flux d'exécution avec radare2 est exclusivement due à ces variables. Vu qu'il n'y a pas de problème avec QEMU, j'en ai déduit qu'il s'agit d'une autre technique anti-debug. De plus, lorsqu'on dump les SMCs, on remarque un SMC qui s'active périodiquement (500ms) dès le démarrage de la VM (la VM QEMU). Grâce à mon script Python, j'ai vérifié que l'adresse 0x09010000 était un compteur incrémenté par le biais de ce SMC périodique. Ainsi, il y a une protection qui vérifie que le programme ne met pas trop de temps à s'exécuter et sinon altère silencieusement le flux d'exécution de la VM (la VM de l'étape 4). Pour la résolution du challenge, nous n'avons pas besoin de contourner cette mesure car on peut considérer que le fonctionnement normal de la VM (celle de l'étape 4) est le cas où ce compteur vaut 0.

Enfin, le point qui m'a sûrement posé le plus de problème à la fin est une protection assez invisible. Tout d'abord, en analysant le code en TrustZone, on remarque rapidement la présence de chaîne suspecte du type `ps -ef | grep -q qemu-system` ou encore `/proc/self/cmdline`. Je me suis dit que c'était une tentative pour nous faire perdre du temps sur l'analyse (la fonction s'appelle d'ailleurs `wtf()` dans mon projet Ghidra). Mais à la fin, je n'arrivais pas à résoudre le challenge : la lookup table était anormale. En comparant les valeurs du gros buffer (après déchiffrement SM4) à travers les arguments SMCs, je me suis rendu compte que la lookup table était différente suivant si je lance le programme avec QEMU ou si j'utilise radare2. Evidemment, j'ai tout de suite pensé à une autre protection anti-debug. Le comportement normal est celui avec QEMU utilisé pour le dump de SMCs puisque la modification de QEMU est complètement invisible pour le code émulé. J'ai fait plusieurs tests pour déterminer que le code en TrustZone était capable de lire la ligne de commande qu'on donne à QEMU. Il y a du code qui vérifie qu'on ne passe pas les options de debug du type `"-s"` ou `"-gdb"`. La solution de contournement que j'ai utilisée est simplement de hard-coder l'option dans QEMU (puisque je le recompile déjà). J'ai joint le patch qui fait ça à ce PDF.

## 4.9 Désassemblage de la VM

Une fois qu'on a compris l'implémentation globale de la VM, comment sont encodées les instructions et comment les registres et la mémoire sont gérés, il ne manque plus qu'à faire un désassembleur. J'ai développé le programme **VM.py** qui prend en entrée le code de la VM et qui l'émule. Au final, j'ai simplement eu besoin d'afficher la description de l'instruction plutôt que de l'émuler complètement.

Bien sûr, il faut extraire les différentes informations constantes de la VM : les instructions, la lookup table ainsi que les quelques constantes hard-codées dans le binaire. Pour le code de la VM ainsi que la lookup table, il s'agit de demander au système de pagefault de nous déchiffrer le contenu de la mémoire de la VM. On aurait pu réimplémenter l'algorithme basé sur SM4 pour déchiffrer hors-ligne le gros buffer. J'ai choisi de simplement utiliser le code EL3 grâce à mon script `r2debug.py`. La fonction `dump_vm_code` permet d'extraire la mémoire. Je l'ai d'abord utilisé pour récupérer les instructions avec un pas de 3 octets et à partir du début. Puis j'ai récupéré la lookup table avec un pas de 4 octets (pour aller plus vite que d'umper octet par octet) à partir de l'offset 0x1000.

Listing 4.16 – Extraction de la mémoire de la VM

```

VM_memory=0x1000
pf_to_handle=0
incr=4

def dump_vm_code(r):
    global VM_memory
    global pf_to_handle
    global incr
    registers = r.cmdj('drj')

    if registers['pc'] == 0x0e200c78: # print pagefault result (not the first
time)
        # Inject value
        if VM_memory > 0x1000:
            print "%08x : %08x" % (VM_memory-incr, registers['x0'] & 0xffffffff
)
            sys.stdout.flush()
        if VM_memory > 0x101010: # End of VM code
            sys.exit(0)

    if registers['pc'] == 0x0e200c7c: # override pagefault input
        r.cmd('dr x0 = 0x%08x' % VM_memory)
        pf_to_handle = 1

    if registers['pc'] == 0x0e20229c: # go back to 1
        if pf_to_handle == 1 :
            r.cmd('dr x0 = 0x%08x' % 0x0e200c78)
            VM_memory = VM_memory + incr
            pf_to_handle = 0

def dump_vm_code_init(r):
    set_breakpoint(r, '0e200c78') # before page fault
    set_breakpoint(r, '0e200c7c') # before page fault
    set_breakpoint(r, '0e20229c') # at the end of the pagefault, go back to
first breakpoint
    return dump_vm_code

```

Le code se positionne simplement avant le pagefault et change X0 (entrée du pagefault), puis récupère la sortie du pagefault. La difficulté a été d'automatiser le processus, c'est à dire de revenir avant le pagefault pour extraire l'adresse suivante. Je n'ai pas réussi à changer le PC par une commande r2. Pour palier à ce problème, j'ai écrasé l'adresse du retour dans le pagefault handler de manière à revenir avant celui-ci.

On se retrouve alors avec deux fichiers **lookup\_table.r2** et **vm\_code\_clear.r2** qui sont les entrées de mon émulateur/désassembleur VM.py.

Ensuite, il suffit de lancer le programme VM.py pour désassembler le code. J'ai mis dans le PDF le résultat complet dans le fichier **VM.asm** dont voici un extrait :

Listing 4.17 – Extraction de la mémoire de la VM

```

0x0102 : 023800 0PCODE : reg[0x8] = reg[0xe]

```

```

0x0105 : 5e0003 OPCODE : reg[0x8] >> 3
0x0108 : 7e0001 OPCODE : reg[0x8] = reg[0x8] & 0x00000001
0x010b : cf8000 OPCODE : storage[reg[0xe]] = constants[reg[0xe]]
0x010e : 9e012f OPCODE : goto 0x012f if reg[0x8] != 0
0x0111 : 020c00 OPCODE : reg[0x8] = reg[0x3]
0x0114 : 00f800 OPCODE : reg[0x3] = reg[0xe]
0x0117 : 2cc001 OPCODE : reg[0x3] += 0x0001
0x011a : 60c000 OPCODE : reg[0x3] = reg[0x3] ^ reg[0x0]

```

## 4.10 Analyse du code de la VM

Le code de la VM n'est pas très volumineux au final. Il y a seulement 147 instructions est très lisible et pas du tout offusqué. Il possède 3 parties assez visibles :

- Une récupération du flag d'entrée, les octets sont légèrement mélangés
- Pour chaque **8 octets** de flag : une partie **intermédiaire** constitué d'une boucle qui part du flag avec 0x20 itérations
- Une vérification finale qui réussie (retour de l'ioctl4 = 0) si le résultat de la partie intermédiaire correspond aux constantes hard-codées dans le code

Le code qui n'est pas trivial est donc la boucle dans la partie intermédiaire. Il est important de noter que cette boucle traite les quart de clé de manière indépendante, je l'avais déjà noté en comparant les arguments des SMCs avec des entrées légèrement différentes.

Cette boucle possède un état de 4 registres où seulement 16 bits sont utilisés. Pour y voir plus clair, voici le pseudo-code C de cette boucle :

Listing 4.18 – Partie intermédiaire du code la VM

```

int i = 0;
int j = 7;

uint16_t in1, in2, in3, in4;
uint32_t r4, r5, r6, r9, ra, rb;

#define decrease_j(j) { if (j == 0) { j = 0xa; } j--; }

static inline uint16_t lookup(unsigned int j, uint16_t in1)
{
    unsigned char r4, r5, r6;
    uint32_t idx;
    uint16_t in1_l, in1_h;

    in1_l = in1 & 0xff;
    in1_h = (in1 >> 8) & 0xff;

    idx = (j << 16) + (in1_l << 8) + in1_h;
    r6 = lookup_table[idx] & 0xff;

    decrease_j(j)
}

```

```

    idx = (j << 16) + (in1_h << 8) + r6;
    r5 = lookup_table[idx] & 0xff;

    decrease_j(j)

    idx = (j << 16) + (r6 << 8) + r5;
    r4 = lookup_table[idx] & 0xff;

    decrease_j(j)

    idx = (j << 16) + (r5 << 8) + r4;
    r6 = lookup_table[idx] & 0xff;

    decrease_j(j)

    return (r6 << 8) | r4;
}

for (i=0x1f; i>=0; i--) {

    r9 = lookup(in1);

    if ((i >> 3) & 1 == 0) {
        r8 = in3;
        in3 = (i + 1) ^ in0 ^ in1;
        in0 = r9;
        in1 = in2;
        in2 = r8;
    } else {
        r8 = in0;
        in0 = r9;
        in1 = (i + 1) ^ in0 ^ in2;
        in2 = in3;
        in3 = r8;
    }
}

```

Cette boucle utilise la lookup table en répartissant le lookup sur 4 cases de 1 octet de la table. Elle mélange aussi les octets et les registres à chaque itération. Tout ce qu'il reste à faire est de retourner cette boucle pour retrouver les entrées à partir des sorties (qu'on fixera avec les valeurs hard-codées).

## 4.11 Récupération de la clé

Comme on peut le voir dans le code précédent, la boucle possède deux parties à retourner.

D'abord, il y a la fonction que j'ai nommée `lookup()` et qui utilise la lookup table. Pour cela, j'ai d'abord converti en header C le dump de cette table pour avoir un tableau de char. La fonction prend un entier `j` connu et un nombre codé sur 16bits. On peut donc simplement brute-forcer les 16bits d'entrée jusqu'à trouver la sortie attendue dans la mesure où 65 536 exécutions de la fonction

lookup() est toujours très rapide.

Listing 4.19 – Brute-force de la fonction lookup

```
static uint16_t lookup_reverse(int j, uint16_t r9)
{
    uint32_t try = 0;
    uint16_t result = 0;
    uint16_t good = 0;
    int found = 0;

    j = (j + 4) % 10;

    do {
        result = lookup(j, try);
        if (result == r9) {
            good = try;
            found = 1;
        }
        try++;
    } while (try < 0x10000);

    return good;
}
```

Ensuite, il suffit d'inverser la boucle. Je l'ai fait manuellement comme lors de l'étape 3, en partant de la fin est en remontant. On se retrouve avec le code suivant :

Listing 4.20 – Retournement de la boucle

```
void reverse(uint16_t in0, uint16_t in1, uint16_t in2, uint16_t in3)
{
    int i = 0;
    int j = 9; // Final value of j (seen in trace generated by r2)
    uint32_t r8;

    for (i=0x00; i<0x20; i++) {
        if (((i >> 3) & 1) == 0) {
            r8 = in2;
            in2 = in1;
            in1 = lookup_reverse(j, in0);
            j = (j + 4) % 10;
            in0 = in3 ^ in1 ^ (i+1);
            in3 = r8;
        } else {
            r8 = in3;
            in3 = in2;
            in2 = in1 ^ in0 ^ (i+1);
            in1 = lookup_reverse(j, in0);
            j = (j + 4) % 10;
            in0 = r8;
        }
    }
    printf("Result : 0x%04x 0x%04x 0x%04x 0x%04x\n", in0, in1, in2, in3);
}
```

Le code complet se trouve dans le programme **resolve\_interm.c**. Pour tester, j'ai d'abord utilisé les valeurs finale obtenues avec mon flag de test pour vérifier que je le retrouvais bien après retournement de l'algorithme. Ensuite, j'ai utilisé les valeurs hard-codées dans le binaire :

Listing 4.21 – Récupération du flag

```
gcc -o resolv resolve_interm.c
./resolv
#Result : 0xaaaa 0xaaaa 0xaaaa 0xaaaa <= Flag de test
#Result : 0xacad 0xaa8b 0x5b55 0x306f
#Result : 0xb3c6 0xdfc3 0xb2d1 0xc807
#Result : 0x7008 0x4644 0x225f 0xebd7
#Result : 0x1a91 0x89aa 0x26ec 0x740e
```

On teste ensuite le flag `acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e` dans la VM et on obtient **Win**.

Pour passer à la dernière étape, il suffit d'utiliser le même script `tools/add_key.py $flag` que pour les autres étapes. On se retrouve alors avec le dernier fichier déchiffré : `safe_03/decrypted_file`.



## 5 Etape 5 : Partition Android

L'étape 5 est une formalité. Le fichier est au format **gzip** qui contient une archive **tar**. On se retrouve alors avec un dossier data qui contient une partition de données Android :

Listing 5.1 – Contenu de l'archive

```
ls -l .
# android
# com.android.apps.tag
# com.android.backupcon
# ...
```

Il s'agit de la dernière étape, on cherche une adresse en challenge.sstic.org :

Listing 5.2 – Contenu de l'archive

```
grep -R sstic
Fichier binaire com.google.android.apps.messaging/databases/bugle_db
correspondant
Fichier binaire com.android.providers.telephony/databases/mmssms.db
correspondant
```

Le fichier **mmssms.db** correspond à la base de données du service *Telephony* qui gère les SMS sur un téléphone Android. Un **file** dessus nous apprend qu'il est au format SQLite. Pour dumper la base :

Listing 5.3 – Extraction des SMS

```
sqlite3 ./com.android.providers.telephony/databases/mmssms.db
sqlite> .output ./dump.sql
sqlite> .dump
sqlite> .exit
```

Dans le fichier dump.sql, on retrouve le SMS en question :

Mission accomplie, comme d'habitude la perspective d'une tournée de shooter au cactus a suffi à corrompre le CO et à choisir la date de publication du challenge. Pour être sûr que les experts sont toujours occupés, j'ai redirigé l'adresse 9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org vers votre boîte mail. Tant que vous ne voyez passer aucun mail, la voie est libre...

## 6 Remerciements et conclusion

Je tiens d'abord à remercier Ayman Khamouma (ak42) qui m'a donné pas mal de pistes et m'a motivé à aller jusqu'au bout du challenge. Il m'a également appris à me servir de radare2 sans lequel je n'aurais probablement toujours pas fini. Je remercie également mes professeurs du BADGE SO à l'ESIEA qui m'ont enseigné des techniques et outils utiles pour finir le challenge. Merci également aux concepteurs du challenge pour ce défi qui a sûrement dû leur demander beaucoup de travail de préparation.

Pour conclure, j'ai beaucoup aimé ce challenge car il est orienté embarqué et permet de découvrir un système extrêmement récent (kernel 5.0, BR 2018.11, QEMU 3.1, armv8) avec des technologies à la fois d'actualité et d'avenir (TrustZone en particulier). Ce challenge m'a appris énormément de choses à la fois sur le fond des différentes étapes (RSA, schéma logique, exceptions en C++, TZ armv8) mais aussi sur l'utilisation d'outils de debug et de reverse (radare2, Ghidra, strace, QEMU, Python). Un gros point fort de ce challenge est d'être faisable en utilisant seulement des projets OpenSource, ce qui le rend accessible à tout le monde.

Enfin, en rédigeant ce document, je me rend compte que le challenge m'a également permis de prendre conscience de mes erreurs de méthodologie ou d'approche. Par exemple, j'aurais sûrement dû partir sur l'utilisation d'un debugger directement plutôt que de modifier QEMU (ou strace) dont le code est volumineux. Un autre exemple est que je n'ai presque jamais utilisé le debugger (r2 ou gdb) en mode interactif pour analyser le comportement des programmes alors que ça aurait pu me donner des informations beaucoup plus rapidement (par exemple sur les pagefaults).

Pour finir, la difficulté principale de ce challenge pour moi fut la gestion du temps. Les deux premières étapes sont faisables assez rapidement mais les deux suivantes sont très longues. Ce challenge fut un vrai **marathon**. J'ai dû faire pas mal d'efforts pour trouver du temps suffisamment contigu pour avancer mais aussi pour me vider la tête pendant les périodes où je n'étais pas sur le challenge !