YE Luc - yeluc.88@gmail.com

CHALLENGE SSTIC 2019

Solution détaillée

Introduction

Ce challenge a été pour moi l'occasion d'apprendre énormément de choses, N'ayant eu qu'une petite journée pour rédiger ma solution, j'ai fait au mieux pour retracer toutes les étapes par lesquelles je suis passé, ainsi que les difficultés rencontrées, et d'extraire le code utile de toutes mes tentatives d'implémentations, mises en Annexes à titre informatif. Je m'excuse d'avance pour les possibles incohérences.

Merci à ma compagne Fanny, mes collègues et amis pour leur soutien indéfectible tout au long de l'épreuve qui, pour moi, a duré quasiment deux mois !

Étape 1

La présentation du challenge donne le texte suivant :

```
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 !
```

Visiblement, on a affaire à un téléphone avec des couches de chiffrement. Après avoir téléchargé l'archive **[1]** contenant le challenge, je me retrouve avec les fichiers suivants : `README`, `flash.bin`, `rom.bin`, `power_consumption.npz`. Le `README` donne une commande pour démarrer le téléphone virtualisé avec `QEMU`.

```
qemu-system-aarch64 -nographic -machine virt,secure=on -cpu max -smp 1 -m 1024 -bios rom.bin
-semihosting-config enable,target=native -device loader,file=./flash.bin,addr=0x04000000 -netdev
user,id=network0,hostfwd=tcp:127.0.0.1:5555-192.168.200.200:22 -net nic,netdev=network0
```

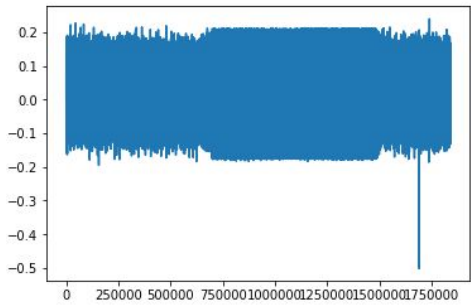
Après avoir téléchargé `QEMU v3.1.0` **[2]**, et non la toute dernière qui ne permet pas de démarrer le challenge (merci à David Bérard pour le support technique), je démarre ma machine virtuelle avec un Ubuntu 16, j'installe `QEMU`, et je peux démarrer le téléphone :

```
########################################
#      virtual environment detected    #
#           QEMU 3.1+ is needed        #
########################################
NOTICE:  Booting SSTIC ARM Trusted Firmware
KEYSTORE: keystore doesn't exist
ERROR:   KEYSTORE: Can't read keystore, reset keystore, try to boot again
```

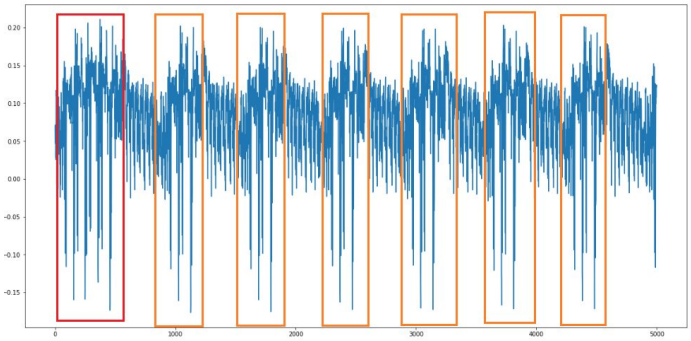
Un nouveau fichier `keystore` apparaît. Je relance la commande comme demandé, et on va un cran plus loin.

```
########################################
#      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 :
```

Je m'intéresse ensuite au fichier `.npz` contenant la trace de consommation de courant. Je démarre `jupyter notebook` pour utiliser `python`, je charge le fichier, je trouve une liste de données, que j'affiche :



En zoomant, dans la partie bien dense au milieu, on distingue deux types de motifs : le premier encadré en rouge, et le deuxième, encadré en orange. Ils se distinguent notamment par le nombre de pics descendants.



Le téléphone attend une clé RSA dont on ignore pour l'instant la taille. L'algorithme le plus connu pour implémenter l'exponentiation modulaire du RSA est le Square and Multiply. Pour chaque bit de la clé, une opération Square est toujours effectuée, et en fonction de sa valeur, une opération Multiply est effectuée en plus ou non. Tous les bits de la clé sont traités séquentiellement.

Visiblement, il y a de la fuite d'information par le courant, puisque l'on distingue deux types de motifs. Le calcul est donc vulnérable aux attaques Side-Channel. Étant donné que l'on est capable de distinguer la différence à l'oeil nu, il est possible de tenter une attaque SPA (Simple Power Analysis), où on considère que les deux motifs différents représentent un 0 ou un 1 pour chaque bit de la clé traité.

La subtilité de cette étape réside dans l'implémentation d'un algorithme pour détecter automatiquement ces motifs, car bien que le faire à la main soit tout à fait faisable, c'est chronophage et source d'erreurs.

J'implémente une détection de motifs à l'aide d'un seuil à paramétrer en me basant sur les pics descendants. Je considère qu'un motif commence lorsque je peux trouver un point qui dépasse mon seuil, par le bas : cela représente un pic descendant. Ce motif n'est pas terminé tant qu'il y a des points qui dépassent ce seuil dans un intervalle restreint de points (500 points), Puis, lorsque je trouve une zone sans pic descendant (500 points sans pic), alors le motif est terminé. Pour le détail du code, cf. Annexe A.

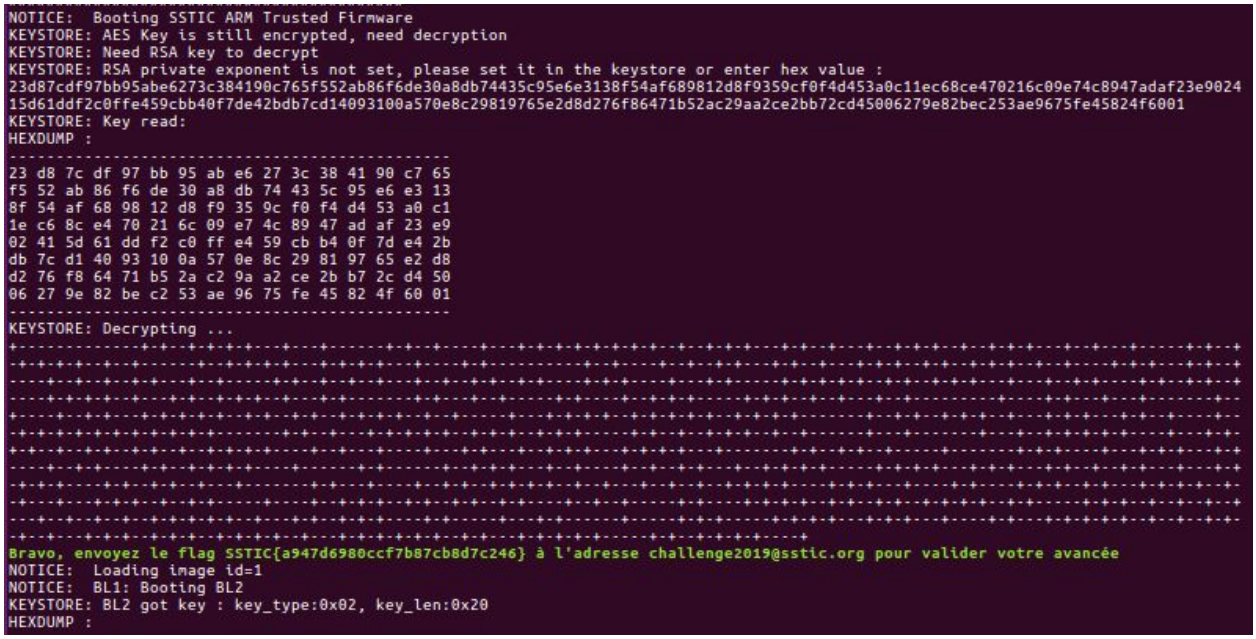
Une fois mon algorithme de détection implémenté et exécuté, j'obtiens 1024 motifs. C'est cohérent avec la taille d'une clé RSA 1024 bits. Par conséquent, j'associe chaque motif à un 0 ou un 1 et j'obtiens la clé suivante :

```
8006f241a27fae6975ca437d4179e4600a2b34edd47345594354ad8e261f6e4b1b47a6e981943170ea5008c9028b
3edbd427bef02dd39a27ff034fbb86ba824097c4f5b5e29132e79036840e273163788305ca2b2f0f39ac9f1b4819
16f52af1c8c767a93ac22edb150c7b6f61d54aafa6e309821c3ce467d5a9dde9fb3e1bc4
```

Je m'empresse de mettre cette clé dans l'émulateur, mais ça ne marche pas. J'inverse alors les 1 et les 0 dans mon algorithme, mais sans succès. On peut alors remarquer une série de + et de -, qui semblent correspondre aux bits de la clé, et semblent indiquer que l'ordre de la clé n'est pas celui qu'on pense : il faut inverser la valeur de la clé obtenue. J'inverse alors ma première clé :

```
23d87cdf97bb95abe6273c384190c765f552ab86f6de30a8db74435c95e6e3138f54af689812d8f9359cf0f4d453
a0c11ec68ce470216c09e74c8947adaf23e902415d61ddf2c0ffe459cbb40f7de42bdb7cd14093100a570e8c2981
9765e2d8d276f86471b52ac29aa2ce2bb72cd45006279e82bec253ae9675fe45824f6001
```

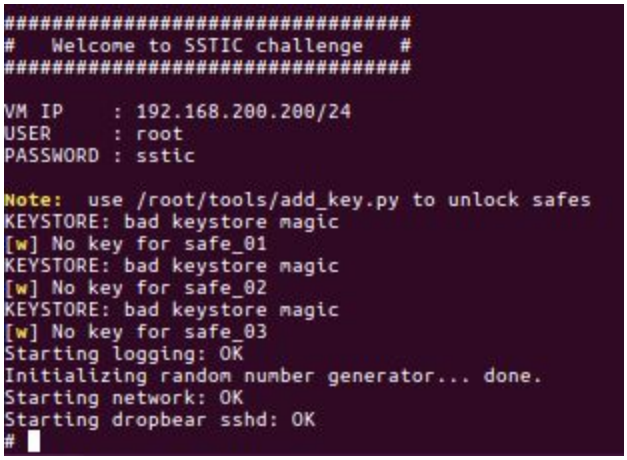
Et ça marche. Premier flag obtenu !



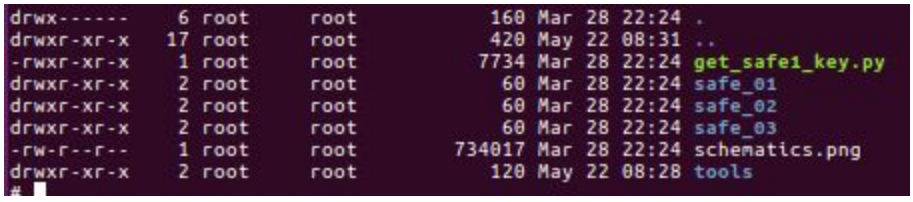
Flag 1 : SSTIC{a947d6980ccf7b87cb8d7c246}

Étape 2

Une fois le téléphone correctement démarré avec la bonne clé, on obtient ceci :



Je trouve un dossier correspondant à l'utilisateur `root` contenant ceci :

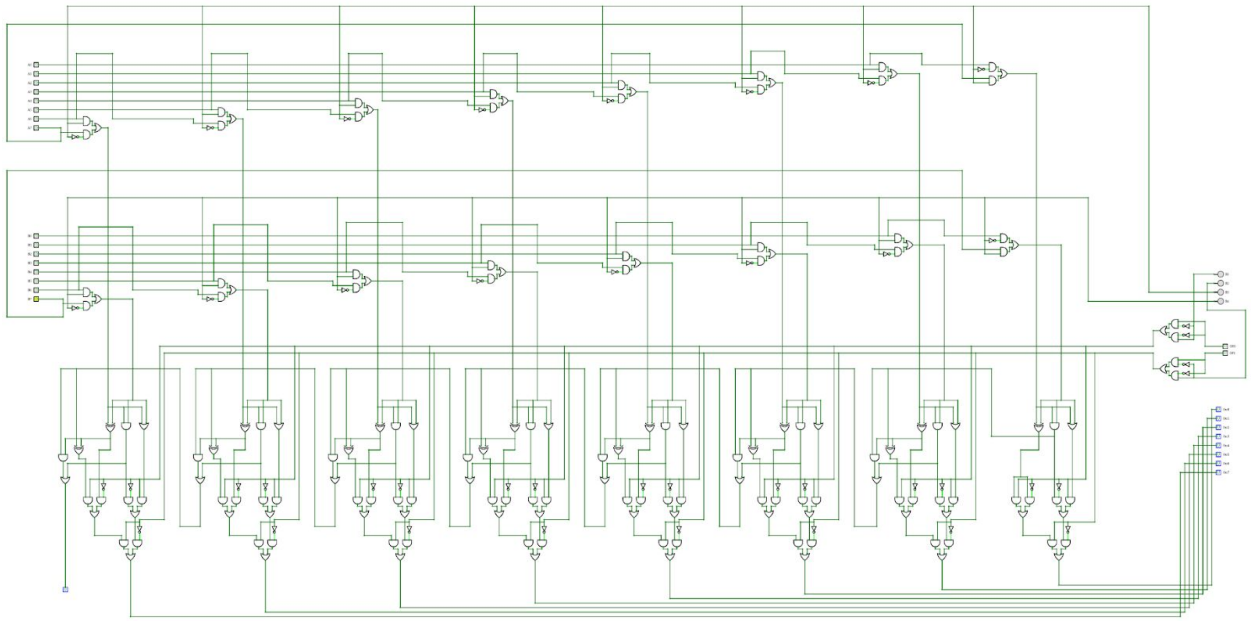


Les dossiers `safe_01`, `safe_02` et `safe_03` contiennent chacun un binaire chiffré, qu'il va falloir déchiffrer en spécifiant à chaque fois une clé trouvée. À chaque fois qu'une clé est trouvée, il y a un script `add_key.py` dans le dossier `/root/tools/` qui permet d'ajouter cette clé au keystore, et ce script se charge ensuite de déchiffrer le bon binaire, permettant d'accéder à l'épreuve suivante.

Je récupère le fichier schematics.png contenant l'épreuve suivante à l'aide de la commande `scp` :

```
scp -P 5555 root@localhost:/root/schematics.png .
```

Le schéma représente un circuit logique avec des entrées et des sorties :



N'ayant aucune idée de ce que cela représente, je me suis mis à recoder toutes les connexions. Une fois implémenté, n'étant pas plus avancé sur ce que cela représentait. Je me suis mis à tester quelques valeurs, et à me renseigner sur les portes logiques jusqu'à ce que je comprenne. L'entrée op (2 bits) et deux des boutons représentent ensemble une opération logique ou arithmétique (AND, OR, XOR ou ADD sur 8 bits). A et B sont deux entrées 8 bits, Les deux autres boutons indiquent si oui ou non A/B doivent subir une rotation sur 8 bits.

Plus simplement, le schéma se résume à ceci :

```
def secure_device(a,b,op):
    global BUTTONS
    buttons = int(BUTTONS)

    bop = (buttons & 0x3 ^ op)
    A = "%02x" % a
    B = "%02x" % b
    if (buttons & 0x4):
        a = ((a<<1) + (a>>7)) & 0xFF
    if (buttons & 0x8):
        b = ((b<<1) + (b>>7)) & 0xFF
    if (bop == 0):
        return a&b
    if (bop == 1):
        return a|b
    if (bop==2):
```

```
        return a^b
    if (bop==3):
        return (a+b)&0xFF
```

Enfin, tout à droite du schéma, il y a 4 entrées représentant des boutons physiques du téléphone. Le but est de retrouver la séquence des boutons appuyés.

Le fichier `get_safe1._key.py` est un script qui demande de reproduire la séquence d'appui de boutons de 8 étapes. à chaque étape de la séquence, l'appui des boutons forme une valeur de 4 bits, Ces 4 bits sont transformés à chaque étape en une valeur de 8 bits. À l'issue de la séquence, on se retrouve avec 8 octets qui sont hashés avec du SHA-256, puis comparés avec une valeur de référence.

En fin de compte, 8 valeurs de 4 bits, ça représente une valeur sur 32 bits, ce qui peut s'attaquer par une attaque en brute-force.

La subtilité du challenge consiste à optimiser dans une certaine mesure l'attaque par brute-force. En effet, si pour chaque valeur à tester, on repasse par l'implémentation logique du circuit, même si cette dernière est optimisée, cela rajoute beaucoup de calculs de manière conséquente, ce qui se ressent sur le temps de résolution de l'algorithme (une première estimation m'avait donné quelques jours). De ce fait, j'ai pré-calculé les sorties possibles de chaque étape de la séquence, et je les ai stockées dans 8 listes. Puis, 8 boucles imbriquées testent à chaque fois le hash de la valeur reconstituée à partir des 8 valeurs de 4 bits (1 valeur par liste), jusqu'à obtenir le hash de référence. En moins d'une demi-heure, la bonne valeur tombe. Les détails sont fournis dans l'Annexe B.

Une fois les bonnes 8 valeurs de 4 bits obtenues, on trouve la clé AES :

```
[+] Hash ok
[i] Dérivation de la clef AES safe_01
[i] aes key : 5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a
[i] Vous pouvez sauvegarder cette clef en utilisant /root/tools/add_key.py key
(base) luc@rootfs$
```

Deuxième clé obtenue :

```
5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a
```

```
# ./add_key.py 5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a
[+] Key with key_id 00000002 ok
[+] Key added into keystore
[+] Envoyez le flag SSTIC{5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a} à l'adresse challenge2019@sstic.org pou
r valider votre avancée
[+] Container /root/safe_01/.encrypted decrypted to /root/safe_01/decrypted_file
#
```

Flag 2 : SSTIC{5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a}

Étape 3

Exploration du binaire

J'exécute le nouveau binaire du dossier `/root/safe_01/` et je vois qu'il attend une chaîne en entrée en guise de flag. Je récupère le binaire avec `scp`, et je l'ouvre avec un outil d'analyse statique, `Ghidra`. Je suis le code depuis le point d'entrée.

```
2  void entry(undefined8 param_1)
3
4  {
5    undefined8 param_9;
6
7    __libc_start_main(main_FUN_00402e68,param_9,&stack0x00000008,FUN_00402fe0,FUN_00403060,param_1);
8                      /* WARNING: Subroutine does not return */
9    abort();
```

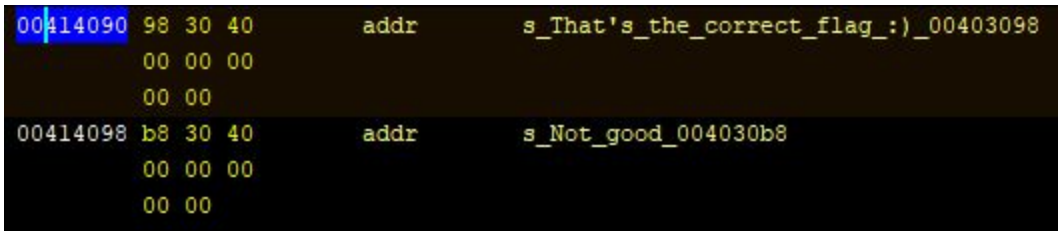
L'exécution de code se poursuit à l'adresse du premier argument de la fonction `__libc_start_main`.

```
2  undefined8 main_FUN_00402e68(int iParm1,undefined8 *puParm2)
3
4  {
5    if (iParm1 != 2) {
6      printf("Usage : %s <flag>\n",*puParm2);
7                      /* WARNING: Subroutine does not return */
8      exit(1);
9    }
0                      /* try { // try from 00402eb4 to 00402eb7 has its CatchHandler @ 00402f20 */
1    FUN_00402e34(puParm2[1]);
2    return 0;
3  }
```

La fonction `main` appelle ensuite une fonction à l'adresse `0x402e34`, qui appelle ensuite la fonction `__cxa_throw`.

```
2  void FUN_00402e34(undefined8 uParm1)
3
4  {
5    undefined8 *puVar1;
6
7    puVar1 = (undefined8 *)__cxa_allocate_exception(8);
8    *puVar1 = uParm1;
9                      /* WARNING: Subroutine does not return */
0    __cxa_throw(puVar1,typeinfo,0);
1  }
```

On trouve dans la liste des chaînes de caractères celles qui affichent “That’s the correct flag :)” ou “Not good”, mais rien qui n’y fait référence.



Pendant la semaine qui a suivi, pas moyen de comprendre ce que fait le code. J’ai tenté de tracer l’exécution de code avec `gdb`, mais impossible de comprendre ce qu’il faisait. Une section du fichier avait tout de même attiré l’attention, la section `.gnu.hash` contenant beaucoup d’octets ne remplissant pas leur réelle fonction de simples valeurs de hash. J’ai tenté de désassembler ce code dans tous les langages assembleurs possibles, À chaque langage testé, j’avais un espoir car le code désassemblé avait du sens, jusqu’à ce que je me rende compte que ça n’allait pas m’aider. En effet, plus un langage possède une liste d’opcode fournie, plus il avait de chance de donner du sens aux octets de la section `.gnu.hash`. De plus je n’avais aucun moyen de déterminer quel était le bon langage qu’il fallait choisir. Le binaire étant du AARCH64, je me suis documenté sur la gestion des exceptions avec de la documentation ARM, sans pouvoir aller plus loin.

Découverte du DWARF

Après avoir cherché pendant un moment, je tombe sur une vidéo sur Youtube de la CppCon 2017. Dave Watson, un ingénieur chez Facebook, y fait une présentation intitulée “*C++ Exceptions and Stack Unwinding*” **[3]**. J’y apprends alors qu’avec les fichiers au format ELF, il y a le format DWARF qui fournit des informations de débogage du code source. Lorsqu’une exception survient, il y a du code qui permet de nettoyer la pile d’exécution et de remettre le contexte des registres dans un état cohérent. Ce code est implémenté en instructions DWARF.

J’ai trouvé deux commandes qui permettent d’obtenir ces instructions DWARF à partir du binaire :

```
(Commande 1) readelf --debug-dump=frames decrypted_file
(Commande 2) objdump --dwarf decrypted_file
```

La spécification DWARF **[4]** décrit les instructions DWARF. J’ai commencé par utiliser la version 3, avant de trouver la 4 et la 5, ces dernières possédant des explications beaucoup plus claires[1].

J’ai alors compris qu’il y avait probablement du code caché en DWARF en lisant le papier de Oakley et Bratus, intitulé “*Exploiting the hard-working DWARF: Trojan and Exploit Techniques With No Native Executable Code*” **[5]**.

[1] Le page rank de Google m’a joué des tours …

DWARF en bref, c'est un langage qui décrit une gigantesque table, dont chaque ligne représente une adresse du Program Counter, et chaque colonne représente un registre, Chaque ligne représente un ensemble de règles pour modifier la valeur des registres à mettre en cas de problème. Voici une représentation de cette table extraite de **[5]**:

PC (eip)	CFA	ebp	ebx	eax	return addr.
0xf000f000	rsp+16	*(cfa-16)			*(cfa-8)
0xf000f001	rsp+16	*(cfa-16)			*(cfa-8)
0xf000f002	rbp+16	*(cfa-16)		eax=edi	*(cfa-8)
⋮	⋮	⋮	⋮	⋮	⋮
0xf000f00a	rbp+16	*(cfa-16)	*(cfa-24)	eax=edi	*(cfa-8)

Cette table est gigantesque, mais c'est avant tout une manière de voir et interpréter les instructions DWARF. De nombreuses cases de cette table sont identiques. C'est en fait une technique de compression pour décrire cette table. Les deux commandes précédentes permettent d'interpréter les Frame Description Entities (FDE) et les Common Information Entities (CIE), situées dans la section `.eh_frame`. Les détails sont dans la spécification **[4]**.

Voici un extrait de ces instructions produites à partir du binaire::

```
[...]
00000090 000000000000001c 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

000000b0 0000000000000024 000000a0 FDE cie=00000014 pc=0000000000402e68..0000000000402f64
  Augmentation data:     a3 00 00 00
  DW_CFA_advance_loc: 1 to 0000000000402e69
  DW_CFA_def_cfa_offset: 64
  DW_CFA_offset: r29 (x29) at cfa-64
  DW_CFA_offset: r30 (x30) at cfa-56
  DW_CFA_advance_loc: 2 to 0000000000402e6b
  DW_CFA_offset: r19 (x19) at cfa-48
  DW_CFA_advance_loc: 59 to 0000000000402ea6
  DW_CFA_restore: r30 (x30)
  DW_CFA_restore: r29 (x29)
  DW_CFA_restore: r19 (x19)
  DW_CFA_def_cfa_offset: 0
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
[...]
```

La spécificationaide à comprendre par exemple que l'instruction `DW_CFA_advance_loc` permet d'avancer de ligne en ligne dans la description de la table DWARF.

Dans le binaire, une instruction ayant attiré mon attention est `DW_CFA_val_expression` (opcode `0x16`) : cette instruction prend un registre et une expression DWARF. L'expression DWARF change la règle pour le registre indiqué, ici `x28`.

L'expression DWARF est décodée par `objdump/readelf`: c'est une instruction `DW_OP_skip -12222`. D'après la spécification, à partir de la valeur de l'adresse de l'instruction qui suit la constante -12222, il faut y ajouter la valeur de cette constante pour trouver une autre adresse où l'exécution des instructions DWARF se poursuit. De ce fait, l'adresse de l'instruction DWARF qui suit la constante -12222 est à `0x403216`. En appliquant l'offset -12222, on obtient l'adresse `0x400258`, adresse qui pointe dans la section `.gnu.hash`, un peu après le début de la section. Des instructions DWARF étaient cachées dans la section `.gnu.hash` !

Désassemblage des instructions DWARF

Afin de désassembler ce bytecode, j'ai tenté d'utiliser `Lief` **[6]** afin de coller toute la section `.gnu.hash` à partir de l'adresse `0x400258`, et de la mettre à la suite de l'instruction `DW_CFA_val_expression`, `Lief` est un outil permettant de parser et de modifier des fichiers exécutables de manière aisée. Une fois le binaire modifié, j'espérais que les outils `objdump/readelf` puissent décoder toutes ces instructions DWARF, sans succès.

Ne trouvant pas ce que je cherchais ou pratique d'utilisation pour désassembler le DWARF, j'ai codé mon propre désassembleur (cf. Annexes, lien Github). Voici un extrait du code désassemblé :

```
|gnu.hash_@| offset | [opcode] description
|0x400258| 0x0000 | [ 0x6f ] DW_OP_reg31
|0x400259| 0x0001 | [ 0x08 ] DW_OP_const1u  0xa8
|0x40025b| 0x0003 | [ 0x22 ] DW_OP_plus
|0x40025c| 0x0004 | [ 0x06 ] DW_OP_deref
|0x40025d| 0x0005 | [ 0x08 ] DW_OP_const1u  0x08
|0x40025f| 0x0007 | [ 0x22 ] DW_OP_plus
|0x400260| 0x0008 | [ 0x06 ] DW_OP_deref
|0x400261| 0x0009 | [ 0x12 ] DW_OP_dup
|0x400262| 0x000a | [ 0x06 ] DW_OP_deref
|0x400263| 0x000b | [ 0x16 ] DW_OP_swap
|0x400264| 0x000c | [ 0x08 ] DW_OP_const1u  0x08
|0x400266| 0x000e | [ 0x22 ] DW_OP_plus
|0x400267| 0x000f | [ 0x12 ] DW_OP_dup
[...]
|0x401977| 0x171f | [ 0x24 ] DW_OP_shl
|0x401978| 0x1720 | [ 0x21 ] DW_OP_or
|0x401979| 0x1721 | [ 0x0c ] DW_OP_const4u  0xffffffff
|0x40197e| 0x1726 | [ 0x1a ] DW_OP_and
|0x40197f| 0x1727 | [ 0x27 ] DW_OP_xor
|0x401980| 0x1728 | [ 0x2f ] DW_OP_skip  0xec6d
```

Le code est scindé en deux parties, avec des données au milieu. La première partie se situe de l'adresse `0x400285` à l'adresse `0x40063f` incluse. Puis, des données sont situées entre entre `0x400640` et `0x400ab4`. La deuxième partie de code se situe entre `0x400ab5` et `0x401980`.

Il s'agit d'une machine virtuelle à pile. On y trouve des opérations logiques (`and`, `or`, `xor`, etc.), des opérations arithmétiques, des opérations de manipulation de pile (`rot` qui décale le sommet de pile deux positions plus loin, `swap`, etc.), etc. On trouve surtout les opérations `bra` et `skip`, qui sont respectivement des branchements conditionnels et inconditionnels. Ce sont elles qui permettent de délimiter les blocs basiques de code. Les noms sont abrégés, mais commencent tous par `DW_OP_`.

Émulation des instructions DWARF

Pour effectuer le reverse du code DWARF, j'ai commencé à essayer d'écrire à la main l'évolution de la pile DWARF. Cela m'a permis au bout d'un certain nombre d'instructions de me familiariser avec les instructions DWARF, Au vu du nombre d'instructions (quelques milliers), ma patience a eu ses limites, et il a fallu que je me munisse de mon propre émulateur afin de pouvoir suivre l'évolution de la pile DWARF.

Avant de me lancer dans l'implémentation de l'émulateur, j'ai tout d'abord investi du temps pour pouvoir déboguer facilement mon émulateur. Pour ce faire, j'ai utilisé `gdb-multiarch` avec l'extension `gef`.

En regardant les bibliothèques logicielles chargées par le binaire, je finis par trouver la lib `libgcc_s.so.1` qui implémente les fonctions de gestion d'exception. Pour l'analyser, je me suis aidé d'une implémentation proche de cette bibliothèque, le projet Github `gcc` de `gcc-mirror` **[7]**. La fonction `execute_stack_op` du fichier `unwind-dw2.c` traite l'adresse pointant vers l'instruction DWARF à traiter. Dans le fichier `libgcc_s.so.1`, cette fonction est mappée à l'adresse `0x10b8b4` (en considérant que `Ghidra` mappe le début du fichier à l'adresse `0x100000`).

J'ai modifié la commande de `QEMU` pour ajouter un port 12345, afin de pouvoir y attacher un client `gdb`.

```
Pour le terminal lançant QEMU, j'ai ajouté ceci à la commande initiale :

(QEMU_host)# [...],hostfwd=tcp:127.0.0.1:12345-192.168.200.200:12345
```

Puis, j'ai exécuté le binaire `decrypted_file` avec `gdbserver`, afin de permettre le débogage à distance (<args> représentant la chaîne d'entrée).

```
(QEMU_emulated_phone)# gdbserver host:12345 ./decrypted_file <args>
```

Côté client, après avoir lancé `gdb-multiarch` avec `gef`, la commande suivante permet de s'attacher au `gdbserver` précédemment exécuté :

```
(VM_side)# gef-remote :12345
```

En tâtonnant, je finis par arriver à l'endroit où la fonction précédemment identifiée `execute_stack_op` est exécutée. Un point d'arrêt `gdb` est mis en place à cet endroit. J'ai configuré `gef` afin d'afficher un maximum d'adresses de la pile d'exécution, le tout empaqueté dans un script `gdb` pour faciliter l'automatisation. Lors du débogage avec `gdb`, la fonction `execute_stack_op` n'est pas mappée à une adresse constante, il m'a fallu automatiser l'atteinte de cette fonction pas à pas (cf. Annexe C).

L'image suivante montre une capture d'écran du contexte affiché par `gef` à chaque fois que l'exécution atteint le point d'arrêt spécifié, La chaîne passée en entrée est `AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH` pour la capture d'écran.

La capture d'écran est prise juste avant l'exécution du premier branchement `bra` (adresse `0x40027b`). J'ai pu ainsi suivre l'évolution de la pile, et comprendre que toutes les instructions précédentes permettaient de charger la chaîne passée en entrée, dans la pile. Le sommet de la pile contient les valeurs ASCII de `GGGGHHHH`, et la pile grandit vers les adresses hautes lorsque les éléments sont poussés sur la pile.

De cette manière j'ai pu vérifier l'implémentation de mon émulateur, ainsi que suivre l'évolution de la pile.

Après avoir codé mon émulateur (cf. Annexes, lien Github), j'ai pu simuler l'exécution du code. J'ai dû ruser pour certaines instructions DWARF, notamment à l'instruction `deref` qui va récupérer une valeur à l'adresse spécifiée. J'ai contourné ça en démarrant l'émulation après la dernière instruction `deref` (adresse relative `0x23` ou absolue `0x40027b`: `bra 0x0044`), et en définissant un état initial de la pile juste avant l'exécution des instructions suivantes.


registers
$x0 : 0x000000000040027b → 0x2f03150315004428 ("(D"?)
$x1 : → ldrb w6, [x6]
$x2 : 0x0000ffff8f21bd6c → add x6, x6, x1
$x3 : 0x0000ffff8f21bb00 → cmp w7, #0x90
$x4 : 0x3
$x5 : 0x3
$x6 : 0x0
$x7 : 0x28
$x8 : 0x0
$x9 : 0x0
$x10 : 0x0
$x11 : 0x0
$x12 : 0x0
$x13 : 0x0
$x14 : 0x0
$x15 : 0x0000ffff8f210620 → 0x0000000000000000
$x16 : 0x0000ffff8f230000 → 0x0000ffff8f13c190 → prfm pldl1keep, [x1]
$x17 : 0x0000ffff8f13c190 → prfm pldl1keep, [x1]
$x18 : 0x35
$x19 : 0x6
$x20 : 0x0000fffffe5fb2d0 → 0x0000fffffe5fbd80 → 0x0000000016f54ed0 → 0x474e5543432b2b00
$x21 : 0x0000ffff8f21ead4 → 0x009400920090008d
$x22 : 0x0000fffffe5fb058 → 0x0000000000000000
$x23 : 0x0000fffffe5fb050 → 0x0000000000000001
$x24 : 0x0000000000403216 → 0x00a0000000240000
$x25 : 0x000000000040027b → 0x2f03150315004428 ("(D"?)
$x26 : 0x0000fffffe5fb070 → 0x0000fffffe5fcc80 → 0x0000fffffe5fccc0 → 0x0000000000000000
$x27 : 0x0
$x28 : 0x0
$x29 : 0x0000fffffe5fb000 → 0x0000fffffe5fb270 → 0x0000fffffe5fb690 → 0x0000fffffe5fb6b0 → 0x0000fffffe5fbd70 → 0x0000fffffe5fcc30 → 0x0000fffffe5fcc60 → 0x0000fffffe5fcc80
$x30 : 0x0000ffff8f21bc1c → mov x6, x0
$sp : 0x0000fffffe5fb000 → 0x0000fffffe5fb270 → 0x0000fffffe5fb690 → 0x0000fffffe5fb6b0 → 0x0000fffffe5fbd70 → 0x0000fffffe5fcc30 → 0x0000fffffe5fcc60 → 0x0000fffffe5fcc80
$pc : 0x0000ffff8f21b92c → add x25, x0, #0x1
$cpsr: [fast interrupt overflow carry zero NEGATIVE]
$fpsr: 0x0
$fpcr: 0x0
stack
0x0000fffffe5fb000|+0x0000: 0x0000fffffe5fb270 → 0x0000fffffe5fb690 → 0x0000fffffe5fb6b0 → 0x0000fffffe5fbd70 → 0x0000fffffe5fcc30 → 0x0000fffffe5fcc60 → 0x0000fffffe5fcc80 ←$x29, $sp
0x0000fffffe5fb008|+0x0008: 0x0000ffff8f21c004 → b 0xffff8f21bf8c
0x0000fffffe5fb010|+0x0010: 0x0000fffffe5fc1f0 → 0x0000fffffe5fbd80 → 0x0000000016f54ed0 → 0x474e5543432b2b00
0x0000fffffe5fb018|+0x0018: 0x000000000000001c
0x0000fffffe5fb020|+0x0020: 0x0000fffffe5fb6f0 → 0x0000000000000000
0x0000fffffe5fb028|+0x0028: 0x0000fffffe5fb2d0 → 0x0000fffffe5fbd80 → 0x0000000016f54ed0 → 0x474e5543432b2b00
0x0000fffffe5fb030|+0x0030: 0x0000fffffe5fcc80 → 0x0000fffffe5fccc0 → 0x0000000000000000
0x0000fffffe5fb038|+0x0038: 0x0000ffff8f2301c0 → 0x0808080808080808
0x0000fffffe5fb040|+0x0040: 0x0000ffff8f21eb10 → 0x000000423f36260a ("&6?B"?)
0x0000fffffe5fb048|+0x0048: 0x0000fffffe5fb2c8 → 0x0000000000000003
0x0000fffffe5fb050|+0x0050: 0x0000000000000001 ←$x23
0x0000fffffe5fb058|+0x0058: 0x0000000000000000 ←$x22
0x0000fffffe5fb060|+0x0060: 0x0000000000000001
0x0000fffffe5fb068|+0x0068: 0x0000ffff8f4629c0 → 0x0000ffff8f210000 → 0x00010102464c457f
0x0000fffffe5fb070|+0x0070: 0x0000fffffe5fcc80 → 0x0000fffffe5fccc0 → 0x0000000000000000 ←$x26
0x0000fffffe5fb078|+0x0078: "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
0x0000fffffe5fb080|+0x0080: "CCCCDDDDEEEEFFFFGGGGHHHH"
0x0000fffffe5fb088|+0x0088: "EEEEFFFFGGGGHHHH"
0x0000fffffe5fb090|+0x0090: "GGGGHHHH"
0x0000fffffe5fb098|+0x0098: 0x0000000000000000
0x0000fffffe5fb0a0|+0x00a0: 0x0000000000000008
0x0000fffffe5fb0a8|+0x00a8: 0x0000000000000000
0x0000fffffe5fb0b0|+0x00b0: 0x0000000000000000
0x0000fffffe5fb0b8|+0x00b8: 0x0000000000000000
0x0000fffffe5fb0c0|+0x00c0: 0x0000000000000000
0x0000fffffe5fb0c8|+0x00c8: 0x0000000000000000
0x0000fffffe5fb0d0|+0x00d0: 0x0000000000000000
0x0000fffffe5fb0d8|+0x00d8: 0x0000000000000000
0x0000fffffe5fb0e0|+0x00e0: 0x0000000000000000
0x0000fffffe5fb0e8|+0x00e8: 0x0000000000000000
0x0000fffffe5fb0f0|+0x00f0: 0x0000fffffe5fb150 → 0x0000fffffe5fb240 → 0x0000000000000001
0x0000fffffe5fb0f8|+0x00f8: 0x0000000000000000
0x0000fffffe5fb100|+0x0100: 0x0000ffff8f462d20 → 0x0000ffff8f468430 → 0x0000ffff8f463478 → 0x0000ffff8f468170 → 0x0000000000000000
0x0000fffffe5fb108|+0x0108: 0x000000018f0c94e0
0x0000fffffe5fb110|+0x0110: 0x0000000000000001
0x0000fffffe5fb118|+0x0118: 0x0000000073c3a79
0x0000fffffe5fb120|+0x0120: 0x0000ffff8f462ea0 → 0x0000ffff8f0b8000 → 0x03010102464c457f
0x0000fffffe5fb128|+0x0128: 0x0000ffff8f0c2a78 → 0x000b001a00001d77 ("w"?)
0x0000fffffe5fb130|+0x0130: 0x0000ffff8f462ea0 → 0x0000ffff8f0b8000 → 0x03010102464c457f
0x0000fffffe5fb138|+0x0138: 0x0000fffffe5fb218 → "CCCCDDDD"
0x0000fffffe5fb140|+0x0140: 0x0000fffffe5fb214 → "BBBBCCCCDDDD"
0x0000fffffe5fb148|+0x0148: 0x0000000000000000
code:arm64:ARM

Reverse de l'algorithme implémenté en DWARF

En m'aidant de l'émulateur, je m'attelle au reverse de l'algorithme DWARF, afin de le simplifier avec du code Python. Parmi les éléments notables rencontrés durant le reverse, il y a une zone contenant quasiment 2000 instructions sans branchement intermédiaire, entre les adresses absolues `0x400ab5` et `0x401985`. À l'aide de l'émulateur, j'ai pu en déduire que cela se résumait à une simple substitution de valeurs : en fonction de la valeur au sommet de la pile, cette zone d'instructions transformait uniquement le sommet de la pile. Comme il ne pouvait y avoir que deux valeurs en entrée de cette zone, la sortie ne donnait que deux possibles valeurs.

Après de nombreuses heures à naviguer à travers toutes ces instructions, et à essayer de reproduire l'algorithme, j'arrive enfin à une première implémentation en Python de l'algorithme : une grosse fonction contenant plusieurs boucles imbriquées. À la fin de l'algorithme, les 4 valeurs du sommet de pile sont comparées avec 4 autres valeurs de référence par un `xor`. Les 4 résultats de `xor` sont additionnés, pour former une valeur finale au sommet de pile, et selon le résultat, l'adresse de la chaîne "That's the correct flag :)" ou de la chaîne "Not good" est placée sur le sommet de pile avant de terminer les instructions DWARF.

Voyant des constantes dans la partie intermédiaire des instructions DWARF, je me suis dit que c'était peut-être un algorithme cryptographique plus ou moins connu avec un nombre atypique de rondes. Après quelques heures de recherche infructueuse sur l'Internet, je consacre encore quelques heures à la factorisation du code, et j'arrive à une version beaucoup plus simplifiée, en comparaison avec l'équivalent en DWARF. L'implémentation complète est disponible en Annexes, lien Github. Voici le début de l'algorithme.

```
def func_b(B0, B1):
    b0, b1 = B0, B1
    for b in range(4):
        b0, b1 = F(b0, b1)
    return b0, b1

def func_a(v0, v1, v2, v3):
    bf0, bf1 = func_b(v2, v3)
    vf0, vf1 = func_h(bf0, bf1, v0, v1)
    bf2, bf3 = func_b(vf0, vf1)
    vf2, vf3 = func_h(bf2, bf3, v2, v3)
    return vf0, vf1, vf2, vf3

def check_sol(v0, v1, v2, v3):
    for a in range(4):
        v0, v1, v2, v3 = func_a(v0, v1, v2, v3)
    if v0 == s0 and v1 == s1 and v2 == s2 and v3 == s3:
        print("FLAG !")
    else:
        print("Lose")
```

```
v0, v1, v2, v3 = 0x4242424241414141, 0x4444444443434343, 0x4646464645454545,
0x4848484847474747
#program starts here
check_sol(v0, v1, v2, v3)
```

Lors de l'étape de reverse, j'ai constaté que le contenu de la pile grandissait. Après chaque passage dans la fonction `func_h`, la pile avait grandi de 2 éléments. Donc après chaque passage dans la fonction `func_a`, il y avait 4 éléments de plus dans la pile.

En effet, en partant de 4 paramètres initiaux, un appel de la fonction `func_a` produit 4 nouvelles valeurs. Ces dernières servent ensuite de paramètres pour l'appel suivant de la fonction `func_a`. Au bout des 4 appels de la fonction `func_a`, la pile a grandi de 16 éléments. 4 valeurs finales étaient produites sur le sommet de pile, `vf0`, `vf1`, `vf2` et `vf3` de la dernière exécution de la fonction `func_a`. Je numérote les "équations" de la fonction `func_a` comme suit :

```
Equations de func_a:
(input: v0, v1, v2, v3)
(output: vf0, vf1, vf2, vf3)

(1)   bf0, bf1 = func_b(v2, v3)
(2)   vf0, vf1 = func_h(bf0, bf1, v0, v1)
(3)   bf2, bf3 = func_b(vf0, vf1)
(4)   vf2, vf3 = func_h(bf2, bf3, v2, v3)
```

La fonction `func_b` n'est pas inversible, mais supposons que `func_h` l'est. Par conséquent, de l'équation (3), on peut calculer `bf2`, `bf3` à partir de `vf0`, `vf1`. Dans ce cas, si `func_h` est inversible (notons-la `reverse_func_h`), selon l'équation (4), on peut espérer trouver `v2`, `v3` à partir de `vf2`, `vf3`, `bf2`, `bf3`. Du coup, ça se débloque pour les deux premières équations. De l'équation (1), on peut obtient `bf0`, `bf1` à partir de `v2`, `v3`. Finalement, avec l'équation (2), on peut trouver `v0`, `v1` à partir de `vf0`, `vf1`, `bf0`, `bf1`.

Par conséquent, il faut inverser la fonction `func_h`. Dans mon implémentation, `func_h` comporte 9 opérations, que je numérote comme suit :

```
Equations de func_h:
(input: B0, B1, C0, C1)
(output: nouvelles valeurs C0, C1)
h est le numéro de l'itération (de 0 à 14 inclus)

(1) V1, V2 = func_m(B0, B1, h)
(2) Y0, Y1 = func_j(LSB4(C1), MSB4(C1))
(3) temp1 = rotateLeft(LSB4(C0) ^ Y1, 4 ) ^ Y0
(4) temp2 = rotateLeft(MSB4(C0) ^ Y0, 14) ^ Y1
(5) C0 = V1 ^ QWORD(temp2, temp1)
(6) Y2, Y3 = func_j(LSB4(C0), MSB4(C0))
(7) temp3 = rotateRight(LSB4(C1)      ^ Y2, 6)
(8) temp4 = rotateRight(MSB4(C1) ^ Y3 ^ Y2, 14)
(9) C1 = V2 ^ QWORD(temp4, temp3)
```

La fonction `func_h` s'inverse comme détaillé dans le code de la fonction `reverse_func_h` qui suit. Toutes les fonctions utilitaires sont indiquées en Annexe D.

```
def reverse_func_h(B0, B1, C0_14, C1_14):
   c1_new = C1_14
   c0_new = C0_14
   for h in range(15):
       V1, V2 = func_m(B0, B1, 14-h)                #(1)

       t4, t3 = split_QWORD(c1_new^V2)              #from (9)

       Y2, Y3 = func_j(LSB4(c0_new), MSB4(c0_new))  #from (6)

       tmp = rotateLeft(t3, 6)                      #from (7)
       c1_lsb = tmp^Y2 #LSB4(c1_old)

       tmp = rotateLeft(t4, 14)                     #from (8)
       c1_msb = tmp^Y3^Y2 #MSB4(c1_old)
       c1 = QWORD(c1_msb, c1_lsb)

       t2, t1 = split_QWORD(c0_new^V1)              #from (5)

       Y0, Y1 = func_j(LSB4(c1), MSB4(c1))          #from (2)
```

```
        tmp = t2^Y1                                     #from (4)
        tmp = rotateRight(tmp, 14)
        c0_msb = tmp ^ Y0

        tmp = t1 ^ Y0                                   #from (3)
        tmp = rotateRight(tmp, 4)
        c0_lsb = tmp ^Y1
        c0 = QWORD(c0_msb, c0_lsb)

        c1_new = c1
        c0_new = c0

    return c0, c1 #C0_0 and C1_0
```

Finalement, il reste plus qu'à inverser la fonction `func_a` 4 fois, le plus dur étant fait avec la fonction `func_h`.

```
#Solution
v0, v1, v2, v3 = 0x65850b36e76aaed5, 0xd9c69b74a86ec613, 0xdc7564f1612e5347,
0x658302a68e8e1c24

for i in range(4):
    bf0, bf1 = func_b(v0, v1)
    v2_old, v3_old = reverse_func_h(bf0, bf1, v2, v3)

    bf0, bf1 = func_b(v2_old, v3_old)
    v0_old, v1_old = reverse_func_h(bf0, bf1, v0, v1)

    v0, v1, v2, v3 = v0_old, v1_old, v2_old, v3_old

print("input reversed found")
print("v0: " + hex(v0))
print("v1: " + hex(v1))
print("v2: " + hex(v2))
print("v3: " + hex(v3))
```

On obtient alors la chaîne à passer en entrée qui permet d'obtenir la bonne chaîne "That's the correct flag :)" :

```
53535449437b44773472665f564d5f31735f636f306c5f69736e5f745f49747d
```

```
# ./decrypted_file $(python -c "print('\x53\x53\x54\x49\x43\x7b\x44\x77\x34\x72\
x66\x5f\x56\x4d\x5f\x31\x73\x5f\x63\x6f\x30\x6c\x5f\x69\x73\x6e\x5f\x74\x5f\x49\
x74\x7d')")
That's the correct flag :)
```

En ajoutant la clé au keystore, on obtient alors le 3e flag.

```
# ./add_key.py 53535449437b44773472665f564d5f31735f636f306c5f69736e5f745f49747d
[+] 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
#
```

Flag 3: SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}

Ah, "Secure OS". ça sent la TrustZone d'ARM… l'étape suivante va être corsée !

Étape 4

Exploration du binaire

Je redémarre comme indiqué dans l'émulation du téléphone, et je m'aperçois que les clés du `keystore` permettent de déchiffrer plusieurs images. Le binaire de l'étape 4 demande une chaîne hexadécimale représentant 32 octets, sans doute la clé à ajouter au `keystore` du téléphone. J'ouvre ce fichier dans `Ghidra`, je trouve la fonction `main`. Voici en gros le code décompilé par `Ghidra`, les noms des fonctions ont été ajoutés après analyse..

```
if ((iParm1 == 2) && (lVar2 = strlen(puParm2[1]), lVar2 == 0x40)) {
  iVar1 = check_hexstr(puParm2[1],&local_38,0x20);
  if (iVar1 == 0) {
    write("can\'t decode hex");
    uVar3 = 1;
  }
  else {
    local_4 = open_at("/dev/sstic",0);
    local_18 = (undefined8 *)&DAT_0044dbd8;
    local_10 = 0x101010;
    ioctl((ulonglong)local_4,0xc0105300,&local_18);
    local_18 = &local_38;
    local_10 = 0x20;
                /* passing input argument string */
    ioctl((ulonglong)local_4,0xc0105301,&local_18);
    do {
      ioctl((ulonglong)local_4,0xc0105302,0);
      local_8 = ioctl((ulonglong)local_4,0xc0105303,0);
      if ((local_8 & 0xffff) == 1) {
        if ((local_8 & 0xffff0000) == 0) {
                /* GOOD FLAG: ret = 0x00000001 */
          write("Win");
        }
        else {
                /* BAD FLAG: ret = 0x00010001 */
          write("Loose");
        }
        goto LAB_00400910;
      }
    } while ((local_8 & 0xffff) != 0xffff);
    write("Failure");
```

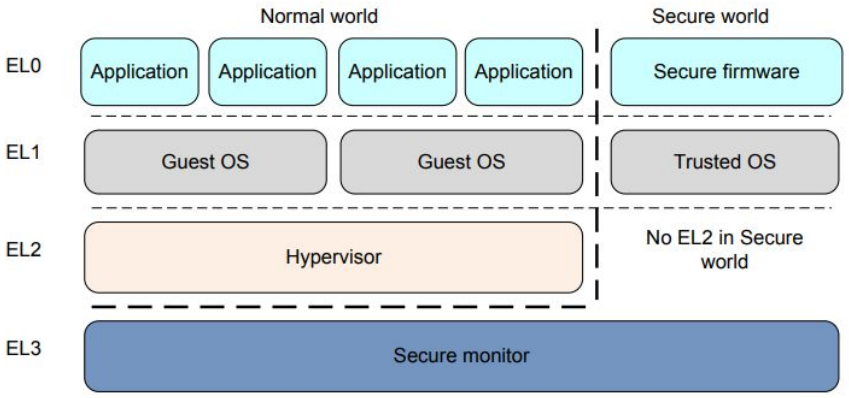
Le binaire semble ouvrir un périphérique appelé `sstic`, avec qui il discute. J'ai lancé un `strace` pour voir les appels systèmes, et j'ai pu constater que beaucoup de `ioctl` sont exécutés, notamment les deux derniers contenus dans la boucle `do/while` (ce qui est conforme au code ci-dessus). Les différents `ioctl` sont identifiables par des numéros. Le code de la fonction que j'ai appelée `ioctl` comporte une instruction `svc 0x0`, ou Supervisor Call.

Il est temps de se documenter...

AARCH64 et les Exception Levels en bref

Au cours de la résolution de cette étape, j'ai beaucoup parcouru le site web de ARM à la recherche d'informations. J'ai même commencé par la mauvaise spécification, ne connaissant pas les distinctions entre ARM-A, ARM-R, ARM-M. Puis, une fois que j'avais compris que c'était ARMv8-A, je n'avais pas la toute dernière version complète, avec les différentes extensions. J'ai fini par utiliser la spécification "ARM Architecture Reference Manual" pour le profil ARMv8-A **[8]**. En passant, une documentation qui m'a beaucoup aidé à démarrer est celle intitulée "ARMv8-A Fundamentals" **[9]**. Ce qui suit décrit ma compréhension des choses[2].

L'architecture ARMv8-A définit 4 Exception Levels, comme décrit dans le schéma ci-dessous (issu de la documentation "ARMv8-A Fundamentals" **[9]**.



Le binaire s'exécute dans le Normal World (NW), au niveau de privilège le plus bas, soit EL0. En exécutant une instruction `svc`, la main passe au superviseur au niveau EL1, soit le kernel Linux qui traite les appels systèmes (comme les syscall traditionnels). La documentation décrit également les instructions `hvc` et `smc`, respectivement pour Hypervisor Call et Secure Monitor Call. Le challenge n'utilise pas de `hvc`, mais il y a des `smc`. Les `smc` permettent de passer du niveau EL1 au niveau EL3, au niveau le plus privilégié. Le Monitor est ce qui permet de faire la transition entre le NW et le Secure World (SW). Généralement, un OS appelé Trusted OS tourne dans le SW, sur lequel peuvent tourner des applications comme dans le NW, appelées Trusted Apps, afin d'exécuter du code plus sécurisé/sensible.

Les instructions `svc` et `smc` génèrent des exceptions qui sont gérées par le niveau de privilège plus élevé. Le retour se fait par une instruction `eret` (ou exception return).

Enfin, EL0/EL1 et EL0(S)/EL1(S) se distinguent par un bit NS (Non-Secure) du registre `SCR_EL3`, indiquant dans quel World le code s'exécute. Maintenant qu'on en sait un peu plus, retour au challenge.

[2] ...pas garanti 100% exact.

Analyse du fichier sstic.ko

L'étape suivante consiste à trouver le driver avec lequel le binaire discute. Je me suis un peu documenté sur les modules kernel, sans trop savoir ce que je cherchais. La commande `lsmod` m'informe qu'il y a bien un module nommé `sstic` qui s'exécute, mais je ne suis pas plus avancé. J'ai fini par tenter un `ls` avec un `grep sstic`, pour finalement trouver un fichier `sstic.ko` (je suppose que l'extension `ko` signifie probablement kernel object). Une fois ouvert avec `Ghidra`, le fichier `sstic.ko` n'étant pas strippé, je trouve la fonction `sstic_ioctl` qui est probablement le gestionnaire des appels `svc` de `decrypted_file`.

En utilisant les identifiants passés en paramètre des `ioctl` du binaire EL0, après analyse, je constate des appels *`smc`* pour chaque `svc`. Chaque `smc` comporte un identifiant, différent de celui des `svc`. Voici la correspondance entre les identifiants `svc` et les identifiants `smc`.

```
ioctl 0: svc 0xc0105300   <======> smc 0x83010004
ioctl 1: svc 0xc0105301   <======> smc 0xf2005003
ioctl 2: svc 0xc0105302   <======> smc 0xf2005001
ioctl 3: svc 0xc0105303   <======> smc 0xf2005002
```

De la même manière que pour les `svc`, il s'agit maintenant de trouver le code gérant des appels `smc`.

Déchiffrement des fichiers BL2, BL31, BL32

N'ayant rien trouvé de plus utile dans le système de fichiers du téléphone, je suis allé fouiller du côté des fichiers `rom.bin` et `flash.bin` qui étaient fournis dans l'archive du challenge. Le fichier `rom.bin` contient le bootloader, qui est exécuté par `QEMU`. Le fichier `flash.bin` semble comporter le code recherché, qui est déchiffré lors du boot par le `keystore`. D'après les logs de `QEMU`, il y a 3 fichiers qui sont déchiffrés, que j'ai fini par nommer après débogage avec `gdb`: `BL2`, `BL31` et `BL32`.

J'ai ouvert `flash.bin` dans un éditeur hexadécimal, à la recherche d'informations dans l'en-tête. J'y trouve beaucoup de séquence d'octets nuls. Je finis par déduire qu'il y a une valeur représentant un offset et la longueur. Il y a 4 grosses sections décrites, que je pense être `BL2`, `BL31`, `BL32`, et le système de fichiers du téléphone.

```
01 00 64 aa => ?
78 56 34 12 => magic or something likewise
00 00 00 00 00 00 00 00 => separator

5f f9 ec 0b 4d 22 3e 4d => ?
a5 44 c3 9d 81 c7 3f 0a => ?
d8 00 00 00 00 00 00 00 => offset 0xd8
30 94 00 00 00 00 00 00 => length 0x9430
                        => end offset 0x9508
00 00 00 00 00 00 00 00 => separator
```

```
47 d4 08 6d 4c fe 98 46 => ?
9b 95 29 50 cb bd 5a 00 => ?
08 95 00 00 00 00 00 00 => offset 0x9508
a0 90 00 00 00 00 00 00 => length 0x90a0
                        => end offset 0x0125A8
00 00 00 00 00 00 00 00 => separator

05 d0 e1 89 53 dc 13 47 => ?
8d 2b 50 0a 4b 7a 3e 38 => ?
a8 25 01 00 00 00 00 00 => offset 0x0125a8
70 53 00 00 00 00 00 00 => length 0x0x5370
                        => end offset 0x017918
00 00 00 00 00 00 00 00 => separator

d6 d0 ee a7 fc ea d5 4b => ?
97 82 99 34 f2 34 b6 e4 => ?
18 79 01 00 00 00 00 00 => offset 0x017918
20 00 22 02 00 00 00 00 => length 0x02220020
                        => end offset
```

J'ai donc le découpage des zones pour les fichiers ainsi que les clés, il n'y a plus qu'à déchiffrer. D'après les logs de `QEMU`, c'est de l'AES. Je code un script, avec les clés représentant en valeur ASCII deux flags précédents, je tente de déchiffrer avec un IV nul, et… cela ne marche pas. AES-ECB, AES-CBC, rien n'y fait. J'ai peut-être un problème d'IV.

En ajoutant les options `-s -S` lors de l'exécution de `QEMU`, il est alors possible de le mettre en attente jusqu'à ce qu'un débogueur y soit attaché. `QEMU` attend que le débogueur soit attaché pour continuer l'exécution. Par ailleurs, j'ai dû utiliser `gdb` pur et dur, et désactiver l'extension `gef`, car visiblement il y a de nombreuses choses manquantes qui l'empêchent de fonctionner correctement.

Avec le débogueur et le code de `rom.bin`, je finis par trouver par trouver l'adresse de la fonction qui déchiffre (fichier `rom.bin`, adresse `0x2c64`), et je constate que l'IV n'est autre que les 16 premiers octets de chaque fichier. Je retente avec les bons IV et de l'AES-CBC, et cette fois, ça marche.

`BL2` ne contient rien de pertinent pour cette étape du challenge, ce n'est, de ma compréhension, qu'un second bootloader, chargé par rom.bin, afin de charger `BL31`. En traçant l'exécution avec gdb, j'ai pu voir que BL2 était mappé à l'adresse `0xe00b000`.

En faisant une recherche sur l'Internet, à propos des termes `BL31`, `BL32`, je finis par tomber sur une implémentation open source d'un Trusted Firmware par ARM **[10]**, qui m'aide dans le reverse des fichiers `BL31` et `BL32`.

Analyse des fichiers BL31 et BL32

Les fichiers `BL31` et `BL32` contiennent un bout de code agissant comme bootloader pour charger d'autres portions de code. `BL31` charge le code de `BL32`, mais aussi le code s'exécutant dans EL3. `BL32` charge le code s'exécutant dans le Secure World en EL0(S)/EL1(S).

Le code de `BL31` est mappé à l'adresse `0xe030000`. Celui de `BL32` est mappé à l'adresse `0xe200000` Pour le savoir, j'ai tracé le code pour voir grosso modo quelles étaient les fonctions que le code exécutait, et à chaque changement d'adresse dans les octets de poids fort, j'ai affiché les premières instructions à partir du Program Counter afin de voir à quel fichier j'avais affaire (et de correctement nommer les fichiers).

Pour la suite, je ne peux pas décrire étape par étape tout le processus de reverse que j'ai suivi. Cela a pris du temps, j'ai progressé dans la compréhension de jour en jour, revenant le lendemain sur ce que j'avais compris le jour précédent pour comprendre quelque chose de plus. J'ai fini par avoir une compréhension de plusieurs blocs de code, avant d'arriver à dresser le schéma global du fonctionnement du code. De ce fait, j'opte pour une description des principaux blocs fonctionnels, avant de donner une idée du schéma global.

Gestionnaires des smc (BL31)

Jusqu'à présent, j'ai vu 4 `smc` (dans le fichier `sstic.ko`), avec des identifiants dont l'octet de poids fort est `0xf200`, et un `smc` avec un identifiant dont l'octet de poids fort est `0x8301`. La documentation ARM "SMC Calling Convention" **[11]** aide à la compréhension, sans toutefois permettre d'y voir complètement clair, mais j'ai pu en tirer quelques informations. Par exemple, l'identifiant permet de savoir que tous les `smc` sont des Fast Calls (grâce au bit de poids fort), c'est-à-dire que ce sont des opérations atomiques d'un point de vue de l'élément qui appelle le `smc`. En bref, le code du `smc` est exécuté, et lorsqu'il a fini, il retourne la main à l'élément appelant. En croisant ce document avec l'implémentation open source du Trusted Firmware **[10]**, j'ai pu identifier deux gestionnaires pertinents.

Le premier gestionnaire est la fonction `tspd_smc_handler` (fichier `BL31`, adresse `0xe034158`), qui d'après l'implémentation du Trusted Firmware et la convention, gère tous les appels `smc` provenant du Normal World. Mais ce n'est qu'une convention, et elle n'est pas forcément respectée par les concepteurs du challenge.

Le deuxième gestionnaire est la fonction que j'ai nommée `oem_smc_handler` (fichier `BL31`, adresse `0xe032014`) par rapport aux identifiants traités. Celle-ci filtre les identifiants avec le deuxième octet de poids faible, et appelle une fonction que j'ai nommée `aese_aesd_process`. Cette dernière possède un immense `switch/case` avec un nombre conséquent d'instructions.

Instructions SIMD[3] aesd et aese (BL31)

Dans la fonction `aese_aesd_process`, on trouve notamment les instructions `aese`, et `aesd`, qui d'après la spécification ARMv8-A, implémentent une ronde partielle d'AES en mode chiffrement (`aese`), ou déchiffrement (`aesd`), sans l'étape du MixColumn. Ce qui est normal puisque toutes les rondes ne comportent pas d'étape MixColumn. Il existe d'ailleurs une instruction dédiée au MixColumn, non utilisée dans le challenge.

En instrumentant dynamiquement le code avec `gdb`, j'ai fini par comprendre qu'une clé AES est chargée dans le vecteur `v1`, 4 octets par 4, depuis des registres spéciaux. La fonction qui permet de mettre les valeurs dans ces registres spéciaux est située à l'adresse `0xe030fe4`, fichier `BL31`. Il s'agit des registres FPEXC32_EL2, DACR32_EL2, IFSR32_EL2 et SDER32_EL3. De plus, en considérant une chaîne en entrée comme celle-ci, `A1A2A3A4A5A6A7A8A9AAABACADAEAFA0B1B2B3B4B5B6B7B8B9BABBBCBDBEBFB0`, la clé chargée correspond à la sous-chaîne `ADAEAFA0B1B2B3B4B5B6B7B8B9BABBBC`.

De plus, les concepteurs du challenge étant assez sournois, l'instruction `aese` est utilisée pour du déchiffrement, et `aesd` pour du chiffrement.

Instruction SIMD fcadd (BL31)

Dans cette même fonction `aese_aesd_process`, on trouve l'utilisation de l'instruction `fcadd`. Elle réalise l'addition de deux nombres complexes à virgule flottante en une seule instruction. De plus, le deuxième argument de l'instruction peut subir une rotation de 90° ou 270° par rapport à l'origine du plan complexe. L'utilisation de cette instruction a servi à obfusquer d'une certaine manière des opérations de soustraction.

Exécution de code dans BL32

Lors de l'instrumentation dynamique, j'ai pu constater que du NW EL0, on passe au NW EL1, puis, à EL3, et ensuite qu'on se retrouve dans EL1(S). À ce moment-là, je n'ai pas réussi à trouver le bon registre me permettant de vérifier dans quel niveau d'EL le code était exécuté. D'après la documentation, le registre en AARCH64 est un registre virtuel nommé PSTATE, et il y a deux bits PSTATE.CurrentEL permettant de le savoir, mais `gdb` ne le sait pas[4]. Par conséquent, pour vérifier j'ai patché en direct l'instruction en cours dans `gdb` pour lire le bon registre, à l'aide de l'instruction `mrs` (ce qui veut dire move system register to register).

[3] La technologie ARM NEON, également appelée Advanced SIMD (Single Instruction Multiple Data), ou encore MPE (Media Processing Engine). permet d'étoffer le jeu d'instructions existant avec des instructions permettant de faire des opérations vectorielles dédiées et optimisées pour le traitement de signal et média, ou autre application scientifique. En plus des registres 64 bits x0, x1, x2, etc. d'autres registres appelés vecteurs permettent de gérer des données de taille 128 bits, v0, v1, v2, etc. De plus, ils sont adressables par octets, mots, double mots, etc. (cf. documentation ARMv8-A **[8]**).

[4] Beaucoup plus tard, je me suis aperçu que j'utilisais une vieille version de `gdb` (7.x), et que la dernière était la 8.2, que je n'ai pas réussi à compiler dans ma VM Ubuntu 16. Je pense c'est peut-être la raison pour laquelle je n'avais pas facilement accès à l'information ….

La fonction dans laquelle l'exécution de code se poursuit depuis EL3 (celle par qui le code passe directement du NW au SW) se situe à l'adresse `0xe200c08`, fichier `BL32`. Dans cette fonction, les deux octets de poids faible des identifiants des `smc` provenant de EL1 sont filtrés (la gestion du 1er `ioctl` n'est pas faite ici), et des instructions `smc` sont appelées, avec d'autres identifiants. (`0x83010001`, `0x83010002`, `0x83010003`). La gestion des 2e et 3e `ioctl` se fait via ces 3 `smc`. En ce qui concerne le 4e `ioctl`, une autre fonction est appelée, à l'adresse `0xe2005a4`, contenant encore plus de `smc` et d'identifiants divers (que j'ai nommée `smc_ioctl_final_handler`).

Instructions SIMD sm4ekey et sm4e (BL32)

La fonction située à l'adresse `0xe200e84` (nommée `process_SM4`), comporte deux instructions qui m'ont intrigué : `sm4ekey` et `sm4e`. D'après la documentation ARM, ce sont des instructions qui exécutent des bouts de l'algorithme chinois appelé SM4. Une excellente description accompagnée de schémas est disponible sur le site des outils de l'IETF **[12]**.

En très bref, c'est un algorithme qui traite des blocs de 128 bits, avec une clé 128 bits. Il y a 32 rondes, avec une dérivation de clé non linéaire par une S-box, et à chaque ronde, un bloc de 32 bits est traité.

Plusieurs choses sont à noter. L'instruction `sm4ekey` permet de dériver 4 sous-clés en une instruction, et `sm4e` exécute 4 rondes de chiffrement en une instruction à partir de 4 sous-clés spécifiées. Il n'y a pas d'instruction de déchiffrement à proprement parler : pour effectuer le déchiffrement, les sous-clés sont utilisées en ordre inverse. D'autre part, la clé est écrite dans le binaire `BL32`, à l'adresse `0xe204048`. Enfin, une subtilité supplémentaire a été ajoutée : un `xor` est effectué entre l'adresse des données modulo 16, et la valeur de clé. Ce `xor` est effectué 4 fois pour chaque 4 octets de la clé, puisque l'adresse est codée sur 4 octets.

L'implémentation des instructions `sm4ekey` et `sm4e` est disponible en Annexe E.

Interruption du flux d'exécution du code dans gdb

En instrumentant le code, je me suis fait surprendre à plusieurs reprises par des ruptures du flux d'exécution. Après investigation, il s'agit d'accès mémoire provoquant un Data Abort Exception. La fonction que j'ai nommée `data_abort_exception_handler` à l'adresse `0xe203204`, fichier `BL32`, est alors exécutée, et fait appel à la fonction `process_SM4`. L'adresse utilisée pour l'accès mémoire est passée en paramètre de la fonction `process_SM4`, en lisant le registre FAR_EL1 (Fault Adress Register). La valeur de l'adresse de retour est contenue dans le registre ELR_EL1. Cette valeur est alors incrémentée de 4 afin de permettre au code de poursuivre l'exécution à l'instruction suivant celle qui a généré l'exception.

Gdb bloqué sur l'instruction eret : découverte des instructions AARCH32

Le fait de stepper avec `gdb` et de voir que l'exécution a l'air de boucler sur `PC=0Xe20260c` (instruction `eret`) m'a laissé dubitatif pendant un long moment. De plus, le nombre le nombre de steps nécessaire pour sortir de la boucle n'était pas le même. Finalement, j'ai fini par comprendre qu'il s'agissait d'instructions AARCH32 exécutées en EL0(S). Dans la fonction à l'adresse `0xe2025a4` (nommée `switchAARCH_64_to_32`), la valeur `0x1d0` est écrite

dans le registre SPSR_EL1, Le bit 4, lorsqu'il vaut 1, indique qu'il s'agit d'une exception provoquée depuis AARCH32, alors que lorsqu'il vaut 0, c'est AARCH64. Après avoir dupliqué le fichier `BL32`, ouvert dans `Ghidra` mais en spécifiant ARM et non AARCH64, on peut alors désassembler les instructions à partir de l'adresse `0xe205000` (déterminée par l'instrumentation, par une analyse plus poussée de la fonction traitant le 4e `ioctl` dans `BL32`, `smc_ioctl_final_handler`).

À partir des adresses `0xe205000`, on peut constater qu'il y a des instructions `swi 0x1338` et `swi 0x1337`. Ce sont des interruptions permettant de passer de EL0(S) vers EL1(S). Lorsque ces instructions sont exécutées, la fonction nommée `SWI_from_aarch32` à l'adresse `0xe203604`, fichier `BL32`, est exécutée. En l'analysant on s'aperçoit que lorsque la valeur est `0x1338`, un `smc` est appelé (pour passer à EL3), tandis que pour la valeur 0x1337, un simple `ret` est effectué. De plus, le code qui s'exécute en AARCH32 ne fait que mettre des identifiants qui sont passés en paramètre pour les `smc`. Pour finir, on y trouve aussi quelques instructions en mode THUMB.

Anti-debug: valeur à l'adresse 0x9010000 (AARCH32)

Dans les bouts de code, je me suis aperçu qu'il y avait une instruction qui allait écrire une valeur à l'adresse `0x9010008`, ainsi qu'une autre qui chargeait la valeur contenue à l'adresse `0x9010000`. L'exécution de code prenait une branche ou non en fonction de si la valeur dépassait 5. Par pur hasard, en affichant la valeur dans cette adresse, je me suis aperçu que la valeur est incrémentée à chaque seconde !

En mesurant l'exécution du binaire sans gdb, le binaire met moins de 5 secondes pour retourner la chaîne finale (Win/Loose). par contre, avec des points d'arrêts et un script pour collecter les valeurs des registres, l'exécution dépassait largement ces 5 secondes.

Pour me défaire de ce piège des concepteurs, j'ai patché les instructions suivantes dans mon script `gdb` :

```
[...] adresse 0e2050cc
movw        r9,#0x901
mov         r9,r9, lsl #0x10     ;; r9 = 0x9010000
ldr         r9,[r9,#0x0]         ;; r9 contient la valeur à l'adresse 0x9010000
[...] adresse 0e2050e4
cmp         r9,#0x5              ;; comparaison avec la valeur 5
addle r0, r0, #3                 ;; instruction patchée en add r0, r0, #3
cpygt r0, r8                     ;; instruction patchée en nop
```

Fonctionnement global

Trop long pour être décrit en détail, voici les principaux éléments du fonctionnement global[5].

Au niveau EL0, des appels `ioctl` sont faits à destination de EL1. Le premier `ioctl` se termine par un smc à destination de EL3, qui ne fait que copier une zone du binaire du NW `decrypted_file` vers une autre zone (smc 0x83010004). La taille de la copie fait `0x101010` octets.

Puis, pour les 2e, 3e et 4e `ioctl` (resp. `smc 0xf2005003`, `0xf2005001`, `0xf2005002`), l'exécution de code va du niveau EL1 vers EL3, puis vers EL1(S) (possiblement EL0(S)), avant de revenir par EL3 à EL1 et EL0. Selon les `ioctl`, il y a un jeu de yoyo entre EL3 et EL1(S) via des `smc` et des `eret`.

Le deuxième `ioctl` traite la chaîne passée en paramètre de l'input. et la copie en mémoire après l'avoir chiffrée. Il est à noter que des `smc` supplémentaires sont effectués afin de stocker la clé de l'AES 4 octets par 4.

Enfin, les 3e et 4e `ioctl` bouclent, jusqu'à obtenir une valeur de retour du 4e `ioctl` dont le bit de poids faible est mis à 1. Le 3e `ioctl` permet de déchiffrer avec `aese` (via `smc 0x83010001`) un offset, qui est ensuite traité par le 4e `ioctl`.

L'exécution du 4e `ioctl` mène dans EL1(S) à la fonction `smc_ioctl_final_handler`, qui représente en fait une VM traitant des instructions codées sur 3 octets. Ces instructions sont situées dans la première portion chiffrée de `decrypted_file`, à partir de l'adresse `0x4dbd8` de ce même binaire. À chaque 4e `ioctl` exécuté, une instruction de la VM est exécutée,

La difficulté de cette étape consiste à reproduire cette VM, étant donné l'obfuscation basée sur la dissémination du code entre les différents EL et fichiers. Dans la fonction reproduisant la VM, selon les instructions, du code AARCH32 est appelé via une fonction qui prend en paramètre l'adresse du code AARCH32. D'autre part, des `smc` sont également appelés avec un identifiant, qui est ensuite traité par la fonction `aese_aesd_process`. Cette dernière, en fonction de l'identifiant du `smc`, déchiffre et/ou chiffre une ou plusieurs données de la pile de la VM, en y incluant ou non un traitement sur les opérandes déchiffrées.

Pour reproduire cette VM, je me suis armé de patience et j'ai littéralement fait du reverse à la main[6].

[5] Des schémas auraient été plus judicieux, certes.

[6] Une photo bonus est insérée en dernière page du rapport. J'ai fait usage du papier car je n'arrivais pas à avoir une vision d'ensemble claire et ergonomique via Ghidra ou via d'autres éditeurs de texte.

Une des autres difficultés de cette étape est qu'il y avait deux chiffrements :

- un pour protéger les données manipulées par la VM, avec de l'AES dont la clé était spécifiée par une sous-chaîne de la chaîne passée en entrée du binaire `decrypted_file` ;
- un pour déchiffrer les instructions avec SM4, mais aussi des données depuis la zone copiée du binaire `decrypted_file`. D'ailleurs, j'ai constaté un décalage étrange de `0x1000` lorsqu'il s'agissait d'aller récupérer les données à déchiffrer. Peut-être un anti-debug ? En tout cas, pour déboguer, j'ai mis un patch temporaire dans mon implémentation de cette VM pour simuler le décalage de `0x1000`, que j'ai fini par enlever une fois que j'avais l'émulateur de la VM opérationnel. Je cherche toujours l'explication…

Les données manipulées par la VM sont identifiées par un nombre entre `0x0` et `0xf`. L'adresse de chaque donnée est déterminée par une adresse de base, à laquelle on applique un offset de l'identifiant, multiplié par 16. Par exemple, le Program Counter de la VM, identifié par `0xf`, était stocké à l'adresse de base + `0xf0`. Plus simplement, cela représente une VM à 16 registres.

Les concepteurs du challenge ont ajouté le chiffrement AES afin de rendre plus difficile le suivi de l'évolution des valeurs des registres de la VM. Les `smc 0x83010001` et `0x83010002` étaient le plus fréquemment appelés : le premier servait à aller lire la valeur d'un registre en la déchiffrant, et le deuxième à écrire une valeur dans un registre en la chiffrant.

Enfin, une fois l'émulateur fonctionnel, j'y ai ajouté du pseudo-code pour décrire l'utilité de chaque instruction de la VM. Par exemple, `data[0x3] = 0x20`, qui voulait dire placer `0x20` dans le 4e registre. Puis j'ai lancé l'émulation des instructions. À la fin de l'exécution, l'émulateur m'avait produit un fichier retraçant l'ordre des instructions, qui m'a permis d'effectuer le reverse final.

Puis, de la même manière que pour l'étape 3 du challenge, j'ai codé l'inverse de la fonction qui transformait la chaîne en entrée, afin de remonter à celle qui produit les valeurs finales. La chaîne en entrée (32 octets) était transformée, puis stockée aux adresses `0x513000` à `0x51301F`. Les bonnes valeurs à obtenir étaient stockées aux adresses `0x513020` à `0x51303F`.

Tous les scripts utilisés sont mis sur un GitHub, trop longs pour être copiés dans ce rapport.

Je finis par obtenir la clé permettant de valider la vérification faite par le binaire :

Clé finale : acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e

Et voici le 4e flag.

Flag 4 : SSTIC{acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e}

Et enfin...

Pour finir, une fois le dernier dossier `/root/safe_03/` déchiffré, la commande `strings * | grep sstic` permet d'obtenir l'adresse mail finale. Après avoir envoyé le mail, je suis allé voir ce qui contenait l'adresse mail. Il s'agissait en fait de sms stockés dans un fichier de base de données du téléphone, en lien avec le texte de la présentation du challenge. La mission est remplie !

Adresse mail finale : 9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org

Épilogue

Merci de m'avoir lu ! Un grand merci aux concepteurs du challenge SSTIC 2019, j'ai beaucoup appris au cours de ce challenge !

Je ne pourrai pas nommer tous les registres AARCH64 que j'ai découverts au cours de la dernière étape du challenge, mais il y en a beaucoup. La dernière version de la documentation ARMV8-A **[8]** est très fournie (plus de 7000 pages), et elle m'a permis de comprendre progressivement plus en profondeur le fonctionnement AARCH64. Pas si évidente à prendre en main de premier abord, on finit par s'y faire un peu plus chaque jour, en allant s'intéresser à un registre, ainsi qu'à ceux qui gravitent dans le même contexte que ce premier. Enfin, il faut veiller à être dans le bon chapitre AARCH64 et non AARCH32 ; à force d'utiliser abusivement le Ctrl+F, on finit par avoir des surprises !

Sources utilisées

[1] Site du challenge SSTIC 2019, description et archive du challenge :
https://www.sstic.org/2019/challenge/

[2] Lien de téléchargement de la version de `QEMU` 3.1.0 :
https://download.qemu.org/qemu-3.1.0.tar.xz

[3] Présentation "*C++ Exceptions and Stack Unwinding*" par Dave Watson à la conférence CppCon 2017
https://www.youtube.com/watch?v=_Ivd3qzgT7U&t=913s

[4] Spécifications DWARF v4 et v5 :
http://dwarfstd.org/doc/DWARF4.pdf et http://dwarfstd.org/doc/DWARF5.pdf

[5] Article "*Exploiting the hard-working DWARF: Trojan and Exploit Techniques With No Native Executable Code*" de Oakley et Bratus :
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.651.2604&rep=rep1&type=pdf

[6] Projet `LIEF` :
https://github.com/lief-project/LIEF

[7] Projet `libgcc` de gcc-mirror :
https://github.com/gcc-mirror/gcc/tree/master/libgcc

[8] Spécification "*ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*" :
https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile

[9] Documentation ARM "*Fundamentals of ARMv8-A*" :
https://developer.arm.com/docs/100878/latest/fundamentals-of-armv8-a

[10] Implémentation de référence du Trusted Firmware pour Armv7-A et Armv8-A :
https://github.com/ARM-software/arm-trusted-firmware

[11] Documentation ARM "*SMC Calling Convention*" :
http://infocenter.arm.com/help/topic/com.arm.doc.den0028b/ARM_DEN0028B_SMC_Calling_Convention.pdf

[12] Description de l'algorithme SM4 par le Crypto Forum Research Group, "*The SM4 Blockcipher Algorithm And Its Modes of Operations*":
https://tools.ietf.org/id/draft-ribose-cfrg-sm4-04.html

Annexes

Le code complet et utile de mes scripts est disponible sur le Github suivant :

https://github.com/Aterwyn/SSTIC2019

Annexe A : Étape 1 - Implémentation de l'attaque SPA sur la clé RSA

Un jupyter notebook avec une présentation plus visuelle est présent sur le Github.

```
from numpy import load
import matplotlib.pyplot as plt
%matplotlib inline
data = load("power_consumption.npz")
lst = data.files
real_data = data[lst[0]]
new_data = real_data[700000-1000:1489000+1000]
THRESHOLD = -0.145
#d'abord, trouvons le premier pic.
first_peak = 0
for i in range(len(new_data)):
   if new_data[i] <= THRESHOLD:
       first_peak = i
       print(first_peak)
       break
idx = first_peak
l2 = []
WINDOW = 500
SLIDE = 600

debug_bits_counter = 0
#tant qu'on n'a pas regardé toute la courbe
while ((idx < len(new_data)) and (idx + WINDOW < len(new_data))):
   if debug_bits_counter == 1024:
       print(idx)
   sum = 0
   idx_window = idx
   while (idx_window < idx + WINDOW):
       if (new_data[idx_window] <= THRESHOLD):
           sum += 1
           idx_window += 5 #pour éviter de compter un pic plusieurs fois
       else:
           idx_window+=1
   if sum == 2: #petit pattern détecté
```

```
        l2.append(0)
    elif sum >= 3: #gros pattern détecté
        l2.append(1)
    else:
        break
    idx += SLIDE
    #ajustement sur le pic suivant
    #si on a atteint la fin, alors quitter
    if (idx + WINDOW > len(new_data)):
        break
    for i in range(idx, idx+WINDOW):
        if new_data[i] <= THRESHOLD:
            idx = i
            break
    debug_bits_counter += 1
binary_str = "".join(str(l) for l in l2)
RSA_key = hex(int(binary_str,2))
inv_binary_str = binary_str[::-1]
final_RSA_key = hex(int(inv_binary_str, 2))
final_RSA_key
```

Annexe B : Étape 2 - Implémentation du bruteforce

```
ref_hash = "00c8bb35d44dcbb2712a11799d8e1316045d64404f337f4ff653c27607f436ea"
s1_l = []
s2_l = []
s3_l = []
s4_l = []
s5_l = []
s6_l = []
s7_l = []
s8_l = []
for i in range(16):
    BUTTONS = i
    s1_l.append(step1())
    s2_l.append(step2())
    s3_l.append(step3())
    s4_l.append(step4())
    s5_l.append(step5())
    s6_l.append(step6())
    s7_l.append(step7())
    s8_l.append(step8())

s = 0
total = pow(2,32)
for s1 in s1_l:
  for s2 in s2_l:
     for s3 in s3_l:
        for s4 in s4_l:
           for s5 in s5_l:
              for s6 in s6_l:
                 for s7 in s7_l:
                    for s8 in s8_l:
                       key = bytearray([s1,s2,s3,s4,s5,s6,s7,s8])
                       h = hashlib.sha256(key).hexdigest()
                       if ref_hash == h:
                          print("WIN")
                          print("key: " + key.hex())
                          raise Exception
        s+=1048576
        if ((s+1) %1000000) == 0:
           print("%0.2f %%" % (s*100/(total/8)))
```

Annexe C : Étape 3 - Script gdb/gef, breakpoint sur execute_stack_op

```
gef-remote :12345
b *0x402d30
b __libc_start_main
b __cxa_atexit
b __cxa_begin_catch
b __cxa_throw
c
c
c
c
c
c
n 17
si
n 3
si
n 4
si
n 28
si
b *_Unwind_RaiseException+256
c
si
p/x *$pc+0xb4
b *$1
c
si
n 5
si
p/x *$pc+0x1fc
b *$2
c
c
si
p/x *$pc+0x78
b *$3
c
gef config context.nb_lines_code 0
gef config context.nb_lines_code_prev 0
gef config context.nb_lines_stack 42
gef config context.nb_lines_backtrace 0
c
```

Annexe D : Étape 3 - Implémentation python de l'algorithme codé en DWARF

```python
def minus(op1, op2):
    #compute op1-op2 on 8 bytes
    r = op1-op2
    if r<0:
        return 0x10000000000000000 + r
    return r

def rotateRight(val, n):
    return ((val>>n) | (val<<(32-n))) & 0xFFFFFFFF

def rotateLeft(val, n):
    return ((val<<n) | (val>>(32-n))) & 0xFFFFFFFF

def MSB4(val):
    return (val>>32)&0xFFFFFFFF

def LSB4(val):
    return val&0xFFFFFFFF

def LSB8(val):
    return val & 0xFFFFFFFFFFFFFFFF

def QWORD(valMSB, valLSB):
    return (valMSB<<32) | valLSB

def split_QWORD(qword):
    return (qword>>32)&0xFFFFFFFF, qword&0xFFFFFFFF

def F(B0, B1):
    # lis est tronqué pour économiser de la place dans le rapport
    lis = [0x5963b39b, 0x30f75add, ... 0x8f5fb519, 0x2b7332a5, 0x10aa767c]

    value = lis[MSB4(B1)&0xFF]
    t0 = LSB4(B0) ^ (MSB4(B0) + LSB4(B1))

    t1 = LSB4(t0 + value)
    t2 = LSB4(MSB4(B0) & B1)
    val1 = QWORD(t2, t1)

    t3 = LSB4(minus(LSB4(B1), t0))
    t4 = t1 ^ (MSB4(B1)>>8)
    val2 = QWORD(t4, t3)
```

```
    return val1, val2

def F_bis(B0, B1, m):

    lis = [0xd6378fea, 0xe23ca8c4, 0x84e3b1bc, 0xce5e10bf, 0xa2b364da, 0x41f250f0,
0x0fe97040, 0x1cc05266, 0x16f87e4b, 0x515e26b7, 0xeea48dcb, 0x62b357e4, 0x39bd2041,
0x72cd387a, 0xf37aac8b]

    t0 = LSB4(MSB4(B0)+0x45786532)
    t1 = t0 ^ LSB4(B1)

    if t1 & 0x80000000:
        Zx = 0x60bf080f # 0x84653217
    else:
        Zx = 0x818f694a # 0x17246549

    t2 = rotateLeft(MSB4(B1), 4)
    V2 = QWORD(t2, t1)

    t3 = lis[m] ^ LSB4(B0)
    t4 = LSB4(minus(t3, t0))
    t5 = Zx^t0 ^ t2
    V1 = QWORD(t5, t4)

    return V1, V2

input_key = [0x4242424241414141, 0x4444444443434343, 0x4646464645454545, 0x4848484847474747]

v0 = input_key[0]
v1 = input_key[1]
v2 = input_key[2]
v3 = input_key[3]

def func_j(J0, J1):
    const_list = [0x489dddde, 0x00000000, 0x95bf74a9, 0x067990f1, 0x0e6d80e3, 0x77941ee7,
0xfb92cd42, 0x2dedaf8b, 0xf2b3a3fb, 0xd0e867c0, 0xe74f99e0, 0x6c39ce47, 0xd6378fea,
0x5a24f221]

    for j in range(6):
        J0 = LSB4(J0 ^ LSB4(J1 + const_list[2*j]))
        J1 = LSB4(J1 ^ (QWORD(const_list[2*(j+1)], const_list[2*(j+1)+1]) | J0))

    return J1, J0
```

```
def func_m(V1, V2, h):
    for m in range(h+1):
        V1, V2 = F_bis(V1, V2, m)

    return V1, V2

def func_h(B0, B1, C0, C1):
    for h in range(15):
        V1, V2 = func_m(B0, B1, h)
        Y0, Y1 = func_j(LSB4(C1), MSB4(C1))

        temp1 = rotateLeft(LSB4(C0) ^ Y1, 4 ) ^ Y0
        temp2 = rotateLeft(MSB4(C0) ^ Y0, 14) ^ Y1

        C0 = V1 ^ QWORD(temp2, temp1)
        Y2, Y3 = func_j(LSB4(C0), MSB4(C0))

        temp3 = rotateRight(LSB4(C1)      ^ Y2, 6)
        temp4 = rotateRight(MSB4(C1) ^ Y3 ^ Y2, 14)
        C1 = V2 ^ QWORD(temp4, temp3)

    return C0, C1

def func_b(B0, B1):
    b0, b1 = B0, B1
    for b in range(4):
        b0, b1 = F(b0, b1)
    return b0, b1

def func_a(v0, v1, v2, v3):
    bf0, bf1 = func_b(v2, v3)
    vf0, vf1 = func_h(bf0, bf1, v0, v1)
    bf2, bf3 = func_b(vf0, vf1)
    vf2, vf3 = func_h(bf2, bf3, v2, v3)

    return vf0, vf1, vf2, vf3

def check_sol(v0, v1, v2, v3):
    s0, s1, s2, s3 = 0x65850b36e76aaed5, 0xd9c69b74a86ec613, 0xdc7564f1612e5347,
0x658302a68e8e1c24

    for a in range(4):
        v0, v1, v2, v3 = func_a(v0, v1, v2, v3)
```

```
    print("Solution checking")
    if v0 == s0 and v1 == s1 and v2 == s2 and v3 == s3:
        print("FLAG !")
    else:
        print("Lose")

#program starts here
check_sol(v0, v1, v2, v3)
```

Annexe E : Étape 4 - Implémentation du chiffrement/déchiffrement SM4

```
class SM4:

   def __init__(self):
       self.SM4_key = bytearray.fromhex("0625f824d5dc439cb4c150382078cc93")
       self.CK = [0x00070E15, 0x1C232A31, 0x383F464D, 0x545B6269, 0x70777E85, 0x8C939AA1,
0xA8AFB6BD, 0xC4CBD2D9, 0xE0E7EEF5]

   def b2i(self, bytes_param, sz=4):
       s = 0
       assert sz > 1
       for i in range(0, sz, 1):
           s += bytes_param[i] << (8*(sz-1-i))
       return s

   def i2b(self, int_param, sz=4):
       s_bytes = bytearray.fromhex("")
       assert sz > 1
       for i in range(sz):
           s_bytes = bytearray.fromhex("%02x" % (int_param>>(8*(i)) & 0xFF)) + s_bytes
       return s_bytes

   def def_master_key(self, key_bytes):
       return
self.b2i(key_bytes[12:]),self.b2i(key_bytes[8:12]),self.b2i(key_bytes[4:8]),self.b2i(key_byt
es[:4])

   def rol4_int(self, a_i, rot):
       return ((a_i << rot) & 0xFFFFFFFF) | ((a_i>> (32-rot)) & 0xFFFFFFFF)

   def func_L(self, B_i):
       LB = B_i ^ self.rol4_int(B_i, 2) ^ self.rol4_int(B_i, 10) ^ self.rol4_int(B_i, 18) ^
self.rol4_int(B_i, 24)
       return LB

   def func_L_(self, B_i):
       #B_i is a 4-bytes int
       L_B = B_i ^ self.rol4_int(B_i, 13) ^ self.rol4_int(B_i, 23)
       return L_B

   def subs(self, val):
       S_box = { ...}
       0: [0xD6, 0x90, 0xE9, 0xFE, 0xCC, 0xE1, 0x3D, 0xB7, 0x16, 0xB6, 0x14, 0xC2, 0x28,
```

```
0xFB, 0x2C, 0x05],
       1: [0x2B, 0x67, 0x9A, 0x76, 0x2A, 0xBE, 0x04, 0xC3, 0xAA, 0x44, 0x13, 0x26, 0x49,
0x86, 0x06, 0x99],
       [...] #tronqué pour économiser de la place dans le rapport
       }

       row = (val&0xF0) >> 4
       col = val&0xF
       return S_box[row][col]

   def func_tau(self, a_i):
       a = self.i2b(a_i, 4)
       temp = (self.subs(a[0])<<24) + (self.subs(a[1]) << 16) + (self.subs(a[2]) << 8) +
self.subs(a[3])
       return temp

   def func_T(self, val_i):
       return self.func_L(self.func_tau(val_i))

   def func_T_(self, val_i):
       return self.func_L_(self.func_tau(val_i))

   def func_F(self, x0, x1, x2, x3, rk):
       #xi are 4 bytes bytearray
       #rk is 4 bytes bytearray
       assert len(x0) == 4
       assert len(x1) == 4
       assert len(x2) == 4
       assert len(x3) == 4
       assert len(rk) == 4
       return self.i2b( self.b2i(x0) ^ self.func_T( self.b2i(x1) ^ self.b2i(x2) ^
self.b2i(x3) ^ self.b2i(rk)), 4)

   def key_schedule(self, k_i0, k_i1, k_i2, k_i3, ck_i):
       #consider k_i as 4-byte parametres, representing K_i as int
       return k_i0 ^ self.func_T_(k_i1 ^ k_i2 ^ k_i3 ^ ck_i)

   def decrypt(self, adr, data):
       mk0, mk1, mk2, mk3 = self.def_master_key(self.SM4_key)
       k0, k1, k2, k3 = mk0, mk1, mk2, mk3

       x0, x1, x2, x3 = (data[i*4:i*4+4] for i in range(4))
       xor_val = adr
```

```
        k4 = self.key_schedule(k0, k1, k2, k3, self.CK[0]^xor_val)
        k5 = self.key_schedule(k1, k2, k3, k4, self.CK[1]^xor_val)
        k6 = self.key_schedule(k2, k3, k4, k5, self.CK[2]^xor_val)
        k7 = self.key_schedule(k3, k4, k5, k6, self.CK[3]^xor_val)
        rk3, rk2, rk1, rk0 = k4, k5, k6, k7

        x4 = self.func_F(x0, x1, x2, x3, self.i2b(rk0))
        x5 = self.func_F(x1, x2, x3, x4, self.i2b(rk1))
        x6 = self.func_F(x2, x3, x4, x5, self.i2b(rk2))
        x7 = self.func_F(x3, x4, x5, x6, self.i2b(rk3))

        output = x7+x6+x5+x4
        return output

    def encrypt(self, adr, data):
        mk0, mk1, mk2, mk3 = self.def_master_key(self.SM4_key)
        k0, k1, k2, k3 = mk0, mk1, mk2, mk3

        #data must be 16 bytes
        x0, x1, x2, x3 = (data[i*4:i*4+4] for i in range(4))
        xor_val = adr

        k4 = self.key_schedule(k0, k1, k2, k3, self.CK[0]^xor_val)
        k5 = self.key_schedule(k1, k2, k3, k4, self.CK[1]^xor_val)
        k6 = self.key_schedule(k2, k3, k4, k5, self.CK[2]^xor_val)
        k7 = self.key_schedule(k3, k4, k5, k6, self.CK[3]^xor_val)
        rk3, rk2, rk1, rk0 = k4, k5, k6, k7

        x4 = self.func_F(x0, x1, x2, x3, self.i2b(rk3))
        x5 = self.func_F(x1, x2, x3, x4, self.i2b(rk2))
        x6 = self.func_F(x2, x3, x4, x5, self.i2b(rk1))
        x7 = self.func_F(x3, x4, x5, x6, self.i2b(rk0))

        output = x7+x6+x5+x4
        return output
```

Photo Bonus: le “reverse à la main”