Challenge SSTIC 2019

Nics

28 avril 2019

Table des matières

Introduction **4**

1 *Un programme qui consomme* **4**
1.1 Amorçage avec accrocs . . . 4
1.2 Déchiffrement RSA et exponentiation rapide . . . 5
1.3 Attaque par canal auxiliaire . . . 6

2 *Un programme de portes et de boutons* **7**
2.1 Environnement mobile . . . 7
2.2 Des portes, beaucoup de portes . . . 8
2.2.1 Sélecteur . . . 10
2.2.2 Ou Exclusif . . . 10
2.2.3 Unité Arithmétique et Logique . . . 10
2.2.4 Étage 1 et Étage 1b . . . 11
2.2.5 Étage 2 . . . 11
2.2.6 Étage 3 . . . 11
2.2.7 Implémentation de `secure_device` . . . 11
2.3 Des boutons, beaucoup de boutons . . . 11

3 *Un nain-génieux programme* **12**
3.1 Un programme beaucoup trop petit . . . 13
3.2 On s'équipe (mal) ! . . . 13
3.3 Structure globale . . . 14
3.4 Un réseau pas comme les autres . . . 16
3.5 Pour aller plus vite... . . . 16

4 *Un programme d'exception* **16**
4.1 Binaire fantoche . . . 17
4.2 Noyau raffiné . . . 18
4.3 Mémoire piégée . . . 19
4.4 Mémoire masquée . . . 19
4.5 Programme rusé . . . 20
4.5.1 Registres obfusqués . . . 20
4.5.2 Binaire à l'envers . . . 20
4.5.3 Programme minuté . . . 21
4.5.4 Un objectif mouvant . . . 22
4.6 Inversion corsée . . . 22

5 *Le `grep` final* **23**

Conclusion **24**

Remerciements **24**

A Ressource pour la première étape **25**
A.1 Script pour l'Analyse Simple de la Consommation . . . 25

B Ressource pour la deuxième étape **26**
B.1 Implémentation de `secure_device` 26

C Ressources pour la troisième étape **26**
C.1 Désassembleur `DWARF` "maison" 26
C.2 Décompilateur "maison" 33

D Ressources pour la quatrième étape **36**
D.1 Décompilation par `ghidra` 36
D.2 Script de déchiffrement de la zone mémoire 38
D.3 Désassembleur pour la quatrième étape 41
D.4 Code de la machine virtuelle désassemblé 43
D.5 Code émulant la fonction modifiant l'"entrée" du programme 47
D.6 Code inversant la fonction `f` 48

Introduction

Chaque année depuis 2009, le SSTIC propose aux amateurs et professionnels francophones de la sécurité informatique un challenge. Celui de cette année se trouve `à cette adresse`. On nous présente une mise en contexte :

```
    Bonjour,

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

Sans preuve, il est difficile d'agir à son encontre. Ainsi, nous avons décidé
de saisir son téléphone portable afin de collecter des éléments confirmant nos
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.

Des instructions techniques plus précises vous seront fournies.

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

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

Ainsi qu'une description claire et concise de notre objectif :

```
    Le défi consiste à analyser le micro-logiciel d'un téléphone. Les traces
    laissées sur ce téléphone permettent de remonter à une adresse e-mail
    (...@challenge.sstic.org). À vous de la récupérer.
```

Le challenge va se découper en quatre étapes majeures, à la difficulté croissante.

1 *Un programme qui consomme*

1.1 Amorçage avec accrocs

Le point de départ du challenge est donc une archive contenant de quoi faire démarré le *téléphone sécurisé* dans un environnement virtualisé à l'aide de `qemu`. Plus exactement, cette archive contient quatre fichier :

```
-> tar --list -f challenge_SSTIC_2019-virtual_phone.tar.gz
virtual_phone/
virtual_phone/README
virtual_phone/rom.bin
virtual_phone/power_consumption.npz
virtual_phone/flash.bin
```

Le fichier `README` contient les instructions pour lancer le démarrage du système. Comme prévu il n'y a pas le coffre de clef permettant le démarrage habituel du téléphone et un fichier `keystore` est crée au premier démarrage avant de quitter. Lors du deuxième démarrage, voici le message qui nous accueille :

```
##########################################
#      virtual environment detected      #
#           QEMU 3.1+ is needed          #
##########################################
NOTICE:  Booting SSTIC ARM Trusted Firmware
KEYSTORE: AES Key is still encrypted, need decryption
KEYSTORE: Need RSA key to decrypt
KEYSTORE: RSA private exponent is not set, please set it in the keystore or
enter hex value :
```

Ainsi pour démarrer le système, il est nécessaire d'avoir un `keystore` contenant un exposant privé RSA ou d'entrer manuellement celui-ci.

1.2 Déchiffrement RSA et exponentiation rapide

Le déchiffrement du protocole RSA repose sur une opération fondamentale : l'exponentiation. Lors de celui-ci, le chiffré c est transformé en un clair p en utilisant l'exposant privé d et le module public n de la manière suivante :

$$p = c^d \ [n]$$

où c, d et n sont des entiers de très grande taille (en l'occurrence, 1024 bits).

Pour être plus efficace qu'une exponentiation effectuant d multiplications, il est possible d'utiliser un algorithme dit "d'exponentiation rapide" ou "mettre au carré et multiplier" pour diminuer la complexité à $\log(d)$ multiplications.

Voici l'algorithme d'exponentiation rapide :

Data: c, (d_i) la liste des bits de d avec d_0 le bit de poids faible
Result: $p = c^d$
$p \leftarrow 1$
$sqr \leftarrow c$
for $d_i \in (d_i)$ **do**
  if $d_i = 1$ **then**
    $p \leftarrow p \times sqr$
  end
  $sqr \leftarrow sqr \times sqr$
end

Algorithm 1: Exponentiation rapide

Le problème majeur de cet algorithme est que, selon la valeur du $i^{\text{ème}}$ bit du secret d, le $i^{\text{ème}}$ tour de boucle effectue une ou deux multiplications. Ainsi, un certain nombre de grandeurs physiques mesurables (son, consommation électrique, émission d'ondes électromagnétiques, durée du calcul, ...) dépendent du secret. S'il n'y a aucune contre-mesure en place, des attaques par canaux auxiliaires sont possibles.

1.3 Attaque par canal auxiliaire

Nous avons à notre disposition une trace de la consommation électrique sous forme d'une archive `numpy`. D'après la description, celle-ci a été enregistrée lors du démarrage du téléphone, en présence du coffre de clef. Lorsque l'on regarde un extrait du graphe (cf. Figure 1) de la consommation électrique au cours du temps, on retrouve une certaine périodicité. De plus on peut classer les pics en deux types distincts : les pics épais (type A) et les pics fins (type B).

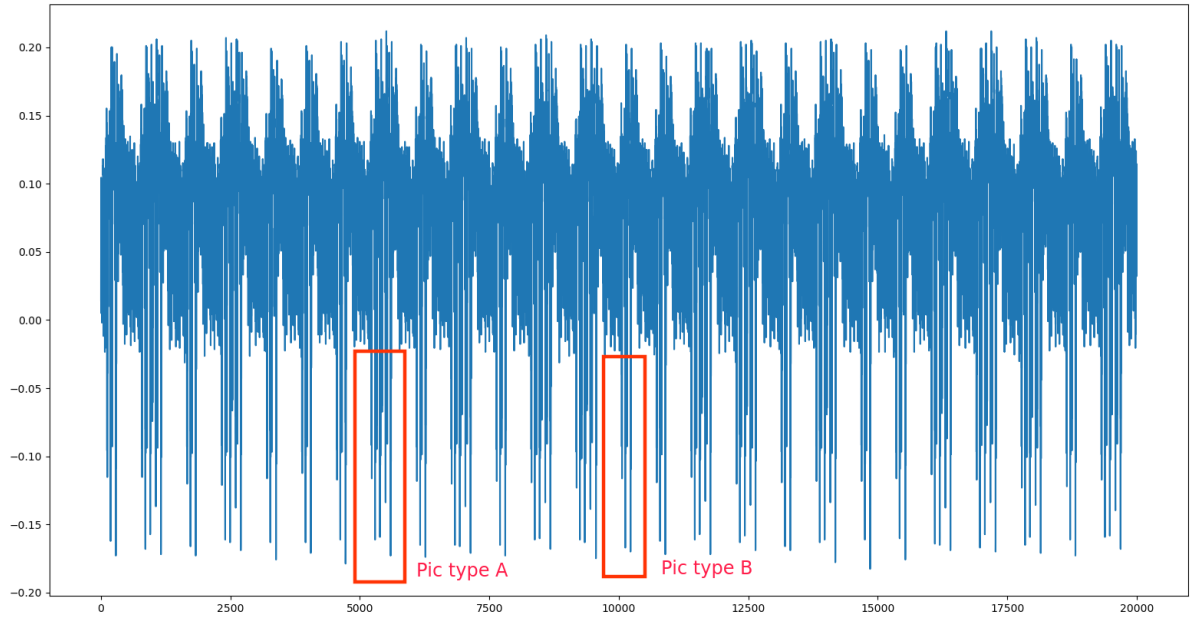

FIGURE 1 – Extrait du profil de consommation électrique

Étant donné que nous n'avons qu'une trace et que celle-ci ne semble pas trop bruitée, Nous allons mener une attaque dite par *Analyse Simple de la Consommation* (*Simple Power Analysis*).

Il y a en tout 1024 pics qui correspondent aux 1024 tours de boucle (un par bit de d). Un pic de type A, le plus long, correspond à deux multiplications. Un pic de type B, le plus court, correspond à une seule multiplication. Ainsi, en déterminant la largeur (et donc le type) de chaque pic, puis en associant à chaque type de pic une valeur pour le bit de d correspondant, nous pouvons retrouver l'exposant secret permettant le déchiffrement lors de l'amorçage du système. Le script `python` utilisé peut être retrouvé en Annexe A.1.

Avec le bon déchiffrement du système, nous obtenons le premier jalon :

Jalon 1. *SSTIC{a947d6980ccf7b87cb8d7c246}*

2 *Un programme de portes et de boutons*

2.1 Environnement mobile

Une fois le système lancé, nous sommes accueilli par un message de bienvenue :

```
##################################
#   Welcome to SSTIC challenge   #
##################################

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

Note:  use /root/tools/add_key.py to unlock safes
KEYSTORE: bad keystore magic
[w] No key for safe_01
KEYSTORE: bad keystore magic
[w] No key for safe_02
KEYSTORE: bad keystore magic
[w] No key for safe_03
```

Il semble donc assez clair que nous allons devoir obtenir successivement les clés pour les différents coffres-forts `safe_01`, `safe_02` et `safe_03`.

Le dossier `/root/` contient :

- Trois coffres-forts ;
- Un dossier `/root/tools/` contenant des outils permettant la validation des clés intermédiaires et le déchiffrement des différents coffres-forts ;
- Une image `/root/schematics.png` décrivant un circuit logique ;
- Un fichier dont le nom est explicite sur son utilité : `/root/get_safe1_key.py`.

Un serveur `SSH` est lancé, nous permettant d'échanger simplement des fichiers entre la machine hôte et le système lancé avec `qemu`.

2.2 Des portes, beaucoup de portes

Le script `get_safe1_key.py` définie une fonction `secure_device` qui est utilisée à de nombreuses reprises dans le processus de génération de la clé du premier coffre-fort. Malheureusement, elle n'est pas complète comme l'indique sa documentation :

```
def secure_device(a,b,op):
    out = 0
    """
    TODO :
            - Implémentation de la communication avec le secure element
                    * Entrées A et B (A0 = bit de poids faible)
                    * Entrée OP
                    * Sortie Out (Out0 = bit de poids faible)
                    * Les boutons permettent à l'utilisateur de rentrer sa
                    combinaison secrète pour le déchiffrement
            - Supprimer les docs de conception
    """
    return out
```

La première étape est donc de réécrire cette fonction à partir de la documentation fournie, c'est-à-dire à partir de l'image `schematics.png` (voir Figure 2). Les annotations en rouge ne sont pas présentes dans le fichier original

Cette fonction prend deux entrées sur 8 bits (a et b), une entrée sur 2 bits (op) ainsi qu'une entrée sur 4 bits (correspondant à la combinaison de boutons appuyées). La sortie est sur 8 bits.

On peut décomposer ce circuit en quatre étages distincts présentés sur les

— L'étage 1, contrôlé par le bouton B3, qui agit sur l'entrée a ;
— L'étage 1b, contrôlé par le bouton B4, qui agit sur l'entrée b ;
— L'étage 2, qui combine op ainsi les boutons B1 et B2 ;
— L'étage 3, qui combine la sortie des étages 1, 1b, et 2.

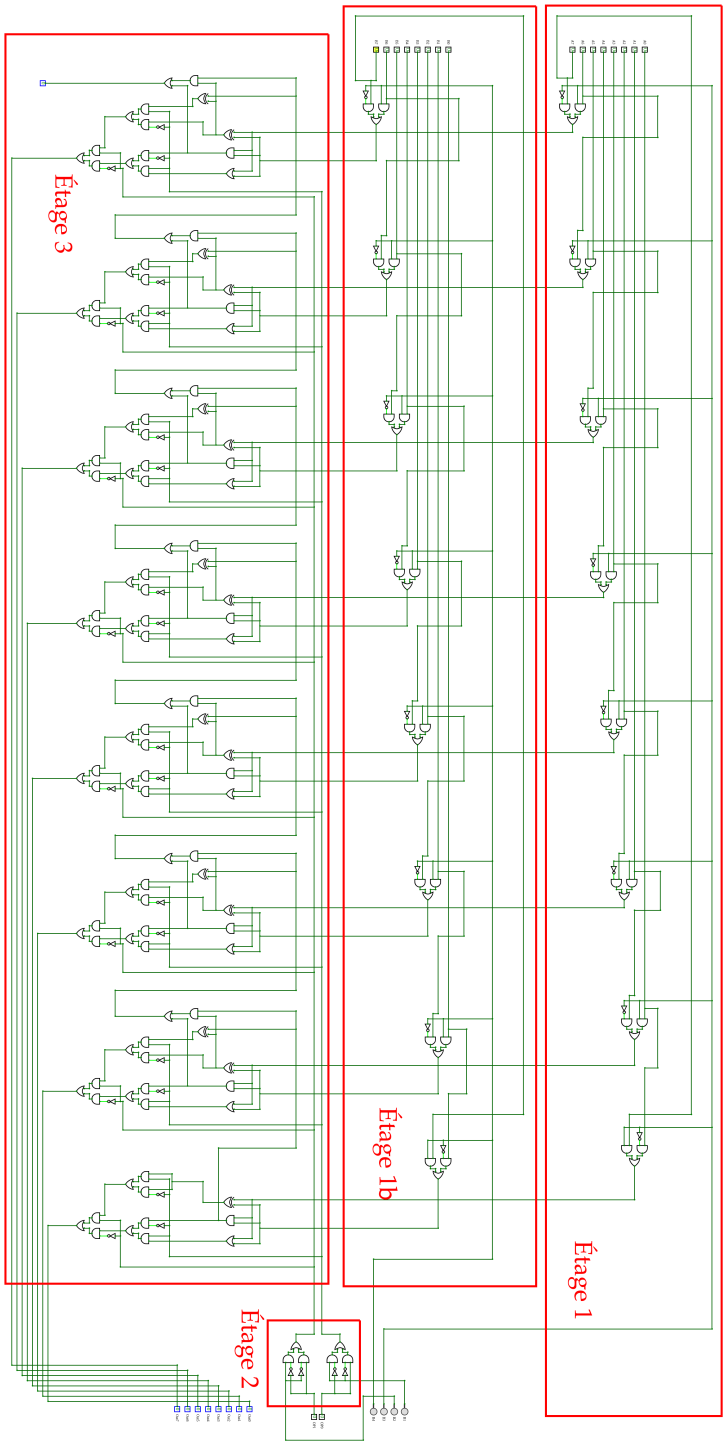

FIGURE 2 – Schema de la fonction `secure_device`

2.2.1 Sélecteur

Le sous-circuit présenté en Figure 3 apparait souvent. Il correspond en réalité à une opération de sélection : selon la valeur de s, le circuit renvoie soit a, soit b.

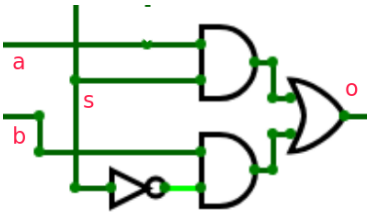

FIGURE 3 – Sous-circuit de sélection

2.2.2 Ou Exclusif

Le sous-circuit présenté en Figure 4 correspond au xor entre a et b.

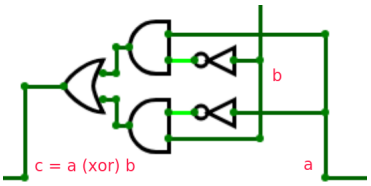

FIGURE 4 – Sous-circuit Ou Exclusif

2.2.3 Unité Arithmétique et Logique

Le sous-circuit présenté en Figure 5 correspond à une sous-partie d'une Unité Arithmétique et Logique. À l'aide de sélecteur la sortie correspond à l'opération a ET b, a OU b, a XOR b. De plus la partie de gauche du sous-circuit permet de propager (ou non) la retenue. Ainsi, ce sous-circuit peut aussi calculer une addition avec retenue entre a et b.

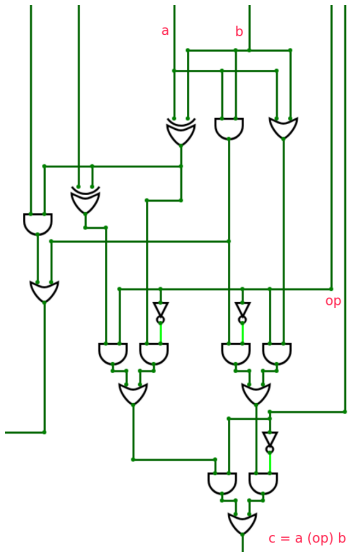

FIGURE 5 – Sous-circuit UAL

2.2.4 Étage 1 et Étage 1b

Ces deux étages renvoient a (respectivement b) ou une rotation circulaire de a (respectivement b) en fonction de la valeur de B3 (respectivement B4). La valeurs de sortie est appelée a' (respectivement b').

2.2.5 Étage 2

Cet étage renvoie op_1 XOR B1 et op_2 XOR B2, avec op_1 le premier bit de op et op_2 le deuxième bit de op. La valeur de sortie est appelée op'.

2.2.6 Étage 3

Cet étage renvoie le résultat de l'opération a' $[op']$ b' où $[op']$ désigne l'opération, définie par op', parmi l'addition, le OU bit à bit, le ET bit à bit, ou le OU EXCLUSIF bit à bit.

2.2.7 Implémentation de `secure_device`

Une fois le circuit compris dans sa globalité, il devient assez facile d'en faire une implémentation en très peu de temps. Le code peut être retrouvé en Annexe B.1.

2.3 Des boutons, beaucoup de boutons

Maintenant que `secure_element` est correctement implémenté, il ne nous reste plus qu'à l'utiliser pour débloquer la clé du premier coffre-fort. Pour ce faire, il faut trouver les huit bonnes combinaisons de boutons. Chaque combinaison est ensuite manipulée au travers de 8 fonctions différentes. Pour l'exemple, voici la première :

```
def step1(buttons):
    r = secure_device(buttons, 0x35,0x27,3)
    r = secure_device(buttons, 0x7e,r,3)
    r = secure_device(buttons, 0x66,r,2)
    r = secure_device(buttons, 0x8,r,1)
    r = secure_device(buttons, 0x13,r,0)
    r = secure_device(buttons, 0x1f,r,1)
    r = secure_device(buttons, 0xa,r,2)
    r = secure_device(buttons, 0xd3,r,0)
    r = secure_device(buttons, 0xc6,r,3)

    return r
```

La sortie de chaque fonction est un entier entre 0 et 255 (8 bits d'entropie supposées) mais dépendant seulement des boutons (4 bits d'entropie). Les 8 sorties sont ensuite concaténées, puis hachées en utilisant `SHA256` et enfin le condensat obtenu est comparé à un condensat de référence : `00c8bb35d44dcbb2712a11799d8e1316045d64404f337f4ff653c27607f436ea` en hexadécimal.

Pour optimiser l'attaque par force brute qui se profile, on précalcule pour chacune des étapes et pour chacune des 16 combinaisons de boutons différents la sortie de `stepi`. Il y a en tout, $16 \times 8 = 128$ valeurs à calculer, ce qui se fait très rapidement. Ensuite on essaye les $16^8 = 2^32$ combinaisons possible en entrée de la fonction de hachage. Pour accélérer le processus, il est possible de faire tourner plusieurs instances en parallèle. Au bout d'un peu plus d'une heure, la bonne combinaison est trouvée et on obtient donc le jalon de cette étape.

Jalon 2. *SSTIC{5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a}*

3 *Un nain-génieux programme*

Le premier coffre-fort est déverrouillé. Il contient un fichier `decrypted_file`.

```
# file decrypted_file
decrypted_file: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0,
BuildID[sha1]=5b5be1337d13c986d0e21441d771a36e41a34d17, stripped

# ./decrypted_file
Usage : ./decrypted_file <flag>

# strings decrypted_file  | grep SSTIC
SSTIC{congolexicomatisation}
```

La dernière chaîne de caractère est évidemment un leurre.

3.1 Un programme beaucoup trop petit

On a donc sous les yeux, un binaire `ARM` lié dynamiquement à ses bibliothèques et dont les symboles ont été décapés (*stripped*).

`ghidra` est utilisé pour désassembler et même décompiler le binaire. Tous les extraits de code ressemblant à du `C` à partir de maintenant seront issus de l'outil de décompilation de `ghidra`.

Le binaire qui nous intéresse ici semble, de premier abord, plutôt simple. La fonction `main` affiche la chaîne de caractère `"Usage : ..."` s'il n'y a pas assez d'argument, sinon appelle `true_main` avec le premier argument en paramètre.

```
undefined8 main(int argc,undefined8 *argv)

{
  if (argc != 2) {
    printf("Usage : %s <flag>\n",*argv);
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
                    /* try { // try from 00402eb4 to 00402eb7 has its
                    CatchHandler @ 00402f20 */
  true_main(argv[1]);
  return 0;
}
```

La fonction `true_main` crée une exception à partir de la chaîne donnée en argument, puis la lance.

```
void true_main(undefined8 argv)

{
  undefined8 *except;

  except = (undefined8 *)__cxa_allocate_exception(8);
  *except = argv;
                    /* WARNING: Subroutine does not return */
  __cxa_throw(except,typeinfo,0);
}
```

En suivant l'exécution du programme dans un débogueur (comme `gdb` par exemple), on s'aperçoit qu'au moment où le programme revient exécuter des instructions dans le programme lui-même, il est déjà trop tard : la valeur d'un registre est telle que le programme affiche `"Not good"` puis quitte. Tout semble donc se passer dans la `libc` directement.

3.2 On s'équipe (mal) !

Après de longues recherches dans la `libc` fournie et une poignée de "watchpoint" posés grâce à `gdb`, on trouve l'existence d'une machine virtuelle dont l'interpréteur commence à l'adresse

0x10b8b4 de la `libc`. En utilisant la fonction `dump memory` de `gdb` il est possible de récupérer les instructions qui seront exécutées par cet interpréteur.

Les instructions de la machine virtuelle interagissent avec une pile pour charger ou décharger des valeurs immédiates, combiner les deux valeurs du sommet de la pile en utilisant une opération (somme, ET logique, ...), ou bien intervertir les deux ou trois valeurs du sommet de la pile. Il y a aussi des mécanismes permettant de sauter (conditionnellement ou non) à une adresse dans le bytecode.

Pour voir de manière plus claire ce que fait le programme, il a été nécessaire de créer deux scripts :

- Un désassembleur pour "traduire" le bytecode en des instructions un peu plus facilement compréhensible (cf. Annexe C.1) ;
- Un décompilateur pour "traduire" un sous ensemble des instructions en sortie de désassembleur en `python` pour pouvoir émuler le comportement de l'interpréteur (cf. Annexe C.2).

3.3 Structure globale

Une fois qu'il est possible de désassembler et émuler des portions de bytecode, le gros du travail commence. S'aidant du débogueur affichant l'état courant de la pile et la prochaine instruction à exécuter d'une part, et d'autre part le code désassemblé, on essaye de comprendre chaque bloc de code se trouvant entre un point d'entrée et un saut. Il y a parfois ce qui s'apparente à des appels de fonction, parfois des boucles "*tant que*". Avoir en tête quelles sont les entrées et les sorties de tels blocs de code est crucial mais s'avère parfois relativement corsé (à cause notamment, de la structure en pile).

Un bloc de code était particulièrement piégeur : très long, et particulièrement indigeste, il ne renvoyait cependant que deux valeurs différentes en fonction de l'entrée.

Outre ce morceaux de code, la structure s'apparente à un réseau de Feistel sans clé appliqué à la chaîne de caractère fournie en entrée du binaire principal (cf. Figure 6). En fin de course, le résultat est comparé à une chaîne de caractère fixée et le résultat de la comparaison était renvoyé au programme principal.

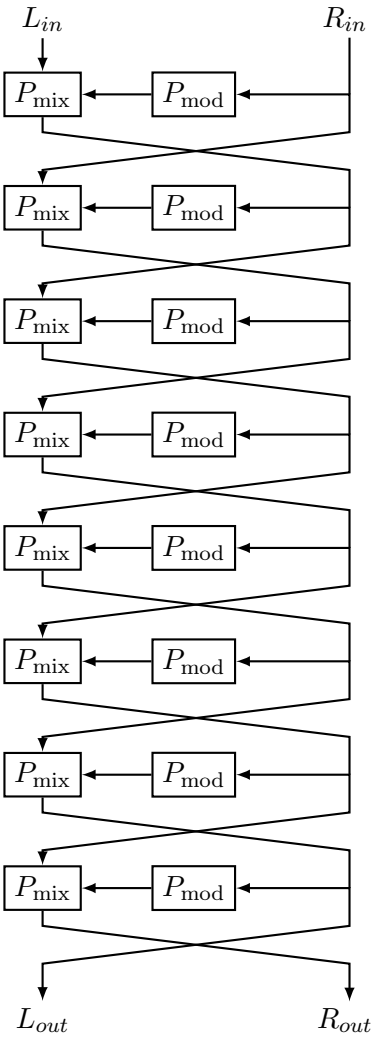

FIGURE 6 – Structure globale (adapté d'un schéma de Jérémy Jean sur `TikZ for Cryptographers`)

3.4 Un réseau pas comme les autres

À partir d'une telle structure, il ne manque plus qu'à inverser la fonction P_{mix}. Dans un réseau de Feistel "classique", celle-ci est simplement un Ou Exclusif bit à bit (et est donc son propre inverse). Dans notre cas, elle est un peu plus compliqué (mais reste évidemment inversible).

Les fonctions P_{mix} et P_{mod} sont en réalité conçues en itérant 15 fois une fonction de tour. Elles reposent toutes les deux sur des constantes stockées en mémoire qu'il ne faut donc pas oublier d'extraire.

Une fois la fonction P_{mix}^{-1} et P_{mod} écrites en `python`, il ne reste plus qu'à partir de la sortie attendue puis de remonter jusqu'à l'entrée attendue.

Ce faisant, nous obtenons un nouveau jalon.

Jalon 3. *SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}*

3.5 Pour aller plus vite...

Pour gagner quelques précieuses heures de travail sur ce challenge, on aurait pu se rendre compte que le bytecode intermédiaire ainsi que la machine l'interprétant n'était en réalité pas quelque chose inventé par le créateur du challenge mais bien quelque chose déjà existant sous le nom de `DWARF` et présent "naturellement" dans la `libc`.

4 *Un programme d'exception*

La validation du jalon précédent permet de déverrouiller le deuxième coffre-fort. Après le déchiffrement du deuxième coffre-fort, un message plutôt curieux s'affiche :

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

Effectivement, après redémarrage, le message qui était précédemment :

```
ERROR:   Secure-OS not not available : need decryption key
```

s'est transformé en un :

```
NOTICE:  BL31: Initializing BL32
NOTICE:  Booting Secure-OS
```

Ainsi on peut s'attendre à ce que cette étape du challenge utilise des mécaniques portant sur le fonctionnement interne du système d'exploitation.

Le fichier déchiffré s'appelle à nouveau `decrypted_file`.

```
# file decrypted_file
decrypted_file: ELF 64-bit LSB executable, ARM aarch64, version 1 (GNU/Linux),
statically linked, for GNU/Linux 3.7.0, stripped
# ./decrypted_file
usage: ./decrypted_file [32-bytes-key-hex-encoded]
# ./decrypted_file 0000000000000000000000000000000000000000000000000000000000
000000
Loose
```

4.1 Binaire fantoche

Cette fois ci nous avons à faire à un binaire lié statiquement à ses bibliothèques et dont les symboles ont été, ici aussi, décapés (*stripped*).

Le résultat de la décompilation par `ghidra` de la fonction principale de ce programme nous permet d'y voir un peu plus clair. Celle-ci peut être retrouvé en Annexe D.1 (certains noms de fonction et de variable ont été modifiées manuellement).

On s'aperçoit que ce programme ne fait en réalité pas grand chose. Après une vérification sommaire des arguments, il appelle deux fois `ioctl` puis boucle. Dans cette boucle, le programme appelle deux fois `ioctl` et la valeur de retour du deuxième permet de décider si la boucle s'arrête et si le mot de passe est validé.

On peut confirmer ce comportement en utilisant `strace` :

```
# strace ./decrypted_file 00000000000000000000000000000000000000000000000000000
0000000000

execve("./decrypted_file", ["./decrypted_file",
"00000000000000000000000000000000"...], 0xffffe1339d38 /* 11 vars */) = 0
brk(NULL)                               = 0x2fd14000
brk(0x2fd14f80)                         = 0x2fd14f80
uname({sysname="Linux", nodename="sstic", ...}) = 0
readlinkat(AT_FDCWD, "/proc/self/exe", "/root/safe_02/decrypted_file",
4096) = 28
brk(0x2fd35f80)                         = 0x2fd35f80
brk(0x2fd36000)                         = 0x2fd36000
openat(AT_FDCWD, "/dev/sstic", O_RDONLY) = 3
ioctl(3, SEV_ISSUE_CMD, 0xffffefce7fd8) = 0
ioctl(3, _IOC(_IOC_READ|_IOC_WRITE, 0x53, 0x1, 0x10), 0xffffefce7fd8) = 0
ioctl(3, _IOC(_IOC_READ|_IOC_WRITE, 0x53, 0x2, 0x10), 0) = 0
ioctl(3, SNDRV_SEQ_IOCTL_RUNNING_MODE, 0) = 0
ioctl(3, _IOC(_IOC_READ|_IOC_WRITE, 0x53, 0x2, 0x10), 0) = 0
ioctl(3, SNDRV_SEQ_IOCTL_RUNNING_MODE, 0) = 0
ioctl(3, _IOC(_IOC_READ|_IOC_WRITE, 0x53, 0x2, 0x10), 0) = 0
ioctl(3, SNDRV_SEQ_IOCTL_RUNNING_MODE, 0) = 0
ioctl(3, _IOC(_IOC_READ|_IOC_WRITE, 0x53, 0x2, 0x10), 0) = 0
ioctl(3, SNDRV_SEQ_IOCTL_RUNNING_MODE, 0) = 0
ioctl(3, _IOC(_IOC_READ|_IOC_WRITE, 0x53, 0x2, 0x10), 0) = 0
ioctl(3, SNDRV_SEQ_IOCTL_RUNNING_MODE, 0) = 0
ioctl(3, _IOC(_IOC_READ|_IOC_WRITE, 0x53, 0x2, 0x10), 0) = 0
ioctl(3, SNDRV_SEQ_IOCTL_RUNNING_MODE, 0) = 0
ioctl(3, _IOC(_IOC_READ|_IOC_WRITE, 0x53, 0x2, 0x10), 0) = 0
ioctl(3, SNDRV_SEQ_IOCTL_RUNNING_MODE, 0) = 0
```

4.2 Noyau raffiné

Ainsi on confirme que le programme communique effectivement à de multiple reprise avec un module du noyau via l'appareil nommé `/dev/sstic`. On retrouve le fichier nommé `sstic.ko` qui est le binaire décrivant le module. Lors de la désassemblage, on s'aperçoit que le module lui-même n'est pas très fourni et qu'il utilise une instruction très spécifique : `smc`. Cette instruction permet de générer une exception qui sera gérée par le plus haut niveau d'exception de l'architecture `ARM` : le *Secure Monitor*.

Ainsi c'est une partie spécifique du système d'exploitation qui va traiter cette exception. À partir de ce moment là, il faut donc faire de la rétro-ingénierie du système d'exploitation lui-même (*Secure OS*) et l'utilisation de `gdb` pour déboguer `qemu` (et non plus un programme lancé à l'intérieur du téléphone virtuel) s'avère très utile. De plus, toute la partie intéressante ne se passe pas qu'au niveau d'exception EL3, mais aussi à d'autres niveaux.

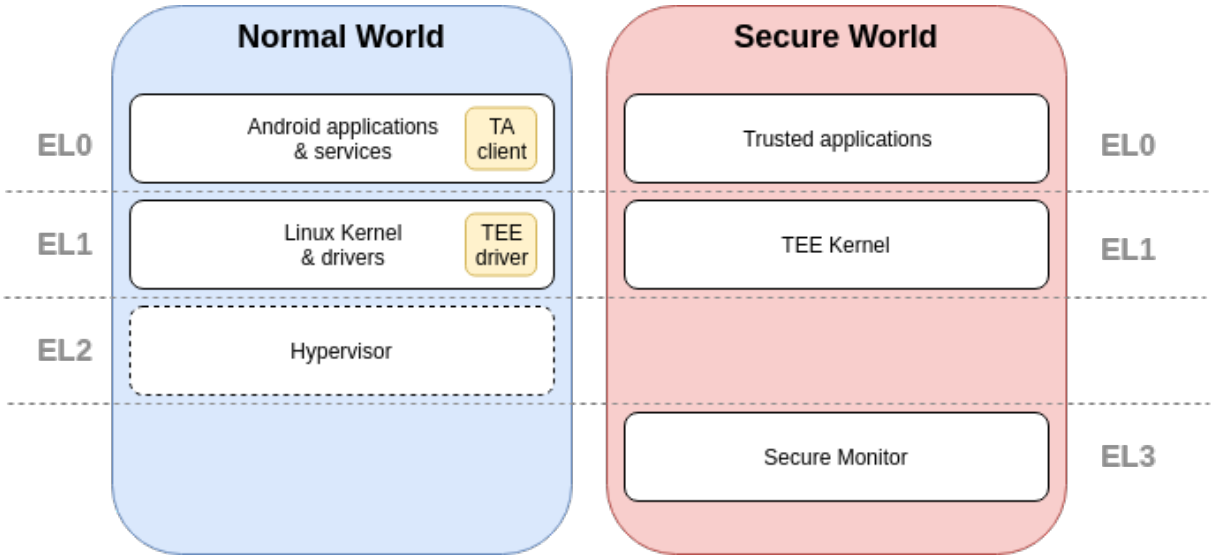

FIGURE 7 – Schéma de l'architecture TrustZone de `ARM` (schéma par David Bérard sur `le blog de Synacktiv`

4.3 Mémoire piégée

Il existe une chaîne de caractère plus que suspecte présente dans une partie de la mémoire :

```
(ps -ef |egrep -q 'qemu-system.* -[s] |qemu-system.* -[g]db ') && exit 42
```

C'est une commande `sh` permettant de lister les processus courant et de quitter s'il en existe un de la forme `'qemu-system.* -[s] |qemu-system.* -[g]db '`, c'est-à-dire s'il y a une instance de `qemu` en train d'être déboguée.

Même si la manière dont elle est utilisée n'est pas très claire (car celle-ci n'est potentiellement exécutée qu'à l'intérieur de `qemu`), inverser le flot d'exécution à la sortie de la fonction modifier une zone mémoire et permettra ultimement de résoudre le challenge (impossible sans ce changement).

4.4 Mémoire masquée

Après de longues heures à chercher comment l'argument du programme est validé (ou non) par celui-ci, un comportement étrange attire notre attention. En effet, une certaine zone mémoire (notamment celle modifiée par la contre-mesure décrite précédemment) est utilisée de manière inhabituelle : la lecture (respectivement écriture) est suivie (respectivement précédée) par l'application d'une instruction vectorielle à la donnée : `sm4e`. Celle-ci correspond à une fonction de chiffrement du nom de `SM4`. Crée par le gouvernement chinois, cette fonction de chiffrement est en réalité un standard dans ce pays.

Un script nous permet de déchiffrer la zone mémoire en question (cf. Annexe D.2). La suite des investigations nous permet de découvrir ce que cache en réalité cette zone mémoire : du code pour une machine virtuelle personnalisée qui va modifier l'"entrée" (chaîne de caractère passée

en argument), des octets nuls, beaucoup de données utilisée comme constante lors de l'exécution du code modifiant notre entrée, la chaîne de caractère "objectif" (celle à laquelle sera comparée notre entrée après modification), et finalement la l'"entrée" en cours de modification. Nous nous attelons donc au désassemblage du programme pour comprendre ce que subis notre "entrée" avant d'être comparée à l'"objectif".

4.5 Programme rusé

Le morceau de programme interprétant le code de la machine virtuelle découverte précédemment n'est pas très long (147 instructions) et a le bon goût d'avoir des instructions de longueur fixe (3 octets). Cela facilite l'écriture d'un désassembleur. En regardant comment son manipulées les données on se rend compte que la machine virtuelle manipule des registres mais que ceux-ci sont protégés par une nouvelle contre-mesure.

4.5.1 Registres obfusqués

Tout comme la zone mémoire "masquée", la zone mémoire qui est utilisé en tant que registres n'est pas accédée et écrite directement. À chaque lecture (respectivement écriture) d'un registre, la donnée est déobfusquée (respectivement obfusquée) en utilisant l'instruction `aese` (respectivement `aesd`). Cette instruction correspond l'application d'un round d'AES (Advanced Encryption Standard) dans un sens ou dans l'autre. Ainsi, le compréhension de ceci permet d'en apprendre plus sur la quantité et le rôle des registres : il y a seize registres dont utilisé comme compteur de programme, c'est-à-dire contenant l'adresse de la prochaine instruction à charger.

À partir de l'information du bytecode en cours d'exécution par la machine virtuelle ainsi que la connaissance des modifications de la donnée dans un ou plusieurs registres, on reconstitue la signification d'une majorité des instructions. Plus le nombre d'instructions correctement décodées augmente, plus la structure du code devient claire et permet d'émettre des hypothèses facilitant la compréhension de nouvelles instructions.

Une poignée d'instructions résiste néanmoins et il devient indispensable de plonger plus profondément dans la partie du code `ARM` interprétant ces instructions.

4.5.2 Binaire à l'envers

Les morceaux de binaire `ARM` interprétant les instructions résistantes à une analyse légère, telle que présentée précédemment, ont tous un point commun : une fonction est appelée avec en paramètre une adresse pointant vers de la mémoire qui n'est pas vide mais dont le contenu n'est pas reconnu par `ghidra` (pas de chaîne de caractère, pas d'instructions détectées, ...). La dernière instruction `ARM` dans cette fonction est l'instruction `eret`. Cette instruction, similaire à `ret`, signe la fin du traitant d'exception. Nous vous rappellons qu'une majorité du binaire `ARM` dont on parle depuis le début est exécuté dans un niveau d'exception strictement supérieur à 0. Mais curieusement, après cette instruction, `gdb` ne comprend plus ce qu'il se passe, l'affichage reste bloqué sur cette instruction, jusqu'au moment où nous sommes de retour à ce niveau d'exception et que tous les calculs intéréssant (ie. nous permettant de comprendre ce que fait réellement l'instruction de la machine virtuelle) sont terminés.

En réalité, les instructions `ARM` précédant `eret` ont un rôle très particulier : elles modifient des registres de contrôle spécifique au niveau d'exception actuel (EL1). Les modifications suivantes

Bit	Name	Fonction
[31]	N	Negative
[30]	Z	Zero
[29]	C	Carry condition code flag
[28]	V	Overflow condition code flag
[27]	Q	Cumulative saturation bit
[26 :25]	IT[0 :1]	If-Then execution state bits for the Thumb IT (If-Then) instruction
[24]	J	Jazelle bit
[19 :16]	GE	Greater than or Equal flags
[15 :10]	IT[7 :2]	If-Then execution state bits for the Thumb IT (If-Then) instruction
[9]	E	Endianness execution state bit : 0 - Little-endian, 1 - Big-endian
[8]	A	Asynchronous abort mask bit
[7]	I	IRQ mask bit
[6]	F	FIRQ mask bit
[5]	T	Thumb execution state bit
[4 :0]	M	Mode field

TABLE 1 – Description de la fonction des bits de `CPSR`. `Source`

expliquent le comportement étrange :

— Le registre `ELR_EL1` est chargé avec l'adresse passée en paramètre de la fonction ;
— La MMU est désactivée, dont l'utilité sera décrite un peu plus tard (cf. 4.5.3) ;
— Le registre `SPSR_EL1` est chargé avec la valeur immédiate `0x1d0`.

`ELR_EL1` (*Exception Link Register for Exception Level 1*) est le registre contenant l'adresse de retour de l'exception en cours. Donc l'instruction suivant l'exécution de `eret` se trouve à l'adresse passée en paramètre. Mais alors, pourquoi `gdb` n'arrive pas à suivre et pourquoi `ghidra` n'arrive pas à désassembler l'instruction à cette adresse ?

La subtilité vient de la valeur de `SPSR_EL1`. En effet, `SPSR_EL1` (*Saved Program Status Register for Exception Level 1*) est le registre contenant la valeur à mettre dans le registre `CPSR` après l'exécution de `eret`. `CPSR` (*Current Program Status Register*) est un registre contenant les informations sur le statut actuel du processeur. Pour une description exacte, voir Table 1. Quand `0x1d0` est chargé dans le registre, la valeur du bit 9 passe de 1 à 0, ayant pour effet... de changer, après l'instruction `eret`, le boutisme du processeur de grand-boutisme (*big-endian*) à petit-boutisme (*little-endian*). Ainsi, `ghidra` et `gdb` ne s'attendant pas à un tel comportement, décode mal les instructions. Il ne reste plus qu'à forcer le boutisme du désassemblage des instructions aux adresses qui nous intéresse pour découvrir ce qui est exécuté lors de ces instructions.

4.5.3 Programme minuté

Deux des instructions que nous n'avions pas réussi à comprendre avant de découvrir la technique expliquée ci-dessus forment en réalité une contre-mesure anti-deboguage. Grâce au fait que la MMU soit désactivée, la première instruction écrit à l'adresse `0x9010008` tandis que la deuxième instruction modifie le flot d'éxecution si la valeur lu à l'adresse `0x9010000` est supérieure à 5. En réalité, ces adresses correspondent à des adresses permettant d'utiliser le mécanisme d'Horloge Temps Réel (`Real Time Clock`) tel que décrit dans `cette documentation`. Ainsi, la

première instruction initialise l'horloge, et la seconde modifie le flot d'exécution si celle-ci prend trop de temps.

4.5.4 Un objectif mouvant

La dernière instruction qu'il reste à comprendre peut aussi être considérée comme une contre-mesure. En effet, la chaîne de caractère "objectif" n'est en réalité pas constante entre le début et la fin du programme de la machine virtuelle. Il faut donc faire attention à considérer comme "objectif" la chaîne de caractère finale qui est "`pr.a.rfg.cnf.fv.snpvyr@ffgvp.bet`" soit la chaîne "`ce.n.est.pas.si.facile@sstic.org`" où chaque lettre a été décalée de treize positions dans l'alphabet (*rot13*).

Ceci est évidemment un leurre.

4.6 Inversion corsée

Maintenant que toutes les instructions sont comprises, il est possible de désassembler le code interprété par la machine virtuelle dans un language ressemblant à un language d'assemblage "classique". Ce code désassemblé peut être trouvé en Annexe D.4 et a été produit à partir du désassembleur donné en Annexe D.3.

Ensuite, ce code est émulé en `python` en définissant une fonction `f` (cf. Annexe D.5).

Ainsi cette fonction `f` de transformation de la chaîne en "entrée" peut se voir comme une variation d'un réseau de Feistel à quatre branches de 16 bits avec une profondeur de 32. Ce réseau est appliqué indépendemment sur les quatres parties de 64 bits (8 octets) composant l'"entrée". Pour chacune des 32 itérations, le mot de 64 bits est découpé en 4 sous-mots r_0, r_1, r_2, r_3. La valeur de r_1 est utilisée pour calculer une nouvelle valeur notée r_9, qui servira de valeur pour r_0 à l'itération suivante. Nous appelons `CASCADE` cette fonction de transformation de r_1.. Ensuite, selon l'ordinalité de l'itération, les valeurs de r_1, r_2, et r_3 sont actualisées de deux manières différentes, utilisant le `Ou Exclusif` bit à bit.

Seule la fonction `CASCADE` est difficile à inverser, et pour cause, elle dépend très fortement d'un tableau de constantes et elle n'est pas à proprement dite "inversible". En effet, si les constantes ne sont pas "correctement" choisies, la fonction `CASCADE` n'est ni surjective, ni injective (deux messages en entrée différents peuvent donner la même sortie, et un message peut être la sortie de zéro, un ou plusieurs messages en entrée). Ainsi, nous avons implementé l'inversion de la fonction `f` (voir en Annexe D.6) pour être sûr de trouver efficacement toutes les pre-images à partir de la donnée d'une sortie, ie. pour tout y les (x_i), s'ils existent, tel que $\texttt{f}(x_i) = y$.

Si la contre-mesure décrite en 4.3 n'est pas correctement contournée, l'inversion de la chaîne "objectif" est impossible : la valeur des constantes n'étant pas les bonnes, la chaîne "objectif" n'a pas de pre-image dans ce contexte. Après avoir contourné toutes les contremesures et en appliquant la fonction `inv_f` à la chaîne "objectif", nous obtenons une entrée nous menant à la victoire. Nous débloquons ainsi le jalon permettant de dévérouiller le dernier coffre-fort.

Jalon 4. *SSTIC{acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e}*

5 *Le grep final*

Le dernier coffre-fort étant déverrouillé, il ne reste plus qu'à trouver l'adresse courriel finale. Le fichier déchiffré est en réalité une archive contenant une partie d'un système de fichier d'un téléphone mobile. Celui-ci contient notamment des échanges de messages préparant un complot contre le SSTIC. Trouver l'adresse courriel recherchée n'est plus qu'une question de choisir la bonne expression régulière à donner en paramètre de `grep`.

```
# grep -ri -aohE '([a-zA-Z0-9_\-\.]+)@challenge.sstic.org' data/ | head -1
9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org
```

Ainsi, nous obtenons le tout dernier jalon nous permettant de terminer le challenge SSTIC 2019.

Jalon 5. *9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org*

Conclusion

Ainsi se termine le challenge SSTIC 2019. Ce rapport n'est sûrement pas exempt d'erreurs. Si vous en trouvez ou si vous avez toutes autres remarques, vous pouvez me contacter en envoyant un courriel à l'adresse nics@grehack.fr ou en venant me parler lors de l'édition 2019 du SSTIC.

Remerciements

Je tiens à remercier les concepteurs des différentes étapes ainsi que de la globalité du challenge SSTIC 2019. Chaque challenge fût intéressant et divertissant, malgré la difficulté toujours plus grande. Loin d'être un habitué des challenges de rétro-ingénierie, ils m'ont permis de me mesurer à un défi duquel je ressors avec des nouvelles connaissances.

Plus particulièrement, je tiens à remercier David Bérard, concepteur de la quatrième étape, pour m'avoir fait découvrir de nombreuses mécaniques qui m'étaient jusqu'alors inconnues.

Par manque de temps à allouer à l'écriture de ce rapport ainsi que par manque de connaissances, celui-ci ne reflète malheureusement que de manière incomplète la complexité des rouages rendant ce challenge aussi difficile que instructif.

Je remercie aussi ceux qui, grâce à leur travail de documentation, création d'outils ou grâce à leurs discussions, m'ont permis de résoudre ce challenge.

A Ressource pour la première étape

A.1 Script pour l'Analyse Simple de la Consommation

```
#!/usr/bin/python

import matplotlib.pyplot as pl
import numpy as np


def get_end_start(thr):
    loc = np.where(int_tr < thr)

    # Get actual list
    loc = loc[0]
    bc = 0
    start = [loc[0]]
    end = []
    for i in range(0, len(loc)-1):
        delta = loc[i+1]-loc[i]
        if delta < 100:
            continue
        end.append(loc[i])
        start.append(loc[i+1])
    return start, end


def simple_power_analysis(start, end, width_thr):
    d = []
    for i in range(len(end)):
        delta = end[i] - start[i]
        if delta > width_thr:
            b = 1
        else:
            b = 0
        d.append(str(b))
    return d


if __name__ == "__main__":
    trace = np.load("power_consumption.npz")["arr_0"]
    int_tr = trace[700000:1490000]  # Select only the interesting part

    thr = -0.05  # Threshold to identify peaks
    start, end = get_end_start(thr)
```

```
    print("There are {} peaks.".format(len(end)))

    width_thr = 300  # Threshold to distinguish between Type A and Type B peaks
    d = simple_power_analysis(start, end, width_thr)
    d = d[::-1]  # LSB first
    d_int = int(''.join(d), 2)
    print("The RSA private exponent is:\n{}".format(hex(d_int)[2:]))
```

B Ressource pour la deuxième étape

B.1 Implémentation de secure_device

```
def secure_device(buttons, a,b,op):
    b1, b2, b3, b4 = buttons
    new_op = op ^ (b1 | (b2 << 1))
    new_a = cond_rot(a, b3) # Rotation conditionnée par b3
    new_b = cond_rot(b, b4) # Rotation conditionnée par b4

    if new_op == 0:
        return new_a & new_b
    elif new_op == 1:
        return new_a | new_b
    elif new_op == 2:
        return new_a ^ new_b
    else:
        return (new_a + new_b) & 0xff
```

C Ressources pour la troisième étape

C.1 Désassembleur DWARF "maison"

```
#!/usr/bin/python

def binary_op(op):
    if op == 0x1a:
        print('s_below & s_top')
    elif op == 0x1b:
        print("s_below/s_top (or 0)")
    elif op == 0x1c:
        print("s_below - s_top")
    elif op == 0x1d:
        print("s_below % s_top")
```

```
    elif op == 0x1e:
        print("s_below * s_top")
    elif op == 0x21:
        print("s_below | s_top")
    elif op == 0x22:
        print("s_below + s_top")
    elif op == 0x24:
        print("s_below << (s_top & 0x3f)")
    elif op == 0x25:
        print("s_below >> (s_top & 0x3f) (unsigned)")
    elif op == 0x26:
        print("s_below >> (s_top & 0x3f) (signed)")
    elif op == 0x27:
        print("s_below ^ s_top")
    elif op == 0x29:
        print("s_below == s_top")
    elif op == 0x2a:
        print("s_below >= s_top")
    elif op == 0x2b:
        print("s_below > s_top")
    elif op == 0x2c:
        print("s_below <= s_top")
    elif op == 0x2d:
        print("s_below < s_top")
    elif op == 0x2e:
        print("s_below != s_top")


def op_with_stack(op, pc):
    print("pop $tmp; ", end='')

    if (op == 0x1f):
        print("push -$tmp")
    else:
        if (op < 0x20):
            if (op == 6):
                print("push [$tmp]")
            else:
                if (op != 0x19):
                    print("ERROR op not 0x19")
                    exit()
                print("push abs($tmp)")
        else:
            if (op == 0x23):
                print("tmp2 = WEIRD FUNC on pc ({})".format(pc))
```

```
                binary_op_tmp(op)

            if (op == 0x94):
                op2 = prog[pc + 1]
                pc = pc + 2
                if op2 == 1:
                    print("push (byte)[$tmp] (1o)")
                elif op2 == 2:
                    print("push (ushort)[$tmp] (2o)")
                elif op2 == 4:
                    print("push (uint)[$tmp] (4o)")
                elif op2 == 8:
                    print("push (longlong)[$tmp] (8o)")

            else:
                if (op != 0x20):
                    print("ERROR op not 0x20")
                    exit()
                print("push !$tmp")


# Not really quick but most certainly dirty

if __name__ == "__main__":
    with open("prog.bin", "rb") as f:
        prog = f.read()

    pc = 0
    prog = b'0' * 0x400258 + prog

    pc = 0x400258  # entry
    pc = 0x400ab5  # func2
    pc = 0x400d7d  # func3 et func4
    pc = 0x4005f0  # func5
    pc = 0x4003bc
    pc = 0x400e88  # func6
    jmp_break = True
    new_pc = 0
    while True:
        #new_pc = input("New addr? ")
        if new_pc:
            pc = int(new_pc, 16)
            new_pc = 0
        if pc > 0x403216:
            print('END OF PROGRAM')
```

```
            exit()
        op = prog[pc]
        print("{}:\t{}\t".format(hex(pc), hex(op)), end="")

        if (op < 0x21):
            if (op < 0x1f):
                if (op == 0x10):
                    print("push WEIRD_FUNC({})".format(pc + 1))
                    pc = pc + 1
                else:
                    if (op < 0x11):
                        if (op == 10):
                            print("push #{}".format(prog[pc + 1:pc + 3]))
                            pc = pc + 3
                            continue
                        else:
                            if (op < 0xb):
                                if (op == 6):
                                    op_with_stack(op, pc)
                                    pc = pc + 1
                                    continue
                                if (op < 7):
                                    if (op != 3):
                                        print("ERROR op not 3")
                                        pc = pc + 1
                                        continue
                                        exit()
                                    print("push #{}".format(
                                        hex(int.from_bytes(prog[pc + 1:pc + 9], 'little'))))
                                    pc = pc + 9

                                else:
                                    if (op == 8):
                                        print("push #{}".format(
                                            hex(int.from_bytes(prog[pc + 1:pc + 2], 'little'))))
                                    else:
                                        if (op != 9):
                                            print("ERROR op not 9")
                                            pc = pc + 1
                                            continue
                                            exit()
                                        print("push #{}".format(
                                            hex(int.from_bytes(prog[pc + 1:pc + 2], 'little', s
                                    pc = pc + 2
```

```
                        else:
                            if (op == 0xd):
                                print("push #{}".format(
                                    prog[pc + 1:pc + 5]))

                            else:
                                if (0xd < op):
                                    if ((op == 0xe) or (op == 0xf)):
                                        print("push  #{}".format(
                                            hex(int.from_bytes(prog[pc + 1:pc + 9], 'little
                                        pc = pc + 9
                                        continue

                                    print("ERROR plus de choix possible")
                                    exit()

                                if (op == 0xb):
                                    print("push #{}".format(
                                        hex(int.from_bytes(prog[pc + 1:pc + 3], 'little', s
                                    pc = pc + 3
                                    continue

                                if (op != 0xc):
                                    print("ERROR op not 0xc")
                                    exit()

                                print("push #{}".format(
                                    hex(int.from_bytes(prog[pc + 1:pc + 5], 'little'))))

                            pc = pc + 5
                            continue

                else:
                    if (op == 0x15):
                        print("push stack[{}]".format(-prog[pc + 1] - 1))
                        pc = pc + 2

                    else:
                        if (0x15 < op):
                            if (op == 0x19):
                                op_with_stack(op, pc)
                                pc = pc + 1
                                continue
                            if (0x19 < op):
                                binary_op(op)
```

```
                                pc = pc + 1
                                continue

                            if (op == 0x16):
                                print("double swap")

                            else:
                                print("triple swap")

                            pc = pc + 1
                            continue

                        if (op == 0x12):
                            print("push $top_of_stack (duplication)")
                            pc = pc + 1
                            continue

                        else:
                            if (op < 0x12):
                                if jmp_break:
                                    new_pc = input("New address? ")
                                print(
                                    "jmp ANOTHER_WEIRD_FUNC({})".format(
                                        pc + 1))
                                pc = pc + 1
                                continue

                            if (op == 0x13):
                                print("pop $tmp")
                                pc = pc + 1
                                continue

                            if (op != 0x14):
                                print("ERROR op not 0x14")
                                exit()
                            print("push $stack[-2]")

        else:
            op_with_stack(op, pc)
            pc = pc + 1
            continue

    else:
        if (op < 0x50):
            if (0x2f < op):
```

```
                print("push #{}".format(op - 0x30))
                pc = pc + 1
                continue

            if (0x27 < op):
                if (0x2e < op):
                    print("jmp {}".format(
                        hex(pc + int.from_bytes(prog[pc + 1:pc + 3], 'little') + 3)))
                    if jmp_break:
                        new_pc = input("New address? ")
                        if new_pc:
                            continue
                    pc = pc + 3

                if (0x28 < op):  # goto binary_op;
                    binary_op(op)
                    pc = pc + 1
                    continue

                print("jnz {}".format(
                    hex(pc + int.from_bytes(prog[pc + 1:pc + 3], 'little') + 3)))
                if jmp_break:
                    new_pc = input("New address? ")
                    if new_pc:
                        continue

                pc = pc + 3
                continue

            if ((op < 0x24) and (0x22 < op)):
                op_with_stack(op, pc)
                pc += 1
                continue

            binary_op(op)
            pc = pc + 1
            continue

        else:
            if (op != 0x90):
                if (op < 0x91):
                    if (op < 0x70):
                        print("push getGR({})".format(op - 0x50))
                        pc = pc + 1
                        continue
```

```
                    print(
                        "push agregate({}) + GetGR({})".format(hex(pc + 1), op - 0x70))
                    pc = pc + 1
                    continue

                else:
                    if (op == 0x94):
                        op_with_stack(op, pc)
                        pc = pc + 2
                        continue
                    if (0x94 < op):
                        if (op != 0x96):
                            if (op == 0xf1):
                                print("push SUPER_WEIRD_FUNC(param3, {},{})".format(
                                    prog[pc + 1], pc + 2))
                                pc = pc + 1
                                continue

                        print("nop")
                        pc = pc + 1
                        continue

                    if (op != 0x92):
                        print("ERROR not op 0x92")
                        exit()
                    exit()
            else:  # op == 0x90
                print("push GetGR(WEIRD2({}))".format(hex(pc + 1)))
                pc = pc + 1
                continue
```

C.2 Décompilateur "maison"

```
#!/usr/bin/python

# Not the cleanest code, but it works well enough.
# Example on the file `func7_dis` containing disassembled code (custom)

print("#0x4003cd")
print("def func7(inp):")
print("    stack = inp")
print("    print([hex(s) for s in stack])")
```

```python
with open("func7_dis", "r") as f:
    line = f.readline()
    while line:
        if line == '\n':
            line = f.readline()
            continue
        elif "push stack[" in line:
            i = line.find("[")
            offset = line[i + 1:i + 3]
            print("    stack.append(stack[{}])".format(offset))
        elif "pop $tmp" in line:
            print("    stack.pop()")
        elif "push #" in line:
            i = line.find('#')
            const = line[i + 1:-1]
            line = f.readline()
            if "-" in line:
                print("    stack[-1] = (stack[-1] - {}".format(const), end="")
            elif "+" in line:
                print("    stack[-1] = (stack[-1] + {}".format(const), end="")
            elif "<<" in line:
                print("    stack[-1] = (stack[-1] << {}".format(const), end="")
            elif ">>" in line:
                print("    stack[-1] = (stack[-1] >> {}".format(const), end="")
            elif "*" in line:
                print("    stack[-1] = (stack[-1] * {}".format(const), end="")
            elif "^" in line:
                print("    stack[-1] = (stack[-1] ^ {}".format(const), end="")
            elif "|" in line:
                print("    stack[-1] = (stack[-1] | {}".format(const), end="")
            elif "&" in line:
                print("    stack[-1] = (stack[-1] & {}".format(const), end="")
            elif "weird swap value" in line:
                print("    stack.append({})".format(const))
                print("    stack[-1], stack[-2] = stack[-2], stack[-1]")
                line = f.readline()
                continue
            elif "triple weird swap" in line:
                print("    stack.append({})".format(const))
                print(
                    "    stack[-1], stack[-2], stack[-3] = stack[-2], stack[-3], stack[-1]")
                line = f.readline()
                continue

            line = f.readline()
```

```
            if "push #0xffffffff" in line:
                line = f.readline()
                if '&' in line:
                    print("    ) & 0xffffffff")
                    line = f.readline()
                    continue
                else:
                    print("    ERROR")
                    break
            else:
                print(")")
                continue
        elif "duplication" in line:
            print("    stack.append(stack[-1])")
        elif "weird swap value" in line:
            print("    stack[-1], stack[-2] = stack[-2], stack[-1]")

        elif "triple weird swap" in line:
            print(
                "    stack[-1], stack[-2], stack[-3] = stack[-2], stack[-3], stack[-1]")

        else:
            if "-" in line:
                print("    tmp = stack.pop()")
                print("    stack[-1] = (stack[-1] - tmp", end="")
            elif "+" in line:
                print("    tmp = stack.pop()")
                print("    stack[-1] = (stack[-1] + tmp", end="")
            elif "<<" in line:
                print("    tmp = stack.pop()")
                print("    stack[-1] = (stack[-1] << tmp", end="")
            elif ">>" in line:
                print("    tmp = stack.pop()")
                print("    stack[-1] = (stack[-1] >> tmp", end="")
            elif "*" in line:
                print("    tmp = stack.pop()")
                print("    stack[-1] = (stack[-1] * tmp", end="")
            elif "^" in line:
                print("    tmp = stack.pop()")
                print("    stack[-1] = (stack[-1] ^ tmp", end="")
            elif "|" in line:
                print("    tmp = stack.pop()")
                print("    stack[-1] = (stack[-1] | tmp", end="")
            elif "&" in line:
                print("    tmp = stack.pop()")
```

```python
                print("    stack[-1] = (stack[-1] & tmp", end="")
            else:
                print("ERORORORORORORORORORRORORORORORORORROR" + line)
                line = f.readline()
                continue

            line = f.readline()
            if "push #0xffffffff" in line:
                line = f.readline()
                if '&' in line:
                    line = f.readline()
                    print("    ) & 0xffffffff")
                    continue
                else:
                    print("    ERROR")
                    break
            elif line == '\n':
                line = f.readline()
                continue
            else:
                print("    )")
                continue

        line = f.readline()

print("    print([hex(s) for s in stack])")
print("    return stack")
```

D Ressources pour la quatrième étape

D.1 Décompilation par ghidra

```c
undefined8 main(int iParm1,undefined8 *puParm2)

{
  int iVar1;
  longlong lVar2;
  undefined8 uVar3;
  undefined8 my_input;
  undefined8 uStack48;
  undefined8 local_28;
  undefined8 uStack32;
  undefined8 *local_18;
  undefined8 constante;
```

```
  uint local_8;
  uint sstic_fd;
  local_18 = (undefined8 *)0x0;
  constante = 0;
  my_input = 0;
  uStack48 = 0;
  local_28 = 0;
  uStack32 = 0;
  if ((iParm1 == 2) && (lVar2 = FUN_004153c0(puParm2[1]), lVar2 == 0x40)) {
    iVar1 = hextobytes(puParm2[1],&my_input,0x20);
    if (iVar1 == 0) {
      puts("can\'t decode hex");
      uVar3 = 1;
    }
    else {
      sstic_fd = openat("/dev/sstic",0);
      local_18 = (undefined8 *)&prog_enc;
      constante = 0x101010;
      ioctl((ulonglong)sstic_fd,0xc0105300,&local_18);
      local_18 = &my_input;
      constante = 0x20;
      ioctl((ulonglong)sstic_fd,0xc0105301,&local_18);
      do {
        ioctl((ulonglong)sstic_fd,0xc0105302,0);
        local_8 = ioctl((ulonglong)sstic_fd,0xc0105303,0);
        if ((local_8 & 0xffff) == 1) {
          if ((local_8 & 0xffff0000) == 0) {
            puts("Win");
          }
          else {
            puts("Loose");
          }
          goto LAB_00400910;
        }
      } while ((local_8 & 0xffff) != 0xffff);
      puts("Failure");
LAB_00400910:
      uVar3 = 0;
    }
  }
  else {
                    /* USAGE */
    printf(0x54ebe8,*puParm2);
    uVar3 = 1;
  }
```

```
  return uVar3;
}
```

D.2 Script de déchiffrement de la zone mémoire

```python
#!/usr/bin/python

# Some code is taken from https://github.com/windard/sm4/tree/master/Python

from binascii import hexlify, unhexlify


def pad(l):
    diff = 8 - len(l)
    return '0'*diff + l


def nrotl(num, bits, n):
    for i in range(n):
        num = rotl(num, bits)
    return num


def nrotr(num, bits, n):
    for i in range(n):
        num = rotr(num, bits)
    return num


def rotl(num, bits):
    bit = num & (1 << (bits-1))
    num <<= 1
    if(bit):
        num |= 1
    num &= (2**bits-1)

    return num


def rotr(num, bits):
    num &= (2**bits-1)
    bit = num & 1
    num >>= 1
    if(bit):
```

```
        num |= (1 << (bits-1))

    return num


SboxTable = [
        [0xd6,0x90,0xe9,0xfe,0xcc,0xe1,0x3d,0xb7,0x16,0xb6,0x14,0xc2,0x28,0xfb,0x2c,0x05],
        [0x2b,0x67,0x9a,0x76,0x2a,0xbe,0x04,0xc3,0xaa,0x44,0x13,0x26,0x49,0x86,0x06,0x99],
        [0x9c,0x42,0x50,0xf4,0x91,0xef,0x98,0x7a,0x33,0x54,0x0b,0x43,0xed,0xcf,0xac,0x62],
        [0xe4,0xb3,0x1c,0xa9,0xc9,0x08,0xe8,0x95,0x80,0xdf,0x94,0xfa,0x75,0x8f,0x3f,0xa6],
        [0x47,0x07,0xa7,0xfc,0xf3,0x73,0x17,0xba,0x83,0x59,0x3c,0x19,0xe6,0x85,0x4f,0xa8],
        [0x68,0x6b,0x81,0xb2,0x71,0x64,0xda,0x8b,0xf8,0xeb,0x0f,0x4b,0x70,0x56,0x9d,0x35],
        [0x1e,0x24,0x0e,0x5e,0x63,0x58,0xd1,0xa2,0x25,0x22,0x7c,0x3b,0x01,0x21,0x78,0x87],
        [0xd4,0x00,0x46,0x57,0x9f,0xd3,0x27,0x52,0x4c,0x36,0x02,0xe7,0xa0,0xc4,0xc8,0x9e],
        [0xea,0xbf,0x8a,0xd2,0x40,0xc7,0x38,0xb5,0xa3,0xf7,0xf2,0xce,0xf9,0x61,0x15,0xa1],
        [0xe0,0xae,0x5d,0xa4,0x9b,0x34,0x1a,0x55,0xad,0x93,0x32,0x30,0xf5,0x8c,0xb1,0xe3],
        [0x1d,0xf6,0xe2,0x2e,0x82,0x66,0xca,0x60,0xc0,0x29,0x23,0xab,0x0d,0x53,0x4e,0x6f],
        [0xd5,0xdb,0x37,0x45,0xde,0xfd,0x8e,0x2f,0x03,0xff,0x6a,0x72,0x6d,0x6c,0x5b,0x51],
        [0x8d,0x1b,0xaf,0x92,0xbb,0xdd,0xbc,0x7f,0x11,0xd9,0x5c,0x41,0x1f,0x10,0x5a,0xd8],
        [0x0a,0xc1,0x31,0x88,0xa5,0xcd,0x7b,0xbd,0x2d,0x74,0xd0,0x12,0xb8,0xe5,0xb4,0xb0],
        [0x89,0x69,0x97,0x4a,0x0c,0x96,0x77,0x7e,0x65,0xb9,0xf1,0x09,0xc5,0x6e,0xc6,0x84],
        [0x18,0xf0,0x7d,0xec,0x3a,0xdc,0x4d,0x20,0x79,0xee,0x5f,0x3e,0xd7,0xcb,0x39,0x48]
        ]


const = [0x70e15, 0x1c232a31, 0x383f464d, 0x545b6269]
keyin = [0x2078cc93, 0xb4c15038, 0xd5dc439c, 0x625f824]  # v0
cipher = [0x304b4e8e, 0x2d96464, 0xaf65c5dd, 0x476f8aaa]


def sm4ekey(v0, v1):
    roundresult = [i for i in v0]

    for i in range(4):
        c = v1[i]
        intval = roundresult[3] ^ roundresult[2] ^ roundresult[1] ^ c

        new_intval = 0
        for j in range(4):
            tmp = (intval >> j*8) & 0xff
            new_intval |= ((SboxTable[tmp >> 4][tmp & 0xf]) << j*8)

        new_intval ^= nrotl(new_intval, 32, 13) ^ nrotl(new_intval, 32, 23)
        new_intval ^= roundresult[0]
```

```
        roundresult[0] = roundresult[1]
        roundresult[1] = roundresult[2]
        roundresult[2] = roundresult[3]
        roundresult[3] = new_intval
    return roundresult

def sm4e(v0, v1):
    roundresult = [i for i in v0]

    for i in range(4):
        rk = v1[i]
        intval = roundresult[3] ^ roundresult[2] ^ roundresult[1] ^ rk

        new_intval = 0
        for j in range(4):
            tmp = (intval >> j*8) & 0xff
            new_intval |= ((SboxTable[tmp >> 4][tmp & 0xf]) << j*8)

        new_intval = new_intval ^ nrotl(new_intval, 32, 2) ^ nrotl(new_intval, 32, 10) ^ nrotl(n
        new_intval = new_intval ^ roundresult[0]

        roundresult[0] = roundresult[1]
        roundresult[1] = roundresult[2]
        roundresult[2] = roundresult[3]
        roundresult[3] = new_intval
    return roundresult


if __name__ == "__main__":
    with open("prog.bin.enc", "rb") as f:
        prog_enc = f.read()

    prog_plain = b''

    v2 = [0, 0, 0, 0]
    for i in range(len(prog_enc)//16):
        if not i % 1000:
            print(i)
        line = prog_enc[i*16:(i+1)*16]
        enc_line = [int.from_bytes(line[i*4:(i+1)*4], 'big') for i in range(4)]

        const_mod = [const[i] ^ v2[i] for i in range(len(const))]
        res = sm4ekey(keyin, const_mod)
        rks = res[::-1]
        res = sm4e(enc_line, rks)
```

```
        res = res[::-1]

        prog_line = ''.join([pad(hex(s)[2:]) for s in res])
        prog_plain += prog_line.encode()

        if (all([p == 0 for p in res])):
            print("Nothing here")

        v2 = [v + 16 for v in v2]

    with open("prog_plain.bin", "wb") as f:
        f.write(unhexlify(prog_plain))
```

D.3 Désassembleur pour la quatrième étape

```
#!/usr/bin/python


if __name__ == "__main__":
    with open("prog_plain.bin", "rb") as f:
        prog = f.read()

    with open("prog_disass", "wb") as f:
        for i in range(len(prog) // 3):
            instr = prog[3 * i:(i + 1) * 3]
            int_inst = int.from_bytes(instr, 'little')
            if int_inst == 0:
                break
            mod_op = int_inst >> 20
            op = (int_inst >> 18) & 0x3
            reg = (int_inst >> 14) & 0xf
            const = int_inst & 0x3fff

            bin_rep = bin(int_inst)[2:]
            bin_rep = '0' * (24 - len(bin_rep)) + bin_rep
            to_write_bin = bin_rep[0:4] + ' ' + bin_rep[4:6] + ' '
            to_write_bin += bin_rep[6:10] + ' ' + bin_rep[10:]

            hex_rep = hex(int_inst)[2:]
            line_nb = hex(i * 3)[2:]
            disass = '0' * (3 - len(line_nb)) + line_nb + ' : '
            disass += '0' * (6 - len(hex_rep)) + hex_rep + ' : '
            disass += to_write_bin + ' -> '
```

```
        if op == 3:
            no_reg = False
            no_const = False
            if mod_op == 0:
                disass += "mov "
            elif mod_op == 2:
                disass += "add "
            elif mod_op == 3:
                disass += "sub "
            elif mod_op == 4:
                disass += "lshift "
            elif mod_op == 5:
                disass += "rshift "
            elif mod_op == 7:
                disass += "and "
            elif mod_op == 8:
                disass += "jmp "
                no_reg = True
            elif mod_op == 9:
                disass += "jnz "
            elif mod_op == 0xe:
                disass += "too_slow_jmp "
                no_reg = True
            elif mod_op == 0xd:
                disass += "time_init "
            elif mod_op == 0xc:
                disass += "update_target "
                no_const = True
            else:
                disass += "??? "

            if not no_reg:
                disass += "$r{}, ".format(reg)
            if not no_const:
                disass += "#{}".format(hex(const))

        elif op == 0:
            op2_reg = False
            if mod_op == 0:
                disass += "mov "
                op2_reg = True
            elif mod_op == 1:
                disass += "dec "
            elif mod_op == 0xb:
                disass += "swap_bytes "
```

```
            elif mod_op == 2:
                disass += "or "
                op2_reg = True
            elif mod_op == 6:
                disass += "xor "
                op2_reg = True
            elif mod_op == 3:
                disass += "sub "
                op2_reg = True
            else:
                disass += "??? "

            disass += "$r{}".format(reg)
            if op2_reg:
                disass += ", $r{}".format(const >> 10)

        elif op == 2:
            if mod_op == 0:
                disass += "store "
            disass += "$r{}, ".format(const >> 10)
            disass += "[$r{}]".format(reg)

        elif op == 1:
            if mod_op == 0:
                disass += "load "
            else:
                disass += "??? "
            disass += "$r{}, ".format(reg)
            disass += "[$r{}]".format(const >> 10)
        else:
            disass += "???"

        f.write(disass.encode() + b'\n')
```

D.4 Code de la machine virtuelle désassemblé

```
000 : 0d0010 : 0000 11 0100 00000000010000 -> mov $r4, #0x10
003 : 4d0010 : 0100 11 0100 00000000010000 -> lshift $r4, #0x10
006 : 2d0020 : 0010 11 0100 00000000100000 -> add $r4, #0x20
009 : 0f4010 : 0000 11 1101 00000000010000 -> mov $r13, #0x10
00c : 4f4010 : 0100 11 1101 00000000010000 -> lshift $r13, #0x10
00f : 2f4020 : 0010 11 1101 00000000100000 -> add $r13, #0x20
012 : 0f0004 : 0000 11 1100 00000000000100 -> mov $r12, #0x4
```

```
015 : 041000 : 0000 01 0000 01000000000000 -> load $r0, [$r4]
018 : 4c0010 : 0100 11 0000 00000000010000 -> lshift $r0, #0x10
01b : 5c0010 : 0101 11 0000 00000000010000 -> rshift $r0, #0x10
01e : b00000 : 1011 00 0000 00000000000000 -> swap_bytes $r0
021 : 2d0002 : 0010 11 0100 00000000000010 -> add $r4, #0x2
024 : 045000 : 0000 01 0001 01000000000000 -> load $r1, [$r4]
027 : 4c4010 : 0100 11 0001 00000000010000 -> lshift $r1, #0x10
02a : 5c4010 : 0101 11 0001 00000000010000 -> rshift $r1, #0x10
02d : b04000 : 1011 00 0001 00000000000000 -> swap_bytes $r1
030 : 2d0002 : 0010 11 0100 00000000000010 -> add $r4, #0x2
033 : 049000 : 0000 01 0010 01000000000000 -> load $r2, [$r4]
036 : 4c8010 : 0100 11 0010 00000000010000 -> lshift $r2, #0x10
039 : 5c8010 : 0101 11 0010 00000000010000 -> rshift $r2, #0x10
03c : b08000 : 1011 00 0010 00000000000000 -> swap_bytes $r2
03f : 2d0002 : 0010 11 0100 00000000000010 -> add $r4, #0x2
042 : 04d000 : 0000 01 0011 01000000000000 -> load $r3, [$r4]
045 : 4cc010 : 0100 11 0011 00000000010000 -> lshift $r3, #0x10
048 : 5cc010 : 0101 11 0011 00000000010000 -> rshift $r3, #0x10
04b : b0c000 : 1011 00 0011 00000000000000 -> swap_bytes $r3
04e : 0f8020 : 0000 11 1110 00000000100000 -> mov $r14, #0x20
051 : 0dc007 : 0000 11 0111 00000000000111 -> mov $r7, #0x7
054 : dd0000 : 1101 11 0100 00000000000000 -> time_init $r4, #0x0
057 : 138000 : 0001 00 1110 00000000000000 -> dec $r14
05a : 010400 : 0000 00 0100 00010000000000 -> mov $r4, $r1
05d : 015000 : 0000 00 0101 01000000000000 -> mov $r5, $r4
060 : 5d0008 : 0101 11 0100 00000000001000 -> rshift $r4, #0x8
063 : 7d00ff : 0111 11 0100 00000011111111 -> and $r4, #0xff
066 : 7d40ff : 0111 11 0101 00000011111111 -> and $r5, #0xff
069 : 02d400 : 0000 00 1011 01010000000000 -> mov $r11, $r5
06c : 4ec008 : 0100 11 1011 00000000001000 -> lshift $r11, #0x8
06f : 029c00 : 0000 00 1010 01110000000000 -> mov $r10, $r7
072 : 4e8010 : 0100 11 1010 00000000010000 -> lshift $r10, #0x10
075 : 22ac00 : 0010 00 1010 10110000000000 -> or $r10, $r11
078 : 229000 : 0010 00 1010 01000000000000 -> or $r10, $r4
07b : 2e9000 : 0010 11 1010 01000000000000 -> add $r10, #0x1000
07e : 05a800 : 0000 01 0110 10100000000000 -> load $r6, [$r10]
081 : 7d80ff : 0111 11 0110 00000011111111 -> and $r6, #0xff
084 : 9dc08a : 1001 11 0111 00000010001010 -> jnz $r7, #0x8a
087 : 0dc00a : 0000 11 0111 00000000001010 -> mov $r7, #0xa
08a : 11c000 : 0001 00 0111 00000000000000 -> dec $r7
```

```
08d : 02d000 : 0000 00 1011 01000000000000 -> mov $r11, $r4
090 : 4ec008 : 0100 11 1011 00000000001000 -> lshift $r11, #0x8
093 : 029c00 : 0000 00 1010 01110000000000 -> mov $r10, $r7
096 : 4e8010 : 0100 11 1010 00000000010000 -> lshift $r10, #0x10
099 : 22ac00 : 0010 00 1010 10110000000000 -> or $r10, $r11
09c : 229800 : 0010 00 1010 01100000000000 -> or $r10, $r6
09f : 2e9000 : 0010 11 1010 01000000000000 -> add $r10, #0x1000
0a2 : 056800 : 0000 01 0101 10100000000000 -> load $r5, [$r10]
0a5 : 7d40ff : 0111 11 0101 00000011111111 -> and $r5, #0xff
0a8 : 9dc0ae : 1001 11 0111 00000010101110 -> jnz $r7, #0xae
0ab : 0dc00a : 0000 11 0111 00000000001010 -> mov $r7, #0xa
0ae : 11c000 : 0001 00 0111 00000000000000 -> dec $r7
0b1 : 02d800 : 0000 00 1011 01100000000000 -> mov $r11, $r6
0b4 : 4ec008 : 0100 11 1011 00000000001000 -> lshift $r11, #0x8
0b7 : 029c00 : 0000 00 1010 01110000000000 -> mov $r10, $r7
0ba : 4e8010 : 0100 11 1010 00000000010000 -> lshift $r10, #0x10
0bd : 22ac00 : 0010 00 1010 10110000000000 -> or $r10, $r11
0c0 : 229400 : 0010 00 1010 01010000000000 -> or $r10, $r5
0c3 : 2e9000 : 0010 11 1010 01000000000000 -> add $r10, #0x1000
0c6 : 052800 : 0000 01 0100 10100000000000 -> load $r4, [$r10]
0c9 : 7d00ff : 0111 11 0100 00000011111111 -> and $r4, #0xff
0cc : 9dc0d2 : 1001 11 0111 00000011010010 -> jnz $r7, #0xd2
0cf : 0dc00a : 0000 11 0111 00000000001010 -> mov $r7, #0xa
0d2 : 11c000 : 0001 00 0111 00000000000000 -> dec $r7
0d5 : 02d400 : 0000 00 1011 01010000000000 -> mov $r11, $r5
0d8 : 4ec008 : 0100 11 1011 00000000001000 -> lshift $r11, #0x8
0db : 029c00 : 0000 00 1010 01110000000000 -> mov $r10, $r7
0de : 4e8010 : 0100 11 1010 00000000010000 -> lshift $r10, #0x10
0e1 : 22ac00 : 0010 00 1010 10110000000000 -> or $r10, $r11
0e4 : 229000 : 0010 00 1010 01000000000000 -> or $r10, $r4
0e7 : 2e9000 : 0010 11 1010 01000000000000 -> add $r10, #0x1000
0ea : 05a800 : 0000 01 0110 10100000000000 -> load $r6, [$r10]
0ed : 7d80ff : 0111 11 0110 00000011111111 -> and $r6, #0xff
0f0 : 9dc0f6 : 1001 11 0111 00000011110110 -> jnz $r7, #0xf6
0f3 : 0dc00a : 0000 11 0111 00000000001010 -> mov $r7, #0xa
0f6 : 11c000 : 0001 00 0111 00000000000000 -> dec $r7
0f9 : 025800 : 0000 00 1001 01100000000000 -> mov $r9, $r6
0fc : 4e4008 : 0100 11 1001 00000000001000 -> lshift $r9, #0x8
0ff : 225000 : 0010 00 1001 01000000000000 -> or $r9, $r4
102 : 023800 : 0000 00 1000 11100000000000 -> mov $r8, $r14
```

```
105 : 5e0003 : 0101 11 1000 00000000000011 -> rshift $r8, #0x3
108 : 7e0001 : 0111 11 1000 00000000000001 -> and $r8, #0x1
10b : cf8000 : 1100 11 1110 00000000000000 -> update_target $r14,
10e : 9e012f : 1001 11 1000 00000100101111 -> jnz $r8, #0x12f
111 : 020c00 : 0000 00 1000 00110000000000 -> mov $r8, $r3
114 : 00f800 : 0000 00 0011 11100000000000 -> mov $r3, $r14
117 : 2cc001 : 0010 11 0011 00000000000001 -> add $r3, #0x1
11a : 60c000 : 0110 00 0011 00000000000000 -> xor $r3, $r0
11d : ecc14a : 1110 11 0011 00000101001010 -> too_slow_jmp #0x14a
120 : 60c400 : 0110 00 0011 00010000000000 -> xor $r3, $r1
123 : 002400 : 0000 00 0000 10010000000000 -> mov $r0, $r9
126 : 004800 : 0000 00 0001 00100000000000 -> mov $r1, $r2
129 : 00a000 : 0000 00 0010 10000000000000 -> mov $r2, $r8
12c : 8c014a : 1000 11 0000 00000101001010 -> jmp #0x14a
12f : 020000 : 0000 00 1000 00000000000000 -> mov $r8, $r0
132 : 002400 : 0000 00 0000 10010000000000 -> mov $r0, $r9
135 : 007800 : 0000 00 0001 11100000000000 -> mov $r1, $r14
138 : 2c4001 : 0010 11 0001 00000000000001 -> add $r1, #0x1
13b : 604000 : 0110 00 0001 00000000000000 -> xor $r1, $r0
13e : 604800 : 0110 00 0001 00100000000000 -> xor $r1, $r2
141 : ec414a : 1110 11 0001 00000101001010 -> too_slow_jmp #0x14a
144 : 008c00 : 0000 00 0010 00110000000000 -> mov $r2, $r3
147 : 00e000 : 0000 00 0011 10000000000000 -> mov $r3, $r8
14a : 9f8057 : 1001 11 1110 00000001010111 -> jnz $r14, #0x57
14d : b00000 : 1011 00 0000 00000000000000 -> swap_bytes $r0
150 : b04000 : 1011 00 0001 00000000000000 -> swap_bytes $r1
153 : 4c4010 : 0100 11 0001 00000000010000 -> lshift $r1, #0x10
156 : 200400 : 0010 00 0000 00010000000000 -> or $r0, $r1
159 : 0b4000 : 0000 10 1101 00000000000000 -> store $r0, [$r13]
15c : 2f4004 : 0010 11 1101 00000000000100 -> add $r13, #0x4
15f : b08000 : 1011 00 0010 00000000000000 -> swap_bytes $r2
162 : b0c000 : 1011 00 0011 00000000000000 -> swap_bytes $r3
165 : 4cc010 : 0100 11 0011 00000000010000 -> lshift $r3, #0x10
168 : 208c00 : 0010 00 0010 00110000000000 -> or $r2, $r3
16b : 0b4800 : 0000 10 1101 00100000000000 -> store $r2, [$r13]
16e : 2f4004 : 0010 11 1101 00000000000100 -> add $r13, #0x4
171 : 013400 : 0000 00 0100 11010000000000 -> mov $r4, $r13
174 : 130000 : 0001 00 1100 00000000000000 -> dec $r12
177 : 9f0015 : 1001 11 1100 00000000010101 -> jnz $r12, #0x15
17a : 0f0010 : 0000 11 1100 00000000010000 -> mov $r12, #0x10
```

```
17d : 4f0010 : 0100 11 1100 00000000010000 -> lshift $r12, #0x10
180 : 0ec020 : 0000 11 1011 00000000100000 -> mov $r11, #0x20
183 : 3f4020 : 0011 11 1101 00000000100000 -> sub $r13, #0x20
186 : 0d0000 : 0000 11 0100 00000000000000 -> mov $r4, #0x0
189 : 043400 : 0000 01 0000 11010000000000 -> load $r0, [$r13]
18c : 7c00ff : 0111 11 0000 00000011111111 -> and $r0, #0xff
18f : 047000 : 0000 01 0001 11000000000000 -> load $r1, [$r12]
192 : 7c40ff : 0111 11 0001 00000011111111 -> and $r1, #0xff
195 : 300400 : 0011 00 0000 00010000000000 -> sub $r0, $r1
198 : 9c01a1 : 1001 11 0000 00000110100001 -> jnz $r0, #0x1a1
19b : 011000 : 0000 00 0100 01000000000000 -> mov $r4, $r4
19e : 8c01a7 : 1000 11 0000 00000110100111 -> jmp #0x1a7
1a1 : 0d0001 : 0000 11 0100 00000000000001 -> mov $r4, #0x1
1a4 : 8c01a7 : 1000 11 0000 00000110100111 -> jmp #0x1a7
1a7 : 2f4001 : 0010 11 1101 00000000000001 -> add $r13, #0x1
1aa : 2f0001 : 0010 11 1100 00000000000001 -> add $r12, #0x1
1ad : 12c000 : 0001 00 1011 00000000000000 -> dec $r11
1b0 : 9ec189 : 1001 11 1011 00000110001001 -> jnz $r11, #0x189
1b3 : 001000 : 0000 00 0000 01000000000000 -> mov $r0, $r4
1b6 : a54000 : 1010 01 0101 00000000000000 -> return $r0
```

D.5 Code émulant la fonction modifiant l'"entrée" du programme

```
def f(inp, slow=False):
    new_inp = b''

    for i in range(len(inp)//8):
        partial_inp = inp[8*i:8*(i + 1)]
        r0 = partial_inp[0] << 8 | partial_inp[1]
        r1 = partial_inp[2] << 8 | partial_inp[3]
        r2 = partial_inp[4] << 8 | partial_inp[5]
        r3 = partial_inp[6] << 8 | partial_inp[7]

        r7 = 7
        for j in range(31, -1, -1):

            # DÉBUT CASCADE
            r4 = r1 >> 8
            r5 = r1 & 0xff
            r6 = tab413000[((r7 << 16) | (r5 << 8) | r4) + 0x1000] & 0xff
            r7 = (r7 - 1) % 10
```

```
            r5 = tab413000[((r7 << 16) | (r4 << 8) | r6) + 0x1000] & 0xff
            r7 = (r7 - 1) % 10
            r4 = tab413000[((r7 << 16) | (r6 << 8) | r5) + 0x1000] & 0xff
            r7 = (r7 - 1) % 10
            r6 = tab413000[((r7 << 16) | (r5 << 8) | r4) + 0x1000] & 0xff
            r7 = (r7 - 1) % 10
            # FIN CASCADE

            r9 = r6 << 8 | r4
            r8 = (j//8) % 2

            if r8 == 0:
                r8 = r3
                r3 = (j + 1) ^ r0
                if slow: # Too slow barrier here
                    continue
                r3 = r3 ^ r1
                r0 = r9
                r1 = r2
                r2 = r8
            else:
                r8 = r0
                r0 = r9
                r1 = (j + 1) ^ r0 ^ r2
                if slow: # Too slow barrier here
                    continue
                r2 = r3
                r3 = r8

        new_inp += ((r0 << 48) | (r1 << 32) | (r2 << 16) | r3).to_bytes(8,
                'big')

        #print(hexlify(new_inp))
        # /!\ Faire attention à la manière dont est encodé l'entier
        # J'ai fait un peu différement
    return new_inp
```

D.6 Code inversant la fonction f

```
# Inverse CASCADE by calling cascade3 with the right inputs (not really nice)

def cascade0(r6, r7, r4_l):
    for r4 in r4_l:
        r5_l = find_in_413000(r6, r7, r4)
```

```
        if r5_l != []:
            yield [r4, r5_l]

def cascade1(r5_fin, r7, r6_l):
    for r6 in r6_l:
        r4_l = find_in_413000(r5_fin, r7, r6)
        for r4, r5_l in cascade0(r6, (r7 + 1) % 10, r4_l):
            yield [r4, r5_l]

def cascade2(r4_fin, r7, r5_fin_l):
    for r5_fin in r5_fin_l:
        r6_l = find_in_413000(r4_fin, r7, r5_fin)
        for r4, r5_l in cascade1(r5_fin, (r7 + 1) % 10, r6_l):
            yield [r4, r5_l]

def cascade3(r6_fin, r7, r4_fin):
    r5_fin_l = find_in_413000(r6_fin, r7, r4_fin)
    for r4, r5_l in cascade2(r4_fin, (r7 + 1) % 10, r5_fin_l):
        yield [r4, r5_l]


def inv_inner(r_l, j):
    """
        Inverse the inner loop from iteration the j-th to the iteration 31-st
        using r_l as input.
    """
    r0 = r_l[0]
    r1 = r_l[1]
    r2 = r_l[2]
    r3 = r_l[3]
    r7 = r_l[4]
    r8 = (j//8) % 2

    if r8 == 0:
        r9 = r0
        r6_fin = r9 >> 8
        r4_fin = r9 & 0xff
        pre_r2 = r1
        pre_r3 = r2
    else:
        r9 = r0
        r6_fin = r9 >> 8
        r4_fin = r9 & 0xff
        pre_r3 = r2
        pre_r0 = r3
```

```
    pre_r1_l = []
    r7 = (r7 + 1) % 10
    # Inverse cascade
    for r4, r5_l in cascade3(r6_fin, r7, r4_fin):
        for r5 in r5_l:
            pre_r1_l.append(r4 << 8 | r5)
    r7 = (r7 + 3) % 10


    pre_r_l_l = []
    for pre_r1 in pre_r1_l:
        if r8 == 0:
            pre_r0 = r3 ^ (j + 1) ^ pre_r1
        else:
            pre_r2 = r1 ^ (j + 1) ^ r0
        pre_r_l_l.append([pre_r0, pre_r1, pre_r2, pre_r3, r7])

    for pre_r_l in pre_r_l_l:
        # Recursion through the 32 loops
        if j == 31:
            yield pre_r_l
        else:
            for pre_r_l in inv_inner(pre_r_l, j + 1):
                yield pre_r_l
    yield []

def inv_f(inp, slow=False):
    """
        Inverse the whole f function
    """
    pre_inp = b''
    for i in range(len(inp)//8):
        print("i = {}".format(i))
        partial_inp = inp[8*i:8*(i + 1)]
        print(hexlify(partial_inp))
        r0 = partial_inp[0] << 8 | partial_inp[1]
        r1 = partial_inp[2] << 8 | partial_inp[3]
        r2 = partial_inp[4] << 8 | partial_inp[5]
        r3 = partial_inp[6] << 8 | partial_inp[7]

        # given r0, r1, r2, r3
        r7 = 9
        r_l_it = inv_inner([r0, r1, r2, r3, r7], 0)

        found = False
```

```
        for r_l in r_l_it:
            if r_l != []:
                r0 = r_l[0]
                r1 = r_l[1]
                r2 = r_l[2]
                r3 = r_l[3]
                sol = ((r0 << 48) | (r1 << 32) | (r2 << 16) | r3).to_bytes(8, 'big')
                # Stop at first solution to get flag
                pre_inp += sol
                found = True
                break
        if not found:
            print("No solutions :(")
            return b''

    print(hexlify(pre_inp))
    return pre_inp
```