

Solution challenge SSTIC 2019

Tristan Claverie

Table des matières

Introduction	3
Un démarrage bien bavard	4
Une menace plane	4
Le canal d'à côté	5
Un peu de lecture	7
C'est pourtant logique	11
Un système spartiate	11
Dans les couches basses	12
Avec un peu de brutalité	18
Une histoire d'elfes et de nains	20
À une exception près	20
Premiers pas en rétro-ingénierie	20
Pendant ce temps là	22
Une question de spécification	24
Suivre la trace, compter ses pas	26
Un langage bien sympathique	28
Cours de nanique	28
Une machine plutôt simple	30
Un automate passe	32
Plus qu'une inversion	35
Reforger l'algorithme	35
Pas à pas	38
Sauvés par la teinture	39
Environnement d'exécution sécurisé ?	42
Un nouveau participant apparaît	42
Délégation de pouvoir	42
Dans le noir	44
Au commencement, il y avait qemu	46
Bien amorcé	46
Une machine pas sympathique	49
Toujours plus de chiffrement	52
Ingénierie à reculons et en musique	55
Son et lumières	55
Les 32 dernières phrases	60
Retrouver les opérations perdues	60
Ultime tromperie	63
Le mystère s'éclaircit	66
Conclusion	68

Remerciements	68
Annexe A : Script pour l'étape 1	68
Annexe B : Scripts pour l'étape 2	70
Annexe C : Programmes pour l'étape 3	83
Annexe D : Programmes pour l'étape 4	129

Table des figures

1	Trace de consommation	6
2	Zoom sur la trace de consommation	8
3	Superposition de la trace originale et du signal de travail	9
4	Histogramme du nombre de pics suivant le nombre d'échantillons	9
5	Schéma de conception	15
6	Signification des portes logiques	16
7	Cellules logiques	16
8	Circuit logique implémentant la fonction choix	17
9	Temps d'exécution	23
10	L'une des FDE de l'exécutable	25
11	Différences de temps d'exécution	26
12	Flot de contrôle du programme DWARF	31
13	Automate servant à la génération de code	35
14	Analyse par teinte de l'algorithme	40
15	Visualisation de la trace, 100 instructions par ligne	56
16	Visualisation de la trace, 257 instructions par ligne	56
17	Visualisation de la trace, 0x858 instructions par ligne	57
18	Visualisation des 30 premières instructions	59

Introduction

Cette année, le challenge SSTIC nous signale qu'un individu aurait décidé de s'en prendre à la communauté française en sécurité informatique. Afin de mettre cette affaire au clair, nous sommes chargés d'étudier des éléments collectés sur son téléphone portable. Ces éléments se composent d'une image du système, ainsi que d'une trace de consommation capturée lors du démarrage du téléphone.

Comme chaque année, l'objectif final est de trouver une adresse mail à partir des éléments fournis. Ce challenge se compose de quatre étapes qui permettent d'obtenir chacune une clé de déchiffrement permettant de passer à l'étape suivante. Pour finir, une légère analyse forensique permettra de trouver l'adresse mail tant convoitée, en plus d'éclaircir le déroulé d'un drame récent ayant touché la communauté sécurité informatique française. Les motivations de l'attaquant, cependant, restent à ce jour inconnues.

Ce document relate la démarche suivie pour résoudre chacune des étapes. Il se décompose naturellement en cinq parties, une par étape. Lorsque nécessaire, les scripts utilisés pour la résolution seront aussi fournis. Tous les logiciels et bibliothèques utilisés au cours de la résolution sont disponibles en source ouverte gratuitement.

Un démarrage bien bavard

Une menace plane

Le vendredi 29 Mars 2019, l'annonce suivante est publiée sur le site du SSTIC :

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 !

Voilà qui semble plutôt intéressant, je vais donc voir de quoi il en retourne. Avec ce message, l'archive qui contient le challenge est fournie. Cette archive contient quatre fichiers:

- flash.bin qui contient les données du système
- rom.bin qui contient le démarrage du téléphone
- README qui contient des instructions
- power_consumption.npz qui contient la trace de consommation mentionnée

Le README contient les informations suivantes :

Pour démarrer le téléphone dans un environnement virtuel vous devez utiliser qemu dans une version supérieure ou égale à 3.1. La ligne de commande suivante permet d'amorcer le système :

```
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
```

```
En ajoutant les options suivantes, vous pouvez obtenir un accès SSH dans
le téléphone virtuel (sur le port 5555 de votre localhost):
-netdev user,id=network0,hostfwd=tcp
:127.0.0.1:5555-192.168.200.200:22 -net nic,netdev=network0
```

Bonne chance

Au final, il y a assez peu d'informations supplémentaires. On note quand même que les concepteurs du challenge ont été assez gentils pour nous donner la ligne de commande exacte pour lancer la machine virtuelle. L'accès SSH sera aussi très utile par la suite pour récupérer les fichiers du système et les traiter en local¹.

Lorsque j'essaye de lancer une première fois la machine virtuelle, un message d'erreur indique que le 'keystore' n'existe pas, il est donc créé et on est supposé relancer le système. Au deuxième démarrage, le message change et le système demande une clé RSA pour déchiffrer le coffre à clés :

```
#####
#       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 peux maintenant faire le lien entre les différents éléments à disposition : pour démarrer, le système a besoin d'une clé AES, stockée dans le coffre à clés. Ce coffre à clé est stocké chiffré et nécessite une clé RSA pour être déchiffré. Malheureusement, lors de la saisie du téléphone, ce coffre à clés n'a pas pu être copié déchiffré, il faut donc d'une manière ou d'une autre retrouver l'exposant privé RSA utilisé. Pour ce faire, l'énoncé initial donne un indice : une trace de consommation a été capturée lors du démarrage du téléphone, donc lors du déchiffrement du coffre à clés. Tout ceci converge vers un type d'attaque bien connu : les attaques par canaux auxiliaires².

Le canal d'à côté

Sans rentrer dans des détails que je ne maîtrise pas, une attaque par canal auxiliaire repose sur une hypothèse simple : lorsqu'un processeur exécute un calcul, il génère des observables physiques qui dépendent du calcul effectué. Ces observables peuvent être de plusieurs sortes: consommation électrique, émanations électromagnétiques, émanations acoustiques, temps d'exécution, ...

Dans le cas d'une opération cryptographique, (chiffrement, déchiffrement, signature, ...), le calcul effectué dépend en plus d'une donnée secrète (la clé) qui est manipulée par le processeur. Par conséquent, lorsqu'il exécute un calcul cryptographique, le processeur génère des observables physiques qui dépendent du secret manipulé. L'objectif pour un attaquant est de mesurer un ou plusieurs de ces observables pour pouvoir retrouver le secret utilisé.

Pour voir à quoi j'ai affaire, je regarde donc cette trace de consommation. Le fichier qui contient la trace est une archive Zip. Une fois extraite, je me retrouve avec un fichier **arr_0.npy**. Une rapide recherche sur

1. Peu après la publication du challenge, un erratum a été publié : La version de qemu nécessaire est la 3.1.0, les versions supérieures ne fonctionnent pas.

2. En anglais, on verra utilisé le terme de "side-channel" pour désigner ces attaques.

Qwant confirme qu'il s'agit d'un format utilisé par la bibliothèque numpy³ pour stocker des tableaux de données. Après l'avoir chargée, je peux confirmer que ce fichier contient un unique tableau de 1 842 128 éléments : Pour retrouver l'exposant privé RSA, j'ai à ma disposition une unique trace de presque deux millions d'échantillons.

Premier réflexe face à un signal inconnu : je l'affiche pour voir à quoi il ressemble. Pour ce faire, j'utilise la bibliothèque Python matplotlib⁴, qui a le bon goût d'interagir correctement avec les tableaux numpy. A peu près au milieu de la trace, je peux voir ce qui ressemble de loin à un gros rectangle. La génération de tous les graphiques (non annotés) utilisés pour la résolution de cette étape ainsi que la résolution en elle-même tient en un seul script Python, disponible en annexe.

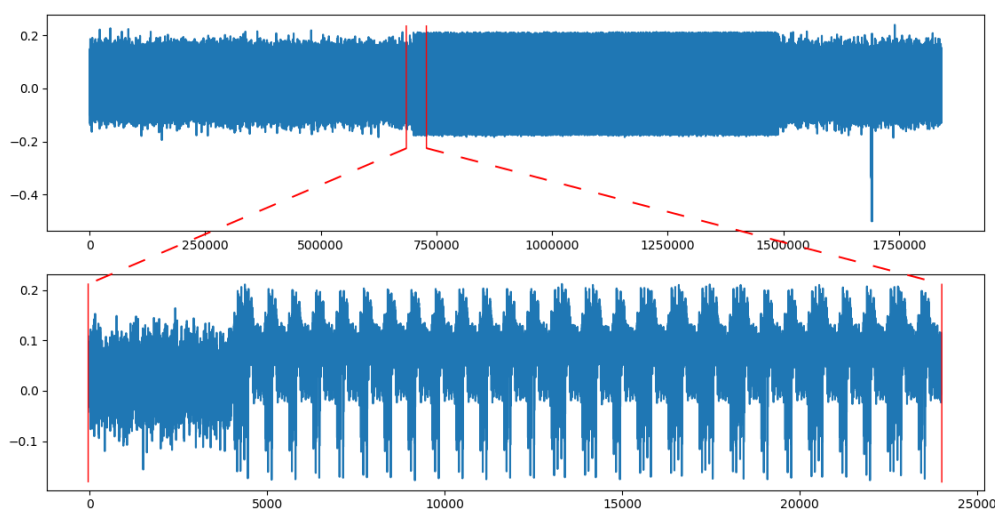


FIGURE 1 – Trace de consommation

Sur la figure 1 deux courbes sont visibles: d'abord la trace entière, puis le début du 'gros rectangle'. Là, je constate qu'il est en réalité composé d'un même motif qui à l'air de se répéter. En me déplaçant dans la trace, je peux confirmer que le motif qui apparaît se répète jusqu'à la fin du rectangle.

J'ai pu assister à quelques cours/présentations sur le sujet des attaques par canaux auxiliaires et l'un des exemple qui revenait systématiquement concernait les attaques dites 'SPA' pour 'Simple Power Analysis'. Lorsque l'implémentation cryptographique est vulnérable à une attaque SPA, j'ai régulièrement entendu dire qu'il était possible de 'lire' la clé utilisée à partir d'une unique trace de consommation. Je pose donc l'hypothèse suivante :

Hyp : L'implémentation de l'algorithme de déchiffrement RSA utilisé est vulnérable à une attaque de type SPA

Une opération de déchiffrement RSA se fait avec une exponentiation modulaire du message chiffré par l'exposant privé. Pour un message chiffré c , le message clair m est calculé par $m = c^d \pmod{N}$ où d est l'exposant privé (la clé secrète) RSA et N est le module RSA. Là encore, l'un des algorithmes couramment présenté pour réaliser cette opération est l'algorithme appelé 'square-and-multiply'. Cet algorithme parcourt tous les bits de d et réalise une ou deux opération suivant la valeur de ce bit. Le pseudo-code de cet algorithme est le suivant :

3. <https://www.numpy.org/>

4. <https://matplotlib.org/>

```
function decrypt(c, d, N) {
    m := c
    for b in d {
        m := m * m % N
        if b == 1 {
            m := (m * c) % N
        }
    }
}
```

Lorsque cet algorithme est exécuté, il va faire autant de tours de boucle qu'il y a de bits dans d . Lorsque le bit courant vaut 0, une multiplication est effectuée ; lorsque le bit courant vaut 1, deux multiplications sont exécutées. Par conséquent, si j'arrive à calculer avec précision le temps d'exécution de chaque tour de boucle fait au cours de l'exponentiation, alors je pourrai savoir quel tour manipule un 0 et quel tour manipule un 1. Un élément important est que pour cette attaque spécifique, la connaissance du module RSA n'est pas nécessaire. Cela tombe bien, puisqu'il est inconnu.

Dans le contexte de l'épreuve, je remarque que les motifs qui semblent répétés dans la trace se ressemblent beaucoup, mais que certains sont plus longs que d'autres⁵. Étant donné que je cherche un exposant privé RSA dans cette trace, je fais l'hypothèse suivante :

Hyp : Dans la trace de consommation, le 'gros rectangle' correspond à une exécution de l'algorithme 'square-and-multiply' et chacun des motifs correspond à un tour de boucle.

Un peu de lecture

Armés de mes deux hypothèses, je peux réduire le problème posé et le résoudre en trois étapes :

1. Identifier individuellement chacun des tours de boucle de l'algorithme de la trace
2. Calculer le temps d'exécution de chaque tour de boucle
3. Basé sur ce temps d'exécution, décider s'il s'agit d'un 1 ou d'un 0

Tout d'abord, plutôt que de traiter toute la trace, je réduis le signal à la partie qui correspond à l'exécution de l'exponentiation modulaire. Matplotlib permet de zoomer et dézoomer dans un signal, je retrouve donc que cet algorithme commence à l'échantillon 700083 et termine à l'échantillon 1488826.

En observant le signal sur la figure 2 il est possible de visuellement discriminer les différents motifs. De plus, je constate que chaque tour de boucle se compose d'un pic (materialisé par une flèche orange) pendant lequel la consommation varie beaucoup et d'un creux (matérialisé par une flèche noire) pendant lequel la consommation varie peu. Je remarque aussi que si temps d'exécution d'un creux a l'air à peu près constant, le temps d'exécution d'un pic varie.

Même si l'intuition me souffle qu'il est possible, dans ce cas précis, de 'lire' les bits de l'exposant privé à partir de la trace en zoomant suffisamment, je préfère utiliser une approche automatisée pour les retrouver. La méthode choisie consiste à éliminer le plus de perturbations du signal afin, à terme, de retrouver le temps d'exécution (en nombre d'échantillons) de chaque tour de boucle.

Pour commencer, je superpose un second signal au premier. Ce second signal encode chaque échantillon en une valeur 'haute' si l'échantillon considéré se trouve dans un pic d'un tour de boucle et en une valeur 'basse' si l'échantillon considéré se trouve dans le creux d'un tour de boucle. L'algorithme utilisé se contente de détecter les creux en vérifiant que les échantillons traités sont inférieurs à un seuil

5. Aucune information n'est donnée sur le taux d'échantillonnage du signal, donc il n'est pas possible de retrouver le temps d'exécution réel. Néanmoins le taux d'échantillonnage est constant, donc on peut calculer ce temps d'exécution de manière indirecte en comptant le nombre d'échantillons.

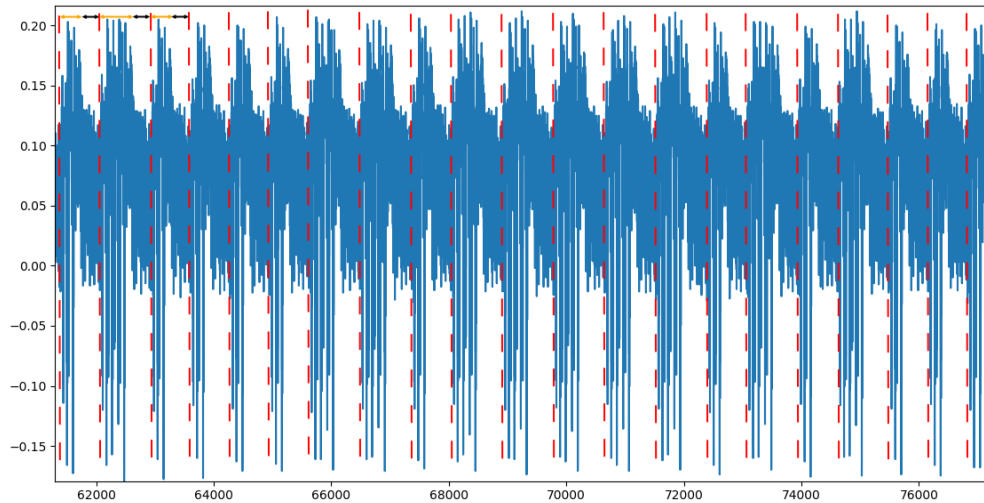


FIGURE 2 – Zoom sur la trace de consommation

pendant suffisamment longtemps⁶. Si l'échantillon ne se trouve pas dans un creux, alors cela signifie qu'il fait partie d'un pic.

Sur la figure 3 on peut voir les deux signaux superposés. D'une part chacun des tours de boucle a clairement été identifié au sein de la trace puisque chaque itération se compose d'un pic puis d'un creux. D'autre part, il est plus simple de calculer le temps d'exécution d'un tour de boucle à partir du signal en rouge que du signal en bleu car les transitions sont plus nettes.

Pour chaque tour de boucle, la partie basse a été laissée de côté car son temps d'exécution semble constant. La variation a l'air en effet porté par la partie haute. À partir du nouveau signal, je peux facilement compter le nombre d'échantillons qui composent chaque pic. En effectuant ce calcul, je constate qu'il y a 1024 pics. Comme chaque pic correspond à un bit de clé, ceci nous indique que l'exposant privé RSA fait 1024 bits, ce qui est une taille standard (même si trop faible) pour un exposant privé. Cette valeur est plutôt rassurante, puisque cela tend à confirmer à la fois mes hypothèses et la méthode de résolution.

Maintenant, il ne reste plus qu'à discriminer les bits à 0 et les bits à 1. Suivant mes hypothèses, un bit à 1 sera caractérisé par un temps d'exécution 'plutôt long' tandis qu'un bit à 0 sera caractérisé par un temps d'exécution 'plutôt court'. Ce qui m'intéresse ici n'est pas que le temps d'exécution soit absolument long ou court, mais que si l'on me donne deux temps d'exécution de deux pics distincts, je puisse décider s'ils dénotent le même bit ou un bit différent chacun.

Dans l'idéal, je souhaiterais que le temps d'exécution soit réparti de manière distincte entre ces deux classes (long/court) afin de simplifier la récupération de la clé secrète. Je vérifie cette répartition en affichant l'histogramme du nombre de pics, répartis suivant le nombre d'échantillon qui les compose.

L'histogramme sur la figure 4 montre qu'effectivement, il y a deux classes parfaitement distinctes. Il est possible d'utiliser la médiane des deux classes (la ligne rouge) pour discriminer les bits :

- Si le nombre d'échantillon dans un pic est supérieur à la médiane, alors le bit manipulé par l'algorithme 'square-and-multiply' lors de ce tour de boucle est un 1.
- Si le nombre d'échantillon dans un pic est inférieur à la médiane, alors le bit manipulé par l'algorithme 'square-and-multiply' lors de ce tour de boucle est un 0.

6. Les constantes précises utilisés sont : 0.145 pour le seuil ; 250 pour la longueur de la série d'échantillons. Si plus de 250 échantillons successifs sont inférieurs au seuil, alors ces échantillons appartiennent à un creux. Ces constantes ont été déterminées visuellement avec matplotlib.

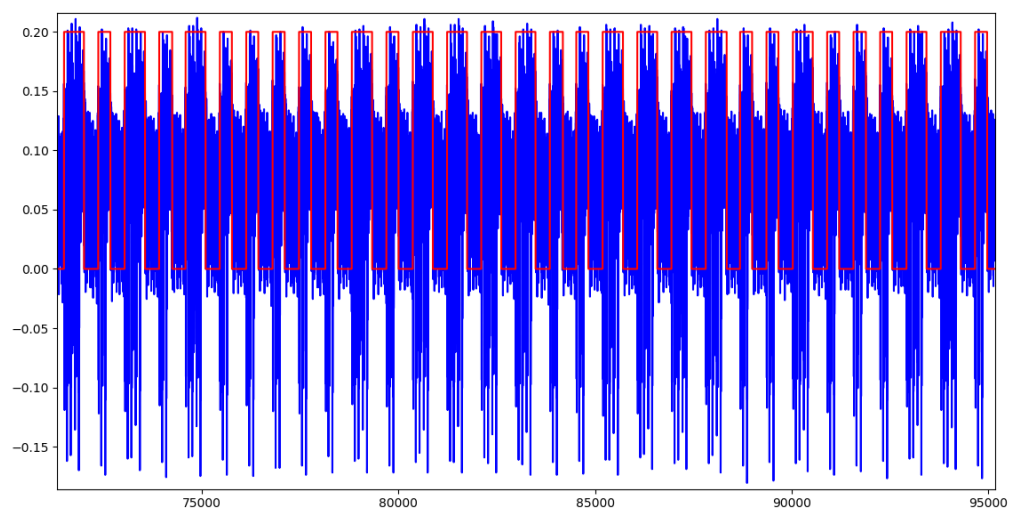


FIGURE 3 – Superposition de la trace originale et du signal de travail

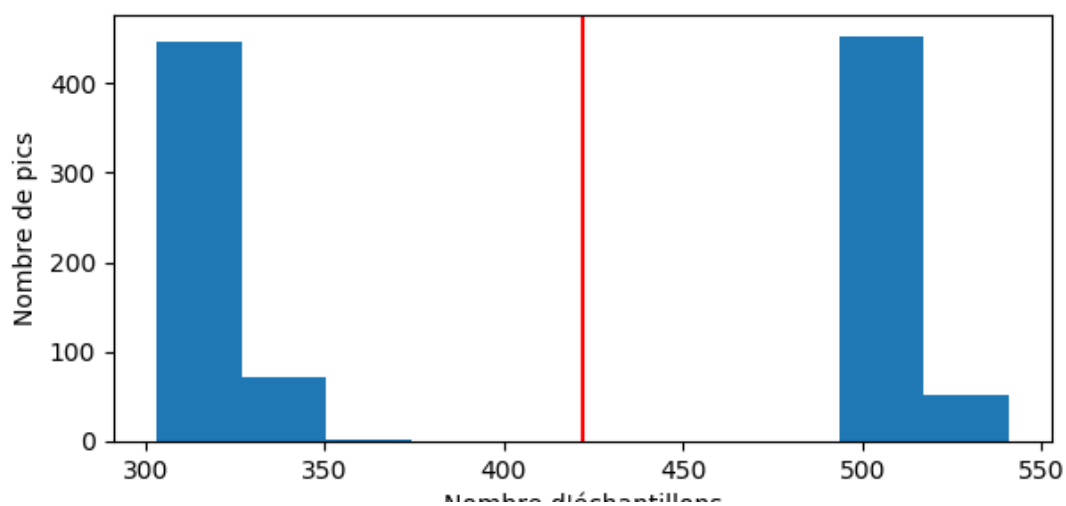


FIGURE 4 – Histogramme du nombre de pics suivant le nombre d'échantillons

Une fois que chacun des bits de clé a été retrouvé, il est possible de la reconstruire. Ne connaissant pas le sens de lecture choisi par le programme de démarrage, j'ai essayé de la lire de gauche à droite et de droite à gauche. Il se trouve que le 'bon' sens est ce dernier. En hexadécimal, la valeur de l'exposant privé est 0x23d87cdf97bb95abe6273c384190c765f552ab86f6de30a8db74435c95e6e3138f54af689812d8f9359cf0f4d453a0c11ec68ce470216c09e74c8947adaf23e902415d61ddf2c0ffe459cbb40f7de42bdb7cd14093100a570e8c29819765e2d8d276f86471b52ac29aa2ce2bb72cd45006279e82bec253ae9675fe45824f6001.

En rentrant cette valeur lors du démarrage du système, le coffre à clé est déchiffré et le système s'amorce avec le message suivant :

```
Bravo , envoyez le flag SSTIC{a947d6980ccf7b87cb8d7c246} à l'adresse  
challenge2019@sstic.org pour valider votre avancée
```

Cette étape est donc terminée, je peux passer à la suite.

C'est pourtant logique

Un système spartiate

Lorsque qemu lance le système, une fois la procédure d'amorçage terminée, les messages suivants apparaissent juste avant de me donner la main sur une invite de commande :

```
#####
#   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
Starting logging: OK
Initializing random number generator... done.
Starting network: OK
Starting dropbear sshd: OK
```

Les identifiants de l'accès SSH sont root:sstic, ce qui est simple à mémoriser. Au démarrage, le système essaye de déchiffrer des coffre-forts à l'aide du coffre à clé, mais n'y arrive pas parce les clés n'ont pas encore été trouvées. L'outil qui permet d'ajouter une clé au coffre est donné, il s'agit de `/root/tools/add_key.py`. La numérotation des coffre-forts a l'air d'indiquer que la progression dans le challenge est linéaire : il faut trouver une clé pour déchiffrer le premier, qui contiendra un challenge pour trouver la clé permettant de déchiffrer le deuxième, qui contiendra un challenge pour trouver la clé permettant de déchiffrer le troisième.

En regardant les binaires accessibles, je constate que l'outil gdb⁷ est présent dans le système, ce qui pourra être utile par la suite. Je remarque aussi la présence d'un module noyau au nom évocateur : `/lib/sstic.ko`, qui servira probablement au cours du challenge.

En navigant sous le répertoire `/root`, voici les fichiers à disposition:

```
# ls -lia .
total 728
 111 drwx-----  6 root root      160 Mar 28 22:24 .
   2 drwxr-xr-x 17 root root      420 May  5 09:00 ..
 191 -rwxr-xr-x  1 root root     7734 Mar 28 22:24 get_safe1_key.py
 194 drwxr-xr-x  2 root root       60 Mar 28 22:24 safe_01
 196 drwxr-xr-x  2 root root       60 Mar 28 22:24 safe_02
 192 drwxr-xr-x  2 root root       60 Mar 28 22:24 safe_03
 190 -rw-r--r--  1 root root    734017 Mar 28 22:24 schematics.png
 198 drwxr-xr-x  2 root root      120 May  5 08:49 tools
# ls -lia tools
```

7. <https://www.gnu.org/software/gdb/>

```
total 16
 198 drwxr-xr-x   2 root root    120 May  5 08:49 .
 111 drwx-----   6 root root    160 Mar 28 22:24 ..
 200 -rwxr-xr-x   1 root root   1511 Mar 28 22:24 add_key.py
 201 -rwxr-xr-x   1 root root    119 Mar 28 22:24 decrypt_containers.
    py
 199 -rwxr-xr-x   1 root root   1809 Mar 28 22:24 keystore.py
1986 -rw-r--r--   1 root root   2507 May  5 08:49 keystore.pyc
```

Les outils présents semblent utilisés pour rajouter les clés trouvées au fur et à mesure ainsi que déchiffrer les coffres-forts. Chaque coffre-fort contient un unique fichier `.encrypted_file` qui est illisible.

Le contenu du répertoire donne un gros indice sur l'étape suivante : le fichier Python `get_safe1_key.py` qui semble utilisé pour récupérer la clé du premier coffre.

Dans les couches basses

Ce fichier contient un petit nombre de fonctions :

- Quatre fonctions de débogage (`info`, `warn`, `error` et `ok`) qui permettent d'afficher des messages de différentes couleurs;
- Une fonction `secure_device` qui prend en entrée trois paramètres, mais qui n'est pas implémentée;
- Huit fonctions `step` numérotées `step1` à `step8` qui ne font que des appels à la fonction `secure_device`;
- Une fonction `init` qui ne fait que des appels à la fonction `secure_device`;
- Une fonction `main` qui exécute le programme.

La fonction `main` est la suivante :

```
def main():
    info("Dechiffrement du conteneur\n")

    info("Initialisation du secure element")

    info("Merci d'appuyer les 4 boutons du secure element puis appuyer sur entrée")
    input()
    if init() != 0xa1:
        error("Mauvaise initialisation, vérifiez l'état du sécure élement")
        sys.exit(0)
    else:
        ok("Initialisation, check 1 OK")

    info("Merci de relâcher les 4 boutons du secure element puis appuyer sur entrée")
    input()
    if init() != 0xe0:
        error("Mauvaise initialisation, vérifiez l'état du sécure élement")
        sys.exit(0)
    else:
        ok("Initialisation, check 2 OK")
```

```

    warn("Calcul de la clef de déchiffrement du conteneur, préparez vous
à appuyer sur les boutons")

    info("Step 1, appuyez sur entrée quand la combinaison de boutons est
enfoncee")
    input()
    s1 = step1()
    info("Step 2, appuyez sur entrée quand la combinaison de boutons est
enfoncee")
    input()
    s2 = step2()
    info("Step 3, appuyez sur entrée quand la combinaison de boutons est
enfoncee")
    input()
    s3 = step3()
    info("Step 4, appuyez sur entrée quand la combinaison de boutons est
enfoncee")
    input()
    s4 = step4()
    info("Step 5, appuyez sur entrée quand la combinaison de boutons est
enfoncee")
    input()
    s5 = step5()
    info("Step 6, appuyez sur entrée quand la combinaison de boutons est
enfoncee")
    input()
    s6 = step6()
    info("Step 7, appuyez sur entrée quand la combinaison de boutons est
enfoncee")
    input()
    s7 = step7()
    info("Step 8, appuyez sur entrée quand la combinaison de boutons est
enfoncee")
    input()
    s8 = step8()

    key = bytearray([s1,s2,s3,s4,s5,s6,s7,s8])
    h = hashlib.sha256(key).hexdigest()

    if "00c8bb35d44dcbb2712a11799d8e1316045d64404f337f4ff653c27607f436ea
"
        == h:
            ok("Hash ok")
            info("Dérivation de la clef AES safe_01")
            aes_key = hashlib.scrypt(key,salt =b"sup3r_s3cur3_k3y_d3r1v4t10n
_s4lt",n=1<<0xd,r=1<<3,p=1<<1,dklen=32)
            info("aes key : %s" % aes_key.hex())
            info("Vous pouvez sauvegarder cette clef en utilisant
/root/tools/add_key.py key")
        else:
            error("Mauvais hash, déchiffrement impossible")

```

Les messages sont plutôt explicites et je comprends que la fonction `secure_device` est censée effectuer la communication avec un **secure element**. Après l'initialisation, à chaque étape l'utilisateur est supposé rentrer une combinaison secrète en appuyant sur des boutons qui se trouvent a priori sur le **secure element**. À chaque étape, le programme dérive une valeur à partir de cette combinaison.

Une fois que huit valeurs ont été générées, le programme forme une clé qui est hachée par la fonction SHA-256⁸ puis comparée à une valeur statique. Si les valeurs correspondent, il utilise la fonction `scrypt`⁹ avec le sel `sup3r_s3cur3_k3y_d3r1v4t10n_s41t` pour dériver la clé qui servira à déchiffrer le premier coffre-fort.

La fonction SHA-256 n'a pas de vulnérabilité connue en pré-image, c'est à dire qu'à partir d'une image par cette fonction, on ne possède aucune information sur son antécédent. Il n'est donc pas possible d'inverser la fonction en partant de la valeur statique présente pour retrouver la clé initiale.

La fonction `secure_device` laisse de nouveaux indices, qui permettent d'avancer :

```
def secure_device(a,b,op):
    out = 0
    """
    TODO :
        - Implémentation de la communication avec le secure element
          * Entrées A et B (AO = 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
```

À la lecture du TODO, je comprends qu'il faut chercher les fameux documents de conception. Ils permettront sans doute de donner des informations sur le **secure element**. Il y a un second fichier qui semble intéressant dans le répertoire `/root`: l'image `schematics.png`. 'schematics' signifie 'schéma', ce qui correspond tout à fait au nom que l'on pourrait donner à un document de conception.

Le schéma de conception (annoté) en figure 5 montre un circuit logique simple. La documentation de la fonction `secure_device` donne en plus des informations sur les entrées et sorties du **secure element** :

- Il a quatre entrées A, B, OP et les boutons utilisés pour rentrer une combinaison;
- Il sort une valeur Out.

Sur le schéma, je retrouve les entrées A, B et OP bit à bit ainsi que la sortie Out bit à bit. Par élimination, les entrées B1, B2, B3 et B4 doivent correspondre aux boutons sur lesquels doit appuyer l'utilisateur. Afin d'éviter toute confusion avec la notation des bits de B (numérotés de B0 à B7), les entrées correspondants aux boutons seront notées BUT1, BUT2, BUT3 et BUT4.

J'en conclus que j'ai affaire au schéma de conception interne du **secure element**. Je peux donc réimplémenter les opérations qu'il exécute et ainsi implémenter la fonction `secure_device`. Le circuit est séparé en quatre parties, identifiées sur la figure 5 par quatre couleurs différentes.

- Le bloc rouge prend en entrée A (8 bits) et BUT3 (1 bit), il sort une valeur intermédiaire AI (8 bits).

8. <https://fr.wikipedia.org/wiki/SHA-2>

9. <https://fr.wikipedia.org/wiki/Scrypt>

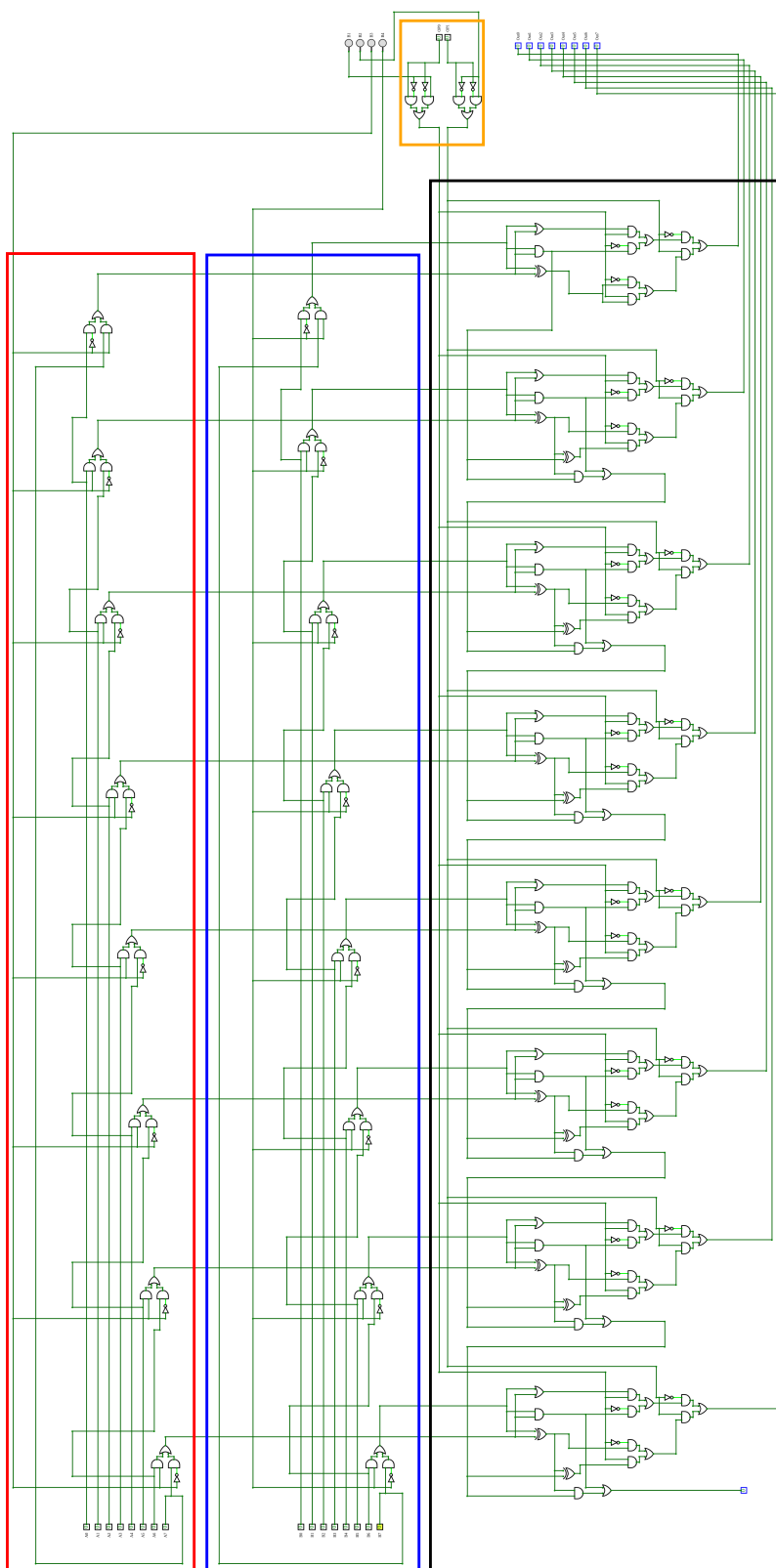


FIGURE 5 – Schéma de conception

- Le bloc bleu prend en entrée B (8 bits) et BUT4 (1 bit), il sort une valeur intermédiaire BI (8 bits).
- Le bloc orange prend en entrée OP (2 bits), BUT1 (1 bit) et BUT2 (1 bit), il sort une valeur intermédiaire OPI (2 bits).
- Le bloc noir prend en entrée AI (8 bits), BI (8 bits) et OPI (2 bits), il sort la valeur finale Out (8 bits) et une retenue (1 bit, non utilisée).

Type de porte	Symbole	Notation
Porte NON	$x \longrightarrow \neg x$	$z = \bar{x}$
Porte XOR	$x, y \longrightarrow x \oplus y$	$z = x \oplus y$
Porte ET	$x, y \longrightarrow x * y$	$z = x * y$
Porte OU	$x, y \longrightarrow x + y$	$z = x + y$

FIGURE 6 – Signification des portes logiques

La figure 6 montre les différentes portes logiques utilisées dans le schéma ainsi que leur signification.

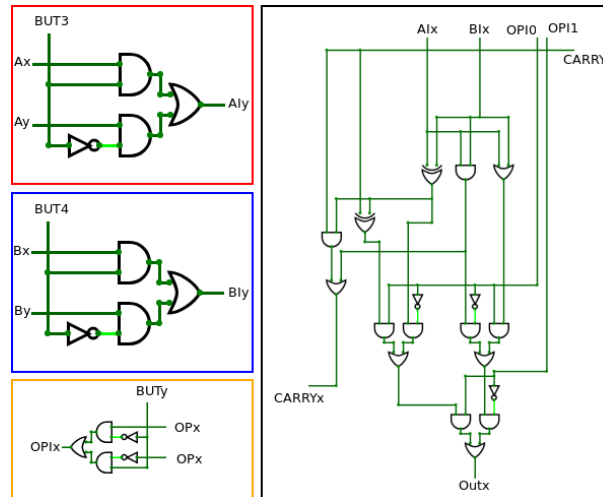


FIGURE 7 – Cellules logiques

Pour réimplémenter le schéma entier, je me concentre sur chacun des blocs séparément. Chaque bloc est constitué de quelques cellules, qui sont les petits amas de portes logiques que l'on voit sur le schéma. Les blocs rouge, bleu et noir sont constitués de huit cellules chacun, tandis que le bloc orange en comprend deux. Il se trouve qu'à l'intérieur de chaque bloc les cellules sont identiques, ce qui simplifie le travail de compréhension. Les différentes cellules sont présentes sur la figure 7. Il y a néanmoins une légère subtilité dans le bloc noir, puisque chaque cellule sort un bit de la sortie Out et un bit qui est propagé à la cellule suivante.

L'un des éléments qui revient souvent dans le circuit est celui représenté dans la figure 8. Il est utilisé par

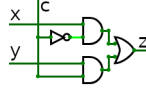


FIGURE 8 – Circuit logique implémentant la fonction choix

les trois principaux blocs. Cet élément prend 3 bits d'entrées que j'ai nommées x , y et c et sort un bit de sortie z . L'équation booléenne associée est

$$z = (y * c) + (x * \bar{c})$$

D'un point de vue plus algorithmique, il s'agit d'une fonction choix dont le pseudo-code est le suivant:

```
function choice(x, y, c) {
    if c {
        return y
    } else {
        return x
    }
}
```

En isolant chaque cellule et en utilisant les valeurs intermédiaires définies précédemment, les équations booléennes qui correspondent au schéma sont les suivantes :

$$AI7 = choice(A7, A6, BUT3)$$

$$AI6 = choice(A6, A5, BUT3)$$

$$AI5 = choice(A5, A4, BUT3)$$

$$AI4 = choice(A4, A3, BUT3)$$

$$AI3 = choice(A3, A2, BUT3)$$

$$AI2 = choice(A2, A1, BUT3)$$

$$AI1 = choice(A1, A0, BUT3)$$

$$AI0 = choice(A0, A7, BUT3)$$

$$BI7 = choice(B7, B6, BUT4)$$

$$BI6 = choice(B6, B5, BUT4)$$

$$BI5 = choice(B5, B4, BUT4)$$

$$BI4 = choice(B4, B3, BUT4)$$

$$BI3 = choice(B3, B2, BUT4)$$

$$BI2 = choice(B2, B1, BUT4)$$

$$BI1 = choice(B1, B0, BUT4)$$

$$BI0 = choice(B0, B7, BUT4)$$

$$OPI0 = OPI0 \oplus BUT1$$

$$OPI1 = OPI1 \oplus BUT2$$

$$Out0 = choice(choice(AI0 * BI0, AI0 + BI0, OPI0), choice(AI0 \oplus BI0, AI0 \oplus BI0, OPI0), OPI1)$$

$$\begin{aligned}
& carry0 = AI0 * BI0 \\
Out1 &= choice(choice(AI1*BI1, AI1+BI1, OPI0), choice(AI1\oplus BI1, AI1\oplus BI1\oplus carry0, OPI0), OPI1) \\
& carry1 = AI1 * BI1 + ((AI1 \oplus BI1) * carry0) \\
Out2 &= choice(choice(AI2*BI2, AI2+BI2, OPI0), choice(AI2\oplus BI2, AI2\oplus BI2\oplus carry1, OPI0), OPI1) \\
& carry2 = AI2 * BI2 + ((AI2 \oplus BI2) * carry1) \\
Out3 &= choice(choice(AI3*BI3, AI3+BI3, OPI0), choice(AI3\oplus BI3, AI3\oplus BI3\oplus carry2, OPI0), OPI1) \\
& carry3 = AI3 * BI3 + ((AI3 \oplus BI3) * carry2) \\
Out4 &= choice(choice(AI4*BI4, AI4+BI4, OPI0), choice(AI4\oplus BI4, AI4\oplus BI4\oplus carry2, OPI0), OPI1) \\
& carry4 = AI4 * BI4 + ((AI4 \oplus BI4) * carry3) \\
Out5 &= choice(choice(AI5*BI5, AI5+BI5, OPI0), choice(AI5\oplus BI5, AI5\oplus BI5\oplus carry2, OPI0), OPI1) \\
& carry5 = AI5 * BI5 + ((AI5 \oplus BI5) * carry4) \\
Out6 &= choice(choice(AI6*BI6, AI6+BI6, OPI0), choice(AI6\oplus BI6, AI6\oplus BI6\oplus carry2, OPI0), OPI1) \\
& carry6 = AI6 * BI6 + ((AI6 \oplus BI6) * carry5) \\
Out7 &= choice(choice(AI7*BI7, AI7+BI7, OPI0), choice(AI7\oplus BI7, AI7\oplus BI7\oplus carry2, OPI0), OPI1) \\
& carry7 = AI7 * BI7 + ((AI7 \oplus BI7) * carry6)
\end{aligned}$$

A partir de ces équations, l'implémentation de la fonction **secure_device** devient très simple. Néanmoins, il est possible de simplifier ces opérations. Les fonctions correspondant aux blocs rouge et bleu servent à effectuer une rotation d'un bit aux entrées A et B, suivant l'état d'un bouton. Le bloc orange sert à définir l'opération à effectuer à partir des deux bits OP0, OP1 et des valeurs des deux boutons BUT1 et BUT2. Il y a quatre opérations possibles suivant la valeur d'OPI :

- Si OPI vaut 0, alors $Out = AI \& BI$ (ET logique)
- Si OPI vaut 1, alors $Out = AI | BI$ (OU logique)
- Si OPI vaut 2, alors $Out = AI \oplus BI$ (OU exclusif)
- Si OPI vaut 3, alors $Out = AI + BI$ (addition modulo 2^8)

Le script complété se trouve en annexe. De plus, il est possible de vérifier l'implémentation avec la fonction d'initialisation. Lorsque aucun bouton n'est enfoncé, la sortie de la fonction **init** est supposée être 0xa1 et lorsque tous les boutons sont enfoncés la sortie de la fonction **init** est supposée être 0xe0.

Avec un peu de brutalité

Le circuit logique qui correspond au **secure element** a été complété, il s'agit maintenant de retrouver la clé. Il y a quatre boutons sur le **secure element**, c'est à dire 2^4 valeurs possibles pour une entrée de l'utilisateur. Je remarque aussi que cette entrée est la seule source de variation pour chaque fonction d'étape. Cela signifie que même si chaque fonction **step1**, **step2**, ... sort une valeur sur huit bits, cette sortie ne peut prendre en réalité que 2^4 valeurs possibles. Chaque fonction d'étape apporte donc 4 bits d'entropie à la clé à trouver.

Les sorties possibles pour chaque fonction sont les suivantes :

- **step1** peut générer les valeurs {0xd7, 0x19, 0xdf, 0x40, 0xaf, 0x62, 0xbf, 0x81, 0x47, 0x39, 0xc7, 0xc2, 0x90, 0x72, 0x8f, 0x89}
- **step2** peut générer les valeurs {0x29, 0xd1, 0x08, 0xdb, 0x52, 0xa4, 0x10, 0xb5, 0x32, 0xd9, 0x90, 0xda, 0xf3, 0xb4, 0x20, 0xf5}
- **step3** peut générer les valeurs {0xed, 0x8d, 0xd9, 0xdf, 0xdb, 0x1c, 0xb0, 0xbf, 0xbd, 0x4f, 0xdc, 0xdd, 0x7b, 0x9f, 0x01, 0xbb}
- **step4** peut générer les valeurs {0xaa, 0x38, 0xa9, 0x28, 0x55, 0x71, 0x53, 0x52, 0x1a, 0x4e, 0xad, 0x00, 0x35, 0x9c, 0x5b, 0x40}
- **step5** peut générer les valeurs {0xfd, 0x00, 0xaf, 0x47, 0xff, 0x01, 0x60, 0x8e, 0xed, 0x41, 0xde, 0xd4, 0xcb, 0x82, 0xbd, 0xa9}

- **step6** peut générer les valeurs {0x31, 0xb2, 0xf6, 0x64, 0x4e, 0x65, 0xed, 0xc8, 0x3d, 0xcf, 0xee, 0xe0, 0x1b, 0x9d, 0xdd, 0xc1}
- **step7** peut générer les valeurs {0xdd, 0x54, 0x4d, 0x4f, 0xbb, 0xa9, 0x9a, 0x9e, 0x60, 0x86, 0x05, 0xdf, 0xca, 0xe8, 0x0a, 0xbf}
- **step8** peut générer les valeurs {0x83, 0x54, 0x78, 0xfb, 0x07, 0xaa, 0xf0, 0xf3, 0x97, 0x85, 0xd8, 0xfc, 0x4d, 0x48, 0xb0, 0xfd}

La clé recherchée est composée des sorties des huit fonctions d'étape, elle a donc une entropie de $(2^4)^8$ soit 2^{32} . Comme la vérification de la bonne valeur se fait par la comparaison à une valeur dérivée avec SHA-256 (qui est une fonction de hachage rapide), il est possible de générer les 2^{32} valeurs de clés possibles et de vérifier laquelle est la bonne. Un calcul de 2^{32} opérations SHA-256, même sur mon ordinateur qui est agé et peu puissant est quelque chose de réalisable rapidement.

Comme j'ai l'habitude de commencer par des implémentations naïves, je laisse un script Python générer toutes les clés successivement et les tester une à une.

Je raffine peu à peu mon approche, avec des programmes en C puis en Go. J'utilise un programme C pour générer toutes les clés possibles et les stocker dans 16 fichiers distincts (de 2^{28} clés chacun sous format textuel, ce qui donne des fichiers de 4 Go). J'utilise ensuite un programme Go pour itérer à travers les clés de ces fichiers et tester si elles sont correctes. Le temps d'implémenter ces éléments et de générer les fichiers, mon programme Python initial a déjà parcouru 85% de l'espace des clés (en 4 heures) sans trouver la réponse et je mets sérieusement en doute ma méthode.

Cependant, en lançant le programme en Go depuis l'autre bout de l'espace des clés, il trouve la solution qui est 8fa4dfa9d4edbbf0. On peut alors dériver la clé qui sert à déchiffrer le premier coffre en utilisant la fonction `scrypt` avec les paramètres `salt="sup3r_s3cur3_k3y_d3r1v4t10n_s4lt"`, `N=8192`, `r=8`, `p=2`, `dkeylen=32`. La clé obtenue est : 5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a.

En ajoutant cette clé avec le script `/root/tools/add_key.py`, le conteneur est déchiffré avec le message suivant :

```
# python add_key.py 5
    fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9b
ad427a8a
[+] Key with key_id 00000002 ok
[+] Key added into keystore
[+] Envoyez le flag SSTIC{5
    fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00
cdee9bad427a8a}
à l'adresse challenge2019@sstic.org pour valider votre avancée
[+] Container /root/safe_01/.encrypted decrypted to
/root/safe_01/decrypted_file
```

J'ai le deuxième flag et la troisième épreuve semble débloquée. Le fichier contenu dans le coffre-fort est déchiffré et je vais voir ce qu'il en est.

Une histoire d'elfes et de nains

Pour ce challenge, j'utilise la connexion SSH à la machine virtuelle pour copier le fichier déchiffré sur mon ordinateur dans le but de l'analyser.

À une exception près

Premiers pas en rétro-ingénierie

En utilisant la commande `file`, je constate que le fichier déchiffré est un exécutable 64 bit au format ELF¹⁰ pour l'architecture AArch64. La commande `strings` permet de lister les chaînes de caractères dans un fichier. Tout de suite, quatre d'entre elles attirent l'oeil :

- "That's the correct flag :)"
- "Not good"
- "Usage : %s <flag>"
- "SSTIC{congolexicomatisation}"

En deux essais, je constate que lorsque je lance l'exécutable sans argument, il s'arrête en affichant "Usage ./decrypted_file <flag>". Lorsque je lance l'exécutable avec une entrée aléatoire, il s'arrête en affichant "Not good". De plus, en essayant de donner la valeur "SSTIC{congolexicomatisation}" en entrée, l'exécutable s'arrête en affichant "Not good". J'en conclus qu'il va falloir regarder les entrailles de ce binaire pour trouver la prochaine clé.

J'ai utilisé le logiciel Ghidra¹¹ pour effectuer l'ingénierie inverse du binaire. Cet outil dispose d'un décompilateur AArch64 intégré et est capable d'analyser un binaire inconnu pour y trouver les fonctions et symboles présents. Après avoir chargé le binaire, je vois que très peu de fonctions sont définies.

J'ai commencé par chercher les références aux chaînes de caractères déjà identifiées. La chaîne "Usage: %s <flag>" est utilisée par la fonction `FUN_00402e68` tandis que les trois autres ne sont pas référencées selon ghidra. En parcourant un peu le binaire, il se trouve que ghidra a manqué d'analyser la fonction qui se trouve à l'adresse `0x00400ec0`. En lui indiquant manuellement qu'elle existe, il finit par indiquer que les trois chaînes de caractères sont utilisées par cette nouvelle fonction. Selon le décompilateur, leur code est le suivant :

```
undefined8 FUN_00402e68(int iParam1,undefined8 *puParm2)
{
    if (iParam1 != 2) {
        printf("Usage : %s <flag>\n",*puParm2);
        /* WARNING: Subroutine does not return */
        exit(1);
    }
    /* try { // try from 00402eb4 to 00402eb7 has its
    CatchHandler @ 00402f20 */
    FUN_00402e34(puParm2[1]);
    return 0;
}

undefined8 FUN_00402ec0(undefined8 param_1,longlong param_2)
{
    int iVar1;
    uint uVar2;
    char *__s2;

    if (param_2 == 1) {
```

10. https://fr.wikipedia.org/wiki/Executable_and_Linkable_Format

11. Ghidra est un framework d'ingénierie inverse logicielle. <https://ghidra-sre.org/>

```

__s2 = (char *)__cxa_begin_catch(param_1);
iVar1 = strcmp("SSTIC{congolexicomatisation}",__s2);
DAT_004140b8 = PTR_s_Not_good_00414098;
if (iVar1 == 0) {
    DAT_004140b8 = PTR_s_That_s_the_correct_flag____00414090;
}
/* catch(type#1 @ 00413db8) { ... } // from try @
00402eb4 with catch @ 00402f20
*/
/* try { // try from 00402f3c to 00402f3f has its
CatchHandler @ 00402f48 */
uVar2 = puts(DAT_004140b8);
__cxa_end_catch((ulonglong)uVar2);
return 0;
}
/* WARNING: Subroutine does not return */
_Unwind_Resume(param_1);
}

```

Le première fonction semble prendre en paramètre argc, le nombre d'arguments passés au programme et argv, le tableau d'arguments passés au programme. Quant à la deuxième, elle semble prendre en paramètre une valeur de vérification. Je remarque aussi qu'il se passe quelque chose au niveau des exceptions mais sans comprendre quoi : ghidra indique que les blocs `catch` sont mal positionnés. Il y a enfin une valeur qui est récupérée, par un moyen encore inconnu et comparée à la chaîne "SSTIC{congolexicomatisation}". Si les deux sont identiques, le binaire va afficher "That's the correct flag :)" et si elles ne le sont pas il va afficher "Not good". Enfin, ghidra ne voit aucune référence à la seconde fonction : selon lui elle n'est jamais appelée.

À ce stade, je formule plusieurs hypothèse pour poursuivre mon analyse :

Hyp : L'exécutable n'implémente pas de mécanisme d'anti-débogage.

Hyp : La chaîne de caractère passée en entrée est modifiée par un procédé encore inconnu, puis la seconde fonction est appelée par un procédé encore inconnu.

En m'appuyant sur la première hypothèse, je lance l'outil de débogage gdb pour faire une analyse dynamique de l'exécutable. Je place un point d'arrêt à l'adresse 0x00400ec0, celle de la seconde fonction. Après plusieurs essais avec des chaînes de caractères de longueur différente, je me rends à l'évidence : cette fonction n'est pas appelée directement. J'essaie de mettre des points d'arrêt à divers endroits dans la fonction et je finis par trouver une adresse qui fonctionne : 0x00402f20.

```

0x402f20:    mov x1, x28
0x402f24:    adrp    x0, 0x414000
0x402f28:    add x0, x0, #0xb8
0x402f2c:    str x1, [x0]
0x402f30:    adrp    x0, 0x414000
0x402f34:    add x0, x0, #0xb8
0x402f38:    ldr x0, [x0]
0x402f3c:    bl 0x402c30 <puts@plt>

```

Le code à cet endroit est assez direct : La chaîne de caractères contenue dans le registre x28 est affichée à l'écran avec la fonction `puts`. C'est ce registre qui contient une adresse vers la chaîne "Not good" qui s'affiche à l'écran.

Comme je n'arrive pas à observer la comparaison qui a lieu en utilisant gdb, je me demande si l'exécutable n'implémente pas en fin de compte des fonctionnalités d'anti-débogage, afin d'empêcher justement ce type d'analyse. J'opte alors pour une autre approche à l'analyse dynamique : `LD_PRELOAD`.

La variable d'environnement `LD_PRELOAD` permet de pré-charger une ou plusieurs bibliothèques partagées avant l'exécution d'un binaire. Il est possible par ce biais de modifier certaines fonctions présentes dans les bibliothèques légitimes. Dans mon cas, je souhaite modifier les fonctions de la `libc` par mon propre code afin de voir quels appels sont effectués et avec quels arguments. L'objectif étant encore de pouvoir observer la comparaison entre la chaîne en argument dérivée et la chaîne "SSTIC{congolexicomatisation}".

Je réimplémente d'abord la fonction `strcmp` seule, puis les fonctions `printf` et `puts`, puis les fonctions `memset` et `memcpy`. Après avoir compilé ce code pour AArch64 et l'avoir transféré sur le système du téléphone, j'exécute le binaire en pré-chargeant ma bibliothèque avec la commande `LD_PRELOAD=/tmp/preload.so /root/safe_01/decrypted_file AAAA`. Cela fonctionne : c'est mon implémentation de ces fonctions qui est appelée. J'observe des comparaisons de chaînes de caractères qui sont effectués par les fonctions de résolution dynamique de symboles, mais aucune qui n'est faite par le programme. Les fonctions additionnelles implémentées ne m'éclairent pas non plus sur le comportement du programme.

Une fois ces résultats négatifs obtenus, je considère que ma seconde hypothèse était fausse et je cherche d'autres méthodes pour avoir des informations sur ce que fait le programme.

Pendant ce temps là

Pendant mes tests où j'essayais de faire une analyse dynamique du programme, je l'ai lancé avec plusieurs arguments différents (dont je ne me souviens plus). Néanmoins, j'ai eu l'impression qu'avec certains arguments, le programme mettait plus longtemps à s'exécuter qu'avec d'autres. Je n'y avais pas prêté attention sur le moment¹², mais là je cherche à récolter le plus d'informations possibles et le canal auxiliaire qu'est le temps d'exécution du programme me semble une piste intéressante.

Je cherche donc à évaluer l'hypothèse suivante :

Hyp : Selon l'argument qui lui est passé, le programme met un temps sensiblement différent à s'exécuter.

En reprenant mes entrées précédentes, je constate que la variation de temps d'exécution que j'avais repéré est bien présente et qu'elle a l'air de dépendre de la longueur de l'argument plutôt que de son contenu. J'écris un script Python qui va tester plusieurs fois toutes les longueurs d'argument possible, de 1 à 256 caractères et afficher le temps moyen (sur 50 essais) d'exécution pour une longueur donnée. La résolution de l'horloge (fonction `time_ns` du package `time`) utilisée est de l'ordre de la nanoseconde, cependant chaque essai lance un nouveau processus, ce qui rajoute nécessairement du bruit de mesure.

La figure 9 montre les résultats de ces mesures. Quatre valeurs se distinguent nettement du reste : Lorsque l'argument fait 7, 14, 22 ou 32 caractères de long, le binaire met cinq fois plus longtemps à s'exécuter. J'en déduis qu'à un moment, il y a une vérification de longueur qui est faite et que donner un argument d'une de ces longueur-ci passe ce test.

J'exécute plusieurs fois ces mesures, avec des arguments différents afin de voir si ces résultats sont constants : ils le sont (pour le moment). Quelques heures plus tard, en reprenant les tests, je constate que deux des pics ont disparus. Lorsque le binaire est lancé avec un argument de longueur 7 ou 14 caractères, je n'observe plus un temps d'exécution anormalement élevé, même sur des tests déjà passés. Seuls les pics à 22 et 32 caractères respectivement demeurent. À ce jour je n'ai pas d'explication quant à l'observation de ces pics en premier lieu.

12. Initialement, j'avais considéré que mon PC, réticent par moments, était la source de cette variation.

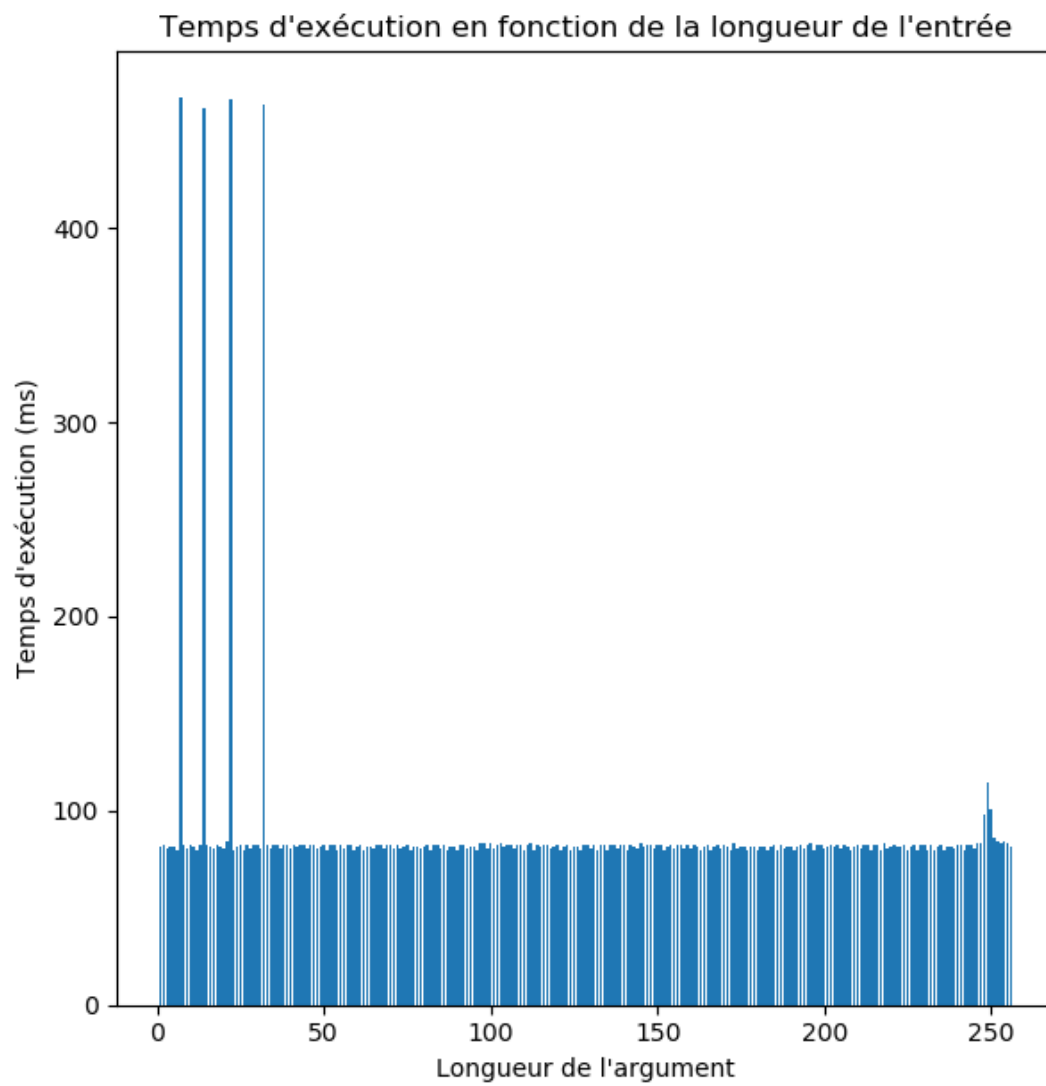


FIGURE 9 – Temps d'exécution

Je repasse les tests précédents (débogage, LD_PRELOAD) en prenant soin de fournir des arguments de longueur 22 ou 32 caractères, mais j'observe les mêmes résultats négatifs que précédemment. J'en conclus donc que suivant la taille de l'argument en paramètre, l'exécution du programme peut être 'lente' ou 'rapide'. Par la suite, je parlerai de programme 'lent' (resp. 'rapide') ou d'exécution 'lente' (resp. 'rapide').

Je continue aussi la rétro-ingénierie en parallèle de ces tests. Cette fois-ci je me concentre sur ce que devient l'argument donné au programme, lorsqu'il est traité par la fonction à l'adresse 0x00402e34. Il y a très peu d'instructions et je reconstruis le programme suivant :

```
void 402e34(char* argv1)
{
    throw argv1;
}

int main(int argc, char** argv)
{
    if (argc != 1) {
        printf("Usage: %s <flag>\n", argv[0]);
    }
    402e34(argv[1]);
}
```

J'ai toujours très peu d'informations sur ce que fait le binaire, mais cela semble impliquer les exceptions. A première vue, il semble que le binaire passe du temps à l'intérieur du mécanisme de gestion d'exceptions et ghidra reconnaît aussi qu'il y a des problèmes dans le positionnement des blocs `catch`. Il faut que j'en apprenne plus sur la gestion d'exceptions en C++.

Une question de spécification

Le mécanisme de gestion d'exceptions qui est utilisé ici se base sur l'ABI (Application Bridge Interface) Itanium, chapitre 4¹³. Ce document spécifie les interfaces et les structures qui sont impliquées dans la gestion d'exceptions. Le compilateur gcc utilise cette interface pour générer le code lié aux exceptions, tandis que l'environnement d'exécution spécifié dans la norme est partagé entre les bibliothèques libstdc++ et libgcc.

Le système d'exploitation Windows utilise son propre mécanisme de gestion d'exceptions, nommé *Structured Exception Handling* (SEH) qui ne sera pas abordé ici.

Les deux principales ressources qui m'ont permises d'appréhender le sujet sont :

- La spécification en elle-même
- La série de 21 articles écrits par Nico Brailovsky¹⁴

J'ai aussi été voir les implémentations des deux bibliothèques libstdc++ et libgcc, plus quelques autres éléments (Stackoverflow, ...) sur certains points spécifiques.

Dans le mécanisme de gestion des exceptions, certaines informations doivent être fournies à l'environnement d'exécution pour que tout fonctionne. Par exemple, il faut lui fournir l'emplacement des différents blocs `catch`, ainsi que les informations de type attendus par chacun des blocs (dans le cas où le code à exécuter est différent selon le type de l'exception attrapée). Ces informations sont présentes dans certaines sections des fichiers ELF, selon le format DWARF¹⁵.

Assez succinctement, le mécanisme de gestion d'exceptions débute de la manière suivante :

13. <https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html>

14. La version intégrale se trouve ici : <https://monoinfinito.wordpress.com/series/exception-handling-in-c/>

15. <http://dwarfstd.org/>

1. Le programme appelle la fonction `__cxa_allocate_exception` (libstdc++) et récupère un pointeur vers une structure `__cxa_exception`
2. Le programme stocke un pointeur vers l'objet lancé dans la structure `__cxa_exception`
3. Le programme appelle la fonction `__cxa_throw` (libstdc++) avec la structure `__cxa_exception` ainsi que le type de l'objet lancé en arguments.
4. La fonction `cxa_throw` initialise une structure `Unwind_Context` et appelle la fonction `Unwind_RaiseException` (libgcc) avec ce contexte.
5. La fonction `Unwind_RaiseException` va dérouler la pile (*stack unwinding* en anglais) et appeler la 'routine de personnalité' (*personality routine*) avec chaque trame. La routine de personnalité fait partie du programme, elle est incluse par le compilateur. Ici, il s'agit de la fonction `__gxx_personality_v0`
6. Pour chaque trame, la routine de personnalité va :
 - Récupérer le pointeur d'instruction qui a lancé l'exception via la fonction `_Unwind_GetIP` (libgcc)
 - Récupérer l'adresse de la fonction qui a lancé l'exception via la fonction `_Unwind_GetRegionStart` (libgcc)
 - Chercher un bloc `catch` qui soit capable de gérer cette exception. Ces informations sont contenues dans la section `.gcc_except_table`.
 - Si un bloc est trouvé, alors la routine retourne le code de retour adéquat et le déroulement de la pile s'arrête. Sinon, il continue.
7. La fonction `Unwind_RaiseException` va dérouler la pile **une seconde fois** et appeler (à nouveau) la routine de personnalité avec chaque trame. L'un des arguments de cette routine permet de discriminer s'il s'agit du premier ou second passage. Pour chaque trame, la routine de personnalité va vérifier s'il y a un bloc `catch` qui peut gérer l'exception. S'il y en a un, la routine retourne. S'il n'y en a pas, elle exécute les destructeurs des objets qui ont été alloués et indique de continuer la boucle.
8. La fonction `Unwind_RaiseException` a trouvé un bloc `catch` capable de gérer l'exception, elle va donc rendre la main à ce bloc.

En observant les sections d'un fichier ELF, les différents blocs de code susceptibles de lancer une exception font l'objet d'une entrée de type *Frame Description Entry* (FDE) à l'intérieur de la section `.eh_frame`. Les différents blocs `catch` pour une entrée FDE sont stockés dans une entrée de type *Language Specific Data Area* (LSDA) à l'intérieur de la section `.gcc_except_table`. Cette entrée contient tous les blocs `catch`, appelés *call sites*. La fonction `Unwind_RaiseException` parcourt les entrées FDE pour dérouler la pile, tandis que la routine de personnalité parcourt les *call sites* du LSDA pour en trouver un capable de gérer l'exception lancée.

Afin de gérer les différentes opérations (e.g. restauration du contexte entre plusieurs trames de pile), des instructions DWARF sont présentes dans chaque FDE. Ces instructions sont exécutées par la machine virtuelle DWARF, qui est une machine à pile implémentée par libgcc.

```
*****
* Frame Descriptor Entry
*****
fde_00403218                                XREF[1]:    00403140(*)
00403218 24 00 00 00      ddw      24h      (FDE) Length
0040321c a0 00 00 00      ddw      cie_0040317c      (FDE) CIE Reference Pointer
00403220 48 fc ff ff      ddw      FUN_00402e68      (FDE) PcBegin
00403224 fc 00 00 00      ddw      FCh      (FDE) PcRange
00403228 04              uleb128  4h      (FDE) Augmentation Data Length
00403229 a3 00 00 00      ddw      lsda_exception_table_004032cc      (FDE Augmentation Data) LSDA Dat...
0040322d 41 0e 40 |      db[19]      (FDE) Call Frame Instructions
          9d 08 9e
          07 42 93 ...
```

FIGURE 10 – L'une des FDE de l'exécutable

La figure 10 montre l'une de FDE de la section `.gcc_except_table` de l'exécutable. On peut voir que cette FDE couvre le bloc de code situé entre les adresse 0x402e68 (*PcBegin*) et 0x402e68+0xfc (*PcRange*).

Suivre la trace, compter ses pas

Maintenant que j'ai plus d'informations sur comment fonctionnent les exceptions, je cherche à savoir quelle opération dans ce processus de gestion d'exception prend du temps. Comme le temps d'exécution est significativement différent entre chaque exécution, je fais l'hypothèse suivante :

Hyp : Suivant que l'exécution est lente ou rapide, le chemin de gestion de l'exception n'est pas le même.

Pour cela, j'opte finalement pour l'utilisation de gdb. La commande `shell` permet d'exécuter une commande bash. De plus, gdb permet de définir des commandes à effectuer pour chaque point d'arrêt. Je peux alors utiliser ces deux mécanismes pour implémenter un système de traçage avec un script gdb. Le principe est le suivant :

- Un script Python (`/tmp/timestamp.py`) affiche un horodatage précis à la milliseconde.
- Une commande gdb est définie pour appeler ce script
- Un point d'arrêt est défini sur chacune des fonctions impliquées dans la gestion d'exception
- À chaque fois qu'un point d'arrêt est atteint, une commande est définie pour dire à gdb d'afficher le nom de la fonction atteinte ainsi que l'horodatage courant.

Pour le reste, il est possible de retraiter ces données pour les afficher avec matplotlib (cf annexe).

Ce mécanisme implémente une fonctionnalité de 'checkpoint', où un horodatage est émis à chaque point de passage. Il n'y a donc pas réellement d'informations sur le temps passé par une fonction donnée. Cependant, l'information que l'on retrouve ici est le temps mis entre deux points de passage successifs. En réalisant une trace pour une exécution lente et une pour une exécution rapide (en faisant varier la longueur de l'argument), il sera possible de savoir :

- Si le chemin d'exécution est différent (e.g. si les listes de points de passage sont différentes)
- Si certaines opérations paraissent anormalement longues

Les scripts gdb et Python permettant de réaliser et visualiser ces traces se trouvent en annexe. En comparant la liste des fonctions appelées par chacune des exécutions, je constate que c'est la même. Il n'y a pas de différence entre les chemins d'exécution, ce qui invalide mon hypothèse.

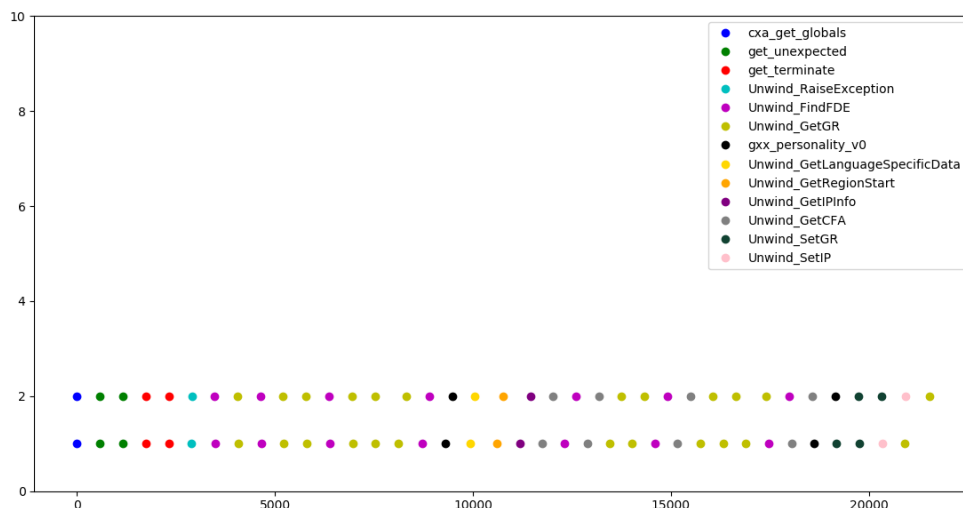


FIGURE 11 – Différences de temps d'exécution

Les deux traces sont visibles sur la figure 11. La ligne du bas correspond à une trace réalisée lorsque

l'argument d'entrée avait une longueur de 31 caractères et la ligne du haut correspond à une trace réalisée lorsque l'argument d'entrée avait une longueur de 32 caractères.

Je retrouve le mécanisme de gestion d'exceptions attendu, avec l'itération à travers les FDE, les deux appels à la routine de personnalité etc. . . Le différence de temps d'exécution n'est pas aussi flagrante qu'avant (où une exécution rapide prenait cinq fois moins de temps qu'une exécution lente) car l'instrumentation avec un mélange de script gdb et Python pour réaliser ces traces rajoute un bruit non négligeable. Cette différence est tout de même visible à l'oeil nu : sur les premiers points de passage les deux traces sont synchronisées, mais à la fin elles sont nettement désynchronisées.

Je constate que cette désynchronisation se fait en deux fois, mais deux fois au même endroit : lorsque la fonction `_Unwind_RaiseException` itère à travers les FDE. Avec les informations collectées jusqu'à présent, je sais que du code DWARF est exécuté à peu près à ce niveau dans le mécanisme de gestion d'exception.

Hyp : Le code DWARF exécuté est à l'origine des différences de temps d'exécution

Pour vérifier cette hypothèse, je choisis de déboguer la machine virtuelle DWARF avec gdb. En regardant le code source de la libgcc, je trouve que la fonction qui exécute une instruction DWARF est `execute_stack_op`. Comme ce symbole n'est pas présent dans la version de la bibliothèque partagée utilisée par le système, je la récupère via SSH, la charge dans ghidra à l'adresse 0x100000 et laisse le logiciel analyser la bibliothèque.

En regardant son code source, je m'aperçois que cette fonction est très longue (plus de 400 lignes), imbrique plusieurs étages de `switch ... case` et appelle la fonction `_Unwind_GetGR`. Dans ghidra, en triant les fonctions par taille décroissante, la première fonction qui n'a pas de nom associé répond à ces critères : elle est à l'adresse 0x10b8b4. En comparant le code source avec sa version décompilée, je confirme qu'il s'agit effectivement de la fonction `execute_stack_op`.

Comme la bibliothèque est chargée dynamiquement, son adresse de base n'est pas toujours la même. Je choisis arbitrairement une fonction qui a conservé son symbole (en l'occurrence `_Unwind_Resume`) et je calcule son décalage (en l'occurrence -3516) par rapport à cette fonction. Je trouve aussi l'adresse de l'instruction qui charge l'instruction DWARF courante de cette manière là (située à `_Unwind_Resume-3392`). Dès lors, je peux voir quelles sont les instructions DWARF qui sont exécutées et savoir si elles sont effectivement différentes.

Le script gdb qui permet de tracer les instructions DWARF exécutées se trouve aussi en annexe.

Je constate que le code DWARF est exécuté deux fois, une fois au cours de chacune des itérations sur les FDE. Lors d'une exécution rapide, le code DWARF exécuté fait 30 instructions. Lors d'une exécution lente, ce sont plusieurs milliers d'instructions DWARF qui sont exécutées (comme cela prenait trop de temps, j'ai arrêté l'exécution avant la fin). Ceci valide l'hypothèse selon laquelle c'est le code DWARF qui est utilisé pour faire des opérations à l'intérieur du programme : lancer l'exception avec `argv[1]` sert juste à lancer un programme compilé en DWARF avec l'argument en entrée.

À ce moment-là, je passe d'abord un certain temps à déboguer la VM et à observer les instructions DWARF exécutées. Typiquement, les premières instructions exécutées sont :

```
2f 6f 08 22 06 08 22 06 12 06 16 08 22 12 06 16
08 22 12 06 16 08 22 12 06 16 08 22 94 28 0e 2f
```

En discutant de mon avancement avec mes collègues¹⁶, l'un d'eux me montre la section `.gnu.hash` qui lui semble étrange. Effectivement, après les éléments standards de cette section qui sont reconnus par ghidra, un long tableau de données non reconnues est présent à l'adresse 0x400258. Le début de ce tableau est :

16. Après avoir demandé s'ils ne comptaient pas faire cette étape du challenge.

```
6f 08 a8 22 06 08 08 22 06 12 06 16 08 08 22 12 06 16
08 08 22 12 06 16 08 08 22 12 06 16 08 08 22 94 01 28
```

Je reconnais là le code DWARF que je cherchais, avec certaines des instructions qui prennent des arguments. Je confirme que c'est bien le code qui est exécuté en plaçant un `watchpoint` gdb avec la commande `rwatch *0x400258` : Dès que la valeur située à l'adresse 0x400258 est lue, gdb suspend l'exécution. Après coup, je me suis aperçu qu'en débarrassant la machine virtuelle je pouvait aussi simplement lire l'adresse depuis laquelle est chargé chaque opcode, ce qui permet aussi de trouver le code DWARF. Celui-ci s'arrête à l'adresse 0x401982, le code fait donc prêt de 6000 octets.

Enfin, quant à savoir pourquoi ce code est exécuté en premier lieu, il est possible d'afficher le contenu des sections relatives à la gestion d'exception avec la commande `readelf -w`. Sur l'entrée FDE en question, le code DWARF désassemblé est :

```
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
```

Il y effectivement un saut qui a été ajouté par rapport au code de gestion d'exception classique, ce qui redirige le programme DWARF vers l'adresse 0x400258. De plus, le résultat de ce saut est mis dans le registre x28. Pour rappel, il avait été déterminé que lorsque l'exécution du programme reprenait à l'adresse 0x402f20, il se contentait d'afficher la chaîne de caractères contenue justement dans le registre x28.

Maintenant que j'ai trouvé la partie du code qui fait des choses avec l'argument qui lui est passé, je peux récupérer le code DWARF et faire sa rétro-ingénierie pour retrouver la clé.

Un langage bien sympathique

Cours de nanique

J'ai maintenant extrait un blob de données depuis la section `.gnu.hash` qui contient du code DWARF. Assez naturellement, je fais l'hypothèse suivante :

Hyp : Le blob contenu dans la section `'gnu.hash'` ne contient que du code DWARF.

Je cherche maintenant un désassembleur de code DWARF afin de pouvoir effectuer la rétro-ingénierie du programme. Si quelques logiciels sont capables de désassembler le code DWARF contenu dans les sections d'un fichier ELF liées à la gestion d'exceptions, je ne trouve pas de logiciel ou de bibliothèque qui me permettrait de donner un blob et de récupérer du DWARF désassemblé. Je décide donc de faire mon propre désassembleur en Golang dont le code se trouve en annexe¹⁷.

J'ai maintenant désassemblé le programme et je peux observer qu'il fait beaucoup de manipulations binaires (xors, décalages etc). Néanmoins, un bout du programme, situé au quart du blob n'a pas l'air de se désassembler correctement, avec par exemple des instructions qui ne sont pas reconnues.

17. Golang dispose par exemple d'un package `debug/dwarf`, mais qui ne permet pas simplement de désassembler du code arbitraire.

Le début du code DWARF exécuté est :

```
0x400258+0000: DW_OP_reg31
0x400258+0001: DW_OP_const1u 0xa8
0x400258+0003: DW_OP_plus
0x400258+0004: DW_OP_deref
0x400258+0005: DW_OP_const1u 0x8
0x400258+0007: DW_OP_plus
0x400258+0008: DW_OP_deref
0x400258+0009: DW_OP_dup
0x400258+000a: DW_OP_deref
0x400258+000b: DW_OP_swap
0x400258+000c: DW_OP_const1u 0x8
0x400258+000e: DW_OP_plus
0x400258+000f: DW_OP_dup
0x400258+0010: DW_OP_deref
0x400258+0011: DW_OP_swap
0x400258+0012: DW_OP_const1u 0x8
0x400258+0014: DW_OP_plus
0x400258+0015: DW_OP_dup
0x400258+0016: DW_OP_deref
0x400258+0017: DW_OP_swap
0x400258+0018: DW_OP_const1u 0x8
0x400258+001a: DW_OP_plus
0x400258+001b: DW_OP_dup
0x400258+001c: DW_OP_deref
0x400258+001d: DW_OP_swap
0x400258+001e: DW_OP_const1u 0x8
0x400258+0020: DW_OP_plus
0x400258+0021: DW_OP_deref_size 0x1
0x400258+0023: DW_OP_bra 0x44
0x400258+0026: DW_OP_pick 0x3
0x400258+0028: DW_OP_pick 0x3
0x400258+002a: DW_OP_skip 0x49
0x400258+002d: DW_OP_const8u 0x658302a68e8e1c24
0x400258+0036: DW_OP_xor
0x400258+0037: DW_OP_swap
0x400258+0038: DW_OP_const8u 0xdc7564f1612e5347
0x400258+0041: DW_OP_xor
0x400258+0042: DW_OP_plus
0x400258+0043: DW_OP_swap
0x400258+0044: DW_OP_const8u 0xd9c69b74a86ec613
0x400258+004d: DW_OP_xor
0x400258+004e: DW_OP_plus
0x400258+004f: DW_OP_swap
0x400258+0050: DW_OP_const8u 0x65850b36e76aaed5
0x400258+0059: DW_OP_xor
0x400258+005a: DW_OP_plus
0x400258+005b: DW_OP_bra 0xc
0x400258+005e: DW_OP_const8u 0x403098
0x400258+0067: DW_OP_skip 0x7fff
0x400258+006a: DW_OP_const8u 0x4030b8
0x400258+0073: DW_OP_skip 0x7fff
```

Sans trop chercher à comprendre pour le moment, je constate qu’au début, le programme fait 4 opérations `DW_OP_deref` avec une adresse qui est incrémentée de 8 octets à chaque fois. Cette opération déréfère l’adresse qui est en haut de la pile et stocke le résultat sur la pile. Il pourrait par exemple s’agir des 32 premiers octets de l’argument passé en entrée.

Par ailleurs, on peut voir aussi que 4 valeurs de 8 octets sont xorées avec des constantes. Enfin, les constantes `0x403098` et `0x4030b8` correspondent à l’adresse des chaînes de caractère “That’s the correct flag :)” et “Not good” respectivement dans le programme. Tous ces éléments semblent indiquer que je suis sur la bonne voie : le code DWARF fait des choses avec l’argument passé au programme, puis vérifie cette valeur par rapport à une valeur statique de 32 octets. Suivant le résultat, l’exception retourne avec l’adresse d’une chaîne de caractères qui sera affichée lorsque le cours du programme reprendra.

Une machine plutôt simple

La première étape est de retrouver le flot de contrôle du programme. En effet, il n’est pas du tout linéaire comme je l’avais espéré au début. J’identifie les blocs de code suivant une logique simple : Dès que le code fait un saut (instruction `DW_OP_skip`) ou un saut conditionnel (instruction `DW_OP_bra`), je rajoute un label. Ceci permet d’identifier des blocs de code. Une fois que tous les labels sont posés, je remplace l’argument des opérations de saut par le label correspondant. Ceci me permet de retrouver le flot de contrôle du programme.

En effectuant ce travail, je m’aperçois qu’une portion du programme n’est jamais exécuté : il s’agit de la portion que je n’avais pas réussi à désassembler, qui commence à l’adresse `0x400640` et termine à l’adresse `0x400ab4`. Par ailleurs, en parcourant le code j’ai pu observer le motif suivant :

```
0x400258+02b7: DW_OP_const8u 0x400648
0x400258+02c0: DW_OP_plus
0x400258+02c1: DW_OP_deref
```

L’adresse `0x400648` est chargée sur la pile, puis additionnée avec le second élément de la pile. L’adresse qui en résulte est déréférée et la valeur est mise sur le haut de la pile. Ceci semble indiquer que des données sont présentes au milieu du code DWARF, ce qui invalide l’une de mes hypothèses : la section `.gnu.hash` ne contient pas que du code DWARF.

Avec ces éléments, j’en déduis que la structure du blob contenu dans la section `.gnu.hash` est finalement :

```
+-----+ <- 0x400258
|       |
|   code |
|  DWARF |
|       |
+-----+ <- 0x400640
|       |
|  données |
|       |
+ +-----+ <- 0x400ab4
+--+
|       |
|   code |
|  DWARF |
|       |
+-----+ <- 0x401980
```


La figure 12 montre le flot de contrôle du programme DWARF tel qu’initialement retrouvé. Mon objectif suivant a été de faire une implémentation naïve mais fonctionnelle de cet algorithme, sans changer sa structure afin de minimiser les risques d’erreurs.

Un automate passe

J’ai donc travaillé bloc de code par bloc de code pour avoir une première implémentation fonctionnelle en Python. Afin de m’aider dans ma tâche, j’ai :

- Développé une machine à pile minimaliste en Golang afin de pouvoir étudier précisément l’effet de bloc de code;
- Instrumenté la machine virtuelle de la libgcc avec gdb, pour pouvoir déboguer l’algorithme et observer les états intermédiaires pour une entrée donnée.

La machine virtuelle DWARF est une machine à pile et les instructions exécutées par ce programme sont très simples : il s’agit dans la grande majorité des cas de rajouter une valeur sur la pile ou d’effectuer une opération avec la ou les premières valeurs stockées sur la pile. Les instructions les plus complexes utilisées sont les sauts, cependant comme je cherche à étudier l’effet de chaque bloc de code situé entre deux sauts, il n’est pas nécessaire de les réimplémenter.

Une machine à pile est une machine qui stocke ses valeurs intermédiaires dans une pile. Les deux primitives standards sont d’empiler une donnée ou de dépiler une donnée. Pour effectuer une opération, une ou deux valeurs sont dépilées (suivant que l’opération prend un ou deux opérandes) et le résultat est rajouté sur le haut de la pile.

Par exemple, voici l’évolution de la pile lorsque le code DWARF suivant est exécuté :

```
+-----+ ; La pile est vide

DW_OP_const1u 0xee

+-----+ ; La pile contient un élément
|  0xee  |
+-----+

DW_OP_lit18

+-----+ ; La pile contient deux éléments
|  0x12  |
+-----+
|  0xee  |
+-----+

DW_OP_const1u 0x34

+-----+ ; La pile contient trois éléments
|  0x34  |
+-----+
|  0x12  |
+-----+
|  0xee  |
+-----+

DW_OP_plus

+-----+ ; La pile contient deux éléments
|  0x46  | ; 0x12 + 0x34 = 0x46
```

```

+-----+
|  0xee  |
+-----+

DW_OP_xor

+-----+
|  0xa8  | ; 0x46 ^ 0xee = 0xa8
+-----+

```

L'implémentation d'une telle machine à pile est simple et le code se trouve en annexe. Il ne s'agit pas à proprement parler d'une machine virtuelle DWARF puisque j'ai fait le choix de simplifier les mnémoniques utilisés et d'unifier les différentes manières définies dans le standard DWARF de charger un élément sur la pile. Par exemple, le programme précédent s'écrit ainsi avec mon implémentation :

```

push 0xee
push 0x12
push 0x34
plus
xor

```

Cette méthode permet de fidèlement étudier les opérations faites par chaque bloc de code indépendamment les uns des autres, ce qui m'a permis de les réimplémenter un par un. J'ai rajouté une instruction `print` permettant d'afficher l'état de la pile, qui a permis d'afficher les états intermédiaires. J'ai aussi rajouté les opérations permettant de lire le bloc de données inclus dans la section `.gnu.hash`.

Dans un second temps, j'ai aussi instrumenté la machine virtuelle de la libgcc avec gdb. En reprenant les points d'arrêt déjà déterminés précédemment, à l'intérieur de la libgcc l'état de la machine virtuelle DWARF est le suivant :

- x0 contient l'adresse de l'instruction courante
- x7 contient l'opcode de l'instruction courante
- x26 contient l'adresse du début de la pile
- x19 contient le nombre d'éléments dans la pile actuellement
- x6 contient l'argument de l'instruction (lorsqu'il y en a un).

Avec ces éléments, il est possible de tracer l'évolution de la pile après chaque instruction. Néanmoins cette opération prend trop de temps et mon ordinateur a crashé avant de pouvoir compléter une telle trace (après déjà cinq heures de calcul). Au final, en utilisant la fonction `rwatch` de gdb pour mettre un point d'arrêt sur la lecture d'une adresse, il est possible de s'arrêter à certains moments choisis de l'algorithme. Ceci a permis de vérifier que le flot de contrôle retrouvé était correct et de le corriger le cas échéant.

En combinant ces deux approches, j'ai petit à petit réimplémenté en Python chacun des blocs de la première section de code DWARF (de l'adresse 0x400258 à l'adresse 0x400640), soit un peu moins de 21% du code. La deuxième section correspond à l'étape `shuffle8` sur la figure 12 : le code de cette section est linéaire, à l'exception d'une boucle interne au milieu. En commençant à faire la rétro-ingénierie de cette section, je m'aperçois petit à petit que les mêmes motifs reviennent tout le temps. Par exemple, les motifs suivants en font partie :

```

1:
DW_OP_const4u <cst>
DW_OP_plus

```

```

DW_OP_const4u 0xffffffff
DW_OP_and

2:
DW_OP_dup
DW_OP_dup
DW_OP_const1u <cst1>
DW_OP_shr
DW_OP_swap
DW_OP_const1u <cst2>
DW_OP_shl
DW_OP_or
DW_OP_const4u 0xffffffff
DW_OP_and
DW_OP_plus
DW_OP_const4u 0xffffffff
DW_OP_and

```

En appelant `s0` la valeur du haut de la pile, ces deux motifs peuvent être implémentés par le code Python suivant :

```

# Motif 1
s0 = (s0 + cst) & 0xffffffff

# Motif 2
s0 = (((s0 >> cst1) | (s0 << cst2)) & 0xffffffff) + s0 & 0xffffffff

```

Au final, je fais l'hypothèse suivante :

Hyp : La seconde section de code est composée majoritairement d'un nombre de motifs restreints qui se répètent durant toute la section.

Sous cette hypothèse, il est possible de générer le code Python qui implémente les opérations faites par cette seconde section en utilisant sur un automate déterministe à états finis. Étant donné que j'avais déjà passé plusieurs heures à réimplémenter 21% de l'algorithme, la possibilité de réimplémenter les 79% restant par génération de code semblait plus efficace, tout en minimisant les risques d'erreurs.

Il est aussi possible de réaliser cette génération de code avec des expressions régulières (comme la fonction `substitute` de la commande `sed`)¹⁸, néanmoins comme celles-ci auraient été plutôt complexes. J'ai préféré passer un peu plus de temps à implémenter l'automate manuellement pour diminuer de beaucoup le risque d'erreur.

La figure 13 présente l'automate qui a été utilisé pour la génération de code. Lorsque l'automate arrive dans les des états finaux (en pointillés), il repasse à l'état initial. Par hypothèse, la grande majorité du code de cette section sera reconnu par cette automate.

Le code qui implémente cet automate est présent en annexe. Dès que l'un des états finaux est atteint, une ligne de Python est générée, suivant le chemin parcouru et les constantes présentes. Le code généré (version réduite) est le suivant :

18. Les expressions régulières sont implémentées comme des automates à états finis, il s'agit donc en réalité de la même approche d'un point de vue théorique.

successifs sur certains point d'arrêt, il est possible de voir les compteurs d'itération des boucles évoluer à l'intérieur de la pile.

Les branchements conditionnels peuvent être implémentés en DWARF avec l'instruction `DW_OP_bra`. Cette instruction prend un seul opérande : un décalage de saut. Si la valeur située sur le haut de la pile est 0, alors le programme continue son exécution et ne saute pas. Si elle est différente de 0, alors le programme effectue le saut.

Cette instruction est utilisée pour deux choses :

- Implémenter des boucles
- Implémenter une bascule de chemin d'exécution

Un exemple d'application du premier cas est la séquence d'instructions suivantes qui correspond à la fin de la fonction `prepare_verif`

```
; Met le compteur de boucle sur le haut de la pile
<0x400258+00a1> DW_OP_pick 5
; Met la valeur 1 sur le haut de la pile
<0x400258+00a3> DW_OP_lit1
; Dépille les deux premiers éléments, les additionne et met le résultat
  sur la pile (incrément)
<0x400258+00a4> DW_OP_plus
; Duplique le haut de la pile
<0x400258+00a5> DW_OP_dup
; Met 4 sur le haut de la pile
<0x400258+00a6> DW_OP_lit4
; Dépille les deux premiers éléments, soustrait le premier au second et
  met le résultat sur la pile
<0x400258+00a7> DW_OP_minus
; Si le résultat est différent de 0, saute sur le label 'continue_init2'
<0x400258+00a8> DW_OP_bra continue_init2
; Si le résultat vaut 0, ré-arrange la pile et saute sur le label 'verif
  '
<0x400258+00ab> DW_OP_drop
<0x400258+00ac> DW_OP_pick 3
<0x400258+00ae> DW_OP_pick 3
<0x400258+00b0> DW_OP_skip verif
```

Initialement, le compteur vaut 0. Ce bloc de code implémente donc assez simplement une vérification : si le compteur vaut 4 alors la boucle s'arrête et l'algorithme continue. Je suis donc face à un `for i = 0; i < 4; i++`. Comme indiqué sur la figure 12, cet algorithme se compose de plusieurs boucles qui sont toutes implémentées de cette manière.

Pour le second cas, un exemple est le choix du chemin d'exécution fait par la fonction `prep4_shuffle9`. En réalité, cette exécution est toujours linéaire : l'algorithme commence par exécuter les fonctions `prep_shuffle10`, `shuffle10` et `shuffle11`. Ensuite, le bit de bascule est inversé et le second chemin est pris : `prep_shuffle12`, `shuffle12` puis `shuffle13`. Le même mécanisme est utilisé par la fonction `after_shuffle3`. Il est possible de supprimer la bascule et de linéariser l'exécution qui correspond à la succession de fonctions `prep_shuffle10`, `shuffle10`, `shuffle11`, `prep_shuffle12`, `shuffle12` et enfin `shuffle13`.

Un autre motif qui revient souvent dans l'algorithme est le `rot rot drop` (la suite d'octets correspondante dans le code est `17 17 13`). Pour une pile contenant 3 éléments, les effets de cette succession d'opérations sont :

```

+-----+
|      s0      |
+-----+
|      s1      |
+-----+
|      s2      |
+-----+

DW_OP_rot
DW_OP_rot

+-----+
|      s2      |
+-----+
|      s0      |
+-----+
|      s1      |
+-----+

DW_OP_drop

+-----+
|      s0      |
+-----+
|      s1      |
+-----+

```

Ces opérations permettent donc de sauvegarder les deux valeurs situées en haut de la pile, tout en détruisant les suivantes. Il s'agit de la manière choisie pour nettoyer la pile des valeurs de calcul intermédiaires.

Finalement, après avoir progressivement retrouvé les boucles et identifiés les éléments constitutifs de l'algorithme, j'arrive peu à peu à le linéariser : passer de la structure récursive initiale à une structure entièrement itérative. Vient ensuite la simplification de l'algorithme, pour fusionner les étapes qui peuvent l'être, supprimer certains états dans la pile qui ne sont pas utilisés, etc...

Toutes ces opérations ont pour but de simplifier l'algorithme pour pouvoir plus simplement l'appréhender, donc plus simplement l'inverser. Le dernier élément que je remarque est l'utilisation de la fonction `shuffle8` dont le code a été généré par automate. Cette fonction est appelée à un seul endroit qui est le suivant :

```

DWARF :
<0x400258+0385> DW_OP_bra <0x400258+0390>
<0x400258+0388> DW_OP_const4u 0x17246549
<0x400258+038d> DW_OP_skip <0x400258+0395>
<0x400258+0390> DW_OP_const4u 0x84653217
<0x400258+0395> DW_OP_skip shuffle8

Python :
if stack.pop() == 0:
    stack.append(0x17246549)
else:
    stack.append(0x84653217)
shuffle8(stack)

```

Or la fonction `shuffle8` possède une propriété particulière : elle ne travaille que sur le haut de la pile et ne modifie rien d'autre. Comme elle est appelée avec deux valeurs qui sont statiques, il est possible de remplacer l'appel à `shuffle8` directement par le résultat du calcul. Ainsi, l'implémentation devient :

```
if stack.pop() == 0:
    stack.append(0x818f694a)
else:
    stack.append(0x60bf080f)
# La fonction shuffle8 saute sur la fonction prep_shuffle9 à la fin de
# son exécution
prep_shuffle9(stack)
```

Ces valeurs ont été obtenues par le code généré, puis vérifiées par l'instrumentation de la libgcc, ce qui valide au final l'approche par génération de code.

Au terme de ces simplifications, j'arrive sur une structure d'algorithme linéaire. L'implémentation complète se trouve en annexe, mais globalement la structure est la suivante :

```
# Boucle 1
for i in range(8):
    # ...
    for j in range(4):
        # ...

        # Boucle 2
        for j in range(15):
            # ...
            for k in range(j+1):
                # ...
            # ...
            for k in range(6):
                # ...
            # ...
        # ...
    # ...

verif(stack)
```

Je n'ai pas réussi à me débarrasser totalement de la pile, même si j'ai pu réduire le nombre de valeurs intermédiaires qu'elle contient.

Pas à pas

Maintenant que l'algorithme est correctement implémenté, je peux chercher à l'inverser : connaissant le résultat attendu, je dois trouver une entrée qui donne ce résultat. En considérant que j'ai à faire à une sorte de fonction de hachage, je dois en trouver une pré-image.

Hyp : Il est possible de remonter l'algorithme pas à pas pour l'inverser.

Je commence donc à reprendre mon implémentation et à essayer de l'inverser en partant de la fin. Il se trouve que rapidement je tombe sur un problème : en remontant pas à pas l'algorithme, j'arrive sur une valeur du type $x \oplus y$ et j'ai besoin de x pour pouvoir continuer, sans avoir aucune information sur y .

En essayant de voir comment il est dérivé, ainsi que certaines hypothèses sur des valeurs précédentes, je n'arrive à rien déduire. J'abandonne donc rapidement cette approche et l'algorithme n'est visiblement pas inversible pas à pas.

Sauvés par la teinture

Je tente alors une approche plus méthodique en effectuant une analyse par teinte : l'état de l'algorithme est sa pile et je connais l'état final. J'essaie de propager cet état final en remontant dans l'algorithme, mais cette fois-ci de manière structurée. J'ai fait l'analyse sur papier avec deux couleurs initialement, mais la version présentée ici est plus visuelle pour voir plus directement les dépendances entre les différents éléments.

Le résultat de cette analyse est montré en figure 14. Les labels indiqués correspondent à la version finale de l'algorithme, qui se déroule de bas en haut (mais l'analyse commence par la fin, d'où l'orientation). Les boucles ne sont pas toutes présentées ici, elles sont pour la plupart cachées dans les étapes intermédiaires. L'analyse est faite ici sur le dernier tour de la boucle 2, cependant elle reste valide pour les autres, il faut donc itérer 16 fois ce processus pour avoir l'analyse complète d'un tour de la boucle 1. Les étapes principales de cette analyse ont été numérotées de 1 à 8. L'état initial sera noté *ABCD* (correspondant aux couleurs jaune, vert, bleu et rouge), l'état final est noté *EFCD* (correspondant aux couleurs orange, cyan, bleu, rouge) et la pile est systématiquement présentée de droite à gauche : le haut de la pile après chaque étape est aligné à gauche. Enfin, les cases blanches correspondent à des éléments inutiles pour l'analyse, tandis que les cases noires correspondent à des éléments connus de l'état (e.g. compteurs de boucle).

1. L'état initial est *ABCD*.
2. Partant de l'état *ABCD*, les éléments *C* et *D* font aussi partie de l'état final *EFCD*. Ils sont remontés le plus possible le long de l'algorithme. Ils sont dérivés en deux éléments distincts (caractérisés par une couleur unique : violet) qui sont eux aussi remontés le plus possible.
3. Partant de l'état *ABCD*, les éléments *A* et *B* sont descendus le plus possible le long de l'algorithme.
4. À la sortie de la fonction `end_prepare_stack` et juste avant la fonction `diversify_step2`, je sais que l'élément *A* se retrouve en 5e position sur la pile. Il se trouve que les deux premiers éléments sont respectivement les 32 bits de poids fort et les 32 bits de poids faible de *A*. Je peux donc dériver les deux premiers éléments de la pile à partir de *A* et les remonter le plus possible.
5. Je connais les 4 premiers éléments de la pile (nommés ici *s0*, *s1*, *s2*, *s3*) à la sortie de la fonction `end_prepare_stack` et le second élément (nommé *v1*) à la sortie de la fonction `diversify_step3`. Je peux retrouver le cinquième élément de la pile (*s4* ou *E*¹⁹, en orange) puis le descendre le long de l'algorithme. Les équations pour le retrouver sont :

$$\text{ror_32}(x, s) = ((x \gg s) | (x \ll (32 - s))) \& 0xffffffff$$

$$\text{hi}(x) = x \gg 32$$

$$\text{lo}(x) = x \& 0xffffffff$$

$$\text{hi_s4} = \text{ror_32}(\text{hi}(v1 \oplus s2), 0x12) \oplus s0 \oplus s1$$

$$\text{lo_s4} = \text{ror_32}(\text{lo}(v1 \oplus s2), 0x1a) \oplus s0$$

$$s4 = (\text{hi_s4} \ll 32) | \text{lo_s4}$$

6. À la sortie de la fonction `inner_step4` et juste avant la fonction `diversify_prepare`, je sais que l'élément *E* se retrouve en 7e position sur la pile. Il se trouve que les deux premiers éléments sont respectivement les 32 bits de poids fort et les 32 bits de poids faible de *E*. Je peux donc dériver les deux premiers éléments de la pile à partir de *E* et les remonter le plus possible.

19. Aussi nommé Leeloo.

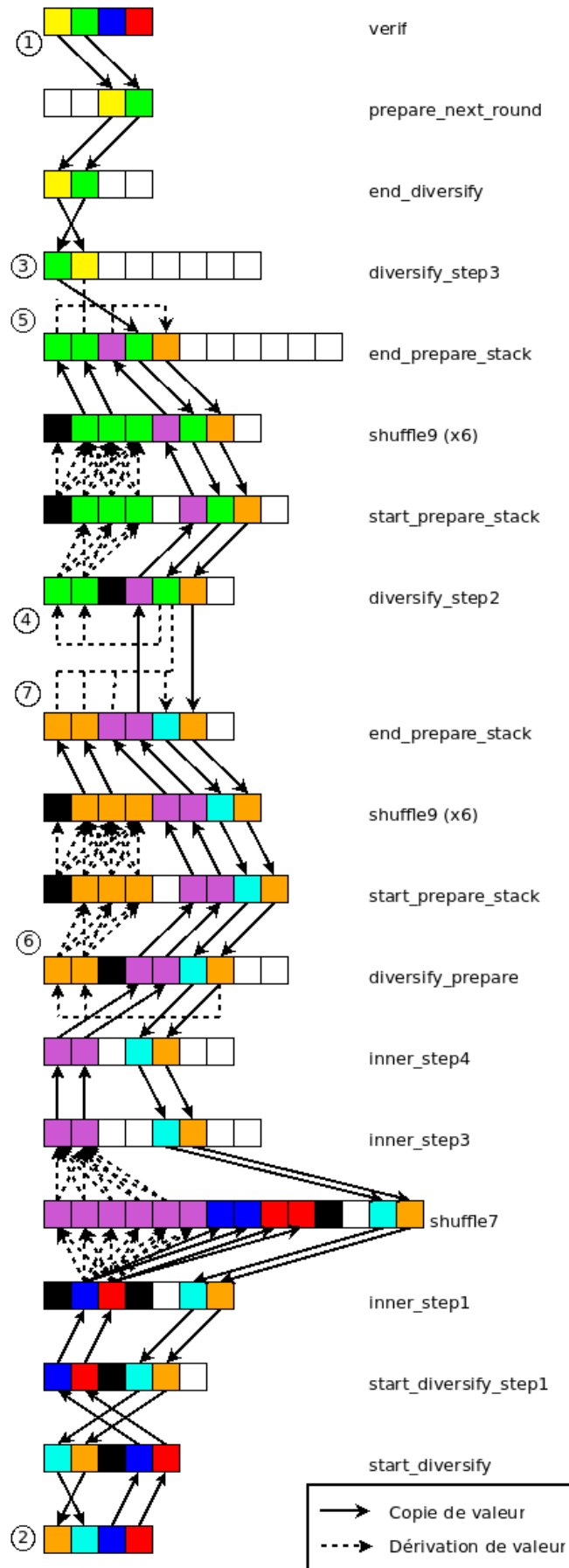


FIGURE 14 – Analyse par teinte de l'algorithme

7. Je connais les 4 premiers éléments de la pile (nommés ici s_0 , s_1 , s_2 et s_3) à la sortie de la fonction `end_prepare_stack` et le cinquième élément (nommé v_1) à la sortie de la fonction `diversify_step2`. Je peux retrouver le cinquième élément de la pile (s_4 ou F , en cyan) puis le descendre le long de l'algorithme. Les équations pour le retrouver sont :

$$\begin{aligned}hi_s4 &= \text{ror_32}(hi(v_1 \oplus s_2) \oplus s_1, 0xe) \oplus s_0 \\lo_s4 &= \text{ror_32}(lo(v_1 \oplus s_2) \oplus s_0, 0x4) \oplus s_1 \\s4 &= (hi_s4 << 32) | lo_s4\end{aligned}$$

8. L'état final est reconstitué, il s'agit de *EFCD*.

En effectuant 16 fois ces opérations, il est possible de retrouver l'état intermédiaire au 7e tour de l'algorithme, qui est *E'F'CD*. À ce moment là, avec la recombinaison des résultats qui est effectué, le nouvel état initial à inverser est *CDE'F'*. Le processus peut être réitéré pour inverser complètement l'algorithme.

L'analyse par teinte m'a permise de mettre en évidence la structure de l'algorithme, pour au final réussir à l'inverser. Il est désormais possible d'écrire un programme qui va retrouver la clé cherchée. Ce programme se trouve en annexe, il est inclus avec le code de l'algorithme puisqu'il appelle les mêmes fonctions.

En inversant l'état final, la clé est `SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}`

En ajoutant cette clé avec le script `/root/tools/add_key.py`, le conteneur est déchiffré avec le message suivant :

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

J'ai le troisième flag et la quatrième épreuve semble débloquée. Le fichier contenu dans le coffre-fort est déchiffré et je vais voir ce qu'il en est.

Environnement d'exécution sécurisé ?

J'utilise à nouveau la connexion SSH à la machine virtuelle pour copier le fichier déchiffré sur mon ordinateur dans le but de l'analyser.

Un nouveau participant apparaît

Délégation de pouvoir

La commande `file` permet de confirmer que c'est encore un binaire dont je dispose, un exécutable 64 bits au format ELF pour l'architecture AArch64. Rien de surprenant donc. Néanmoins, je constate qu'il s'agit d'un 'gros' binaire, puisqu'il fait quand même 1,5Mo. Il y a cette fois-ci beaucoup plus de chaînes de caractères définies, mais dont une partie au moins correspond à la `libc`.

En cherchant pour des références à `sstic` et les résultats alentours, voici le résultat :

```
# strings decrypted_file | grep -i sstic -A 5 -B 5
8)HXa[0
*IG6
`'8D
(L/usage: %s [32-bytes-key-hex-encoded]
can't decode hex
/dev/sstic
Loose
Failure
../csu/libc-start.c
__ehdr_start.e_phentsize == sizeof *GL(dl_phdr)
unexpected reloc type in static binary
```

Cinq chaînes de caractères sortent du lot :

- “usage: %s [32-bytes-key-hex-encoded]”
- “can't decode hex”
- “/dev/sstic”
- “Loose”
- “Failure”

Si je me contente de cette analyse, je peux perdre (“Loose”) ou échouer (“Failure”), ce qui n'est pas très encourageant. Peut-être que d'autres chaînes de caractères sont utilisées et que je les verrai plus tard. En lançant le programme avec une entrée arbitraire de 32 octets en hexadécimal, il fait quelques calculs puis s'arrête en affichant “Loose”.

Quoi qu'il se passe dans ce binaire, cela semble impliquer le **device file** `/dev/sstic`. Peut-être que le binaire `/lib/sstic.ko` va enfin s'avérer utile.

L'étape suivante est donc de le charger dans ghidra pour voir ce qu'il en est. Une fois que son analyse est terminée je regarde les références vers les chaînes de caractères déjà identifiées. Il se trouve qu'elles sont toutes utilisées par la fonction à l'adresse `0x400778`. Cette fonction semble être la fonction `main`. Elle décode l'argument, récupère un pointeur de fichier sur le fichier `/dev/sstic` puis fait des `ioctl` dessus.

Le code décompilé de ghidra est assez clair. En gardant uniquement les parties importantes, le pseudo-code correspondant est le suivant :

```
int main(int argc, char** argv) {
    char *decoded[32];
    undefined8 *ptr;
```

```

undefined8 size;
int ret;
if (argc == 2 && strlen(argv[1]) == 40) {
    int ret = decode_hex(argv[1], decoded);
    if (!ret) {
        printf("usage: %s [32-bytes-key-hex-encoded]", argv[0])
    }
    fp = open("/dev/sstic", 0)
    ptr = 0x44dbd8;
    size = 0x101010;
    ioctl(fp, 0xc0105300, ptr);
    ptr = decoded;
    size = 32;
    ioctl(fp, 0xc0105301, ptr);
    do {
        ioctl(fp, 0xc0105302, 0);
        ret = ioctl(fp, 0xc0105303, 0);
        if ((ret & 0xffff) == 1) {
            if ((ret & 0xffff0000) == 0) {
                printf("Win");
            } else {
                printf("Loose");
            }
            exit(0);
        }
    } while (ret & 0xffff) != 0xffff
    printf("Failure");
}
exit(1);
}

```

Déjà, une autre chaîne de caractères apparaît : “Win”. Celle-ci n’avait pas été reconnue par la commande `strings` puisque la longueur minimale par défaut est 4.

Je constate qu’il y a quatre `ioctl` différents utilisés. Je peux voir que le programme s’exécute en deux étapes : l’utilisation des deux premiers `ioctls`, puis une boucle qui itère et exécute les deux `ioctls` suivant à chaque tour. L’utilisation des variables que j’ai nommé `ptr` et `size` fait penser à l’envoi d’une structure qui serait stockée sur la pile. Cette structure contient apparemment une taille et une adresse vers un tampon, puisque lors du deuxième `ioctl` c’est le tampon contenant l’argument décodé ainsi que la taille de l’argument décodé (32) qui est envoyé.

Hyp : L’exécutable contient un blob stocké à l’adresse 0x44dbd8 et de taille 0x101010.

Pour cette étape, je récupère aussi le module noyau `/lib/sstic.ko` avec la connexion SSH. En l’analysant avec ghidra, je peux voir trois fonctions intéressantes : `sstic_open`, `sstic_ioctl` et `sstic_release`. En parcourant le code de la fonction `sstic_ioctl` je remarque deux appels supplémentaires définis :

- 0xc1105305
- 0xc1105306

Cependant, ce module n’a pas l’air de faire grand chose des `ioctls` qu’on lui donne. Plus précisément, il semble faire passe-plat et appeler la fonction `__arm_smccc_smc` avec tout ce que l’on lui donne. Cette fonction exécute l’instruction `smc` (Secure Monitor Call) définie par ARM. Cette instruction transfère l’exécution à l’environnement d’exécution sécurisé (TEE pour *Trusted Execution Environment*), les valeurs

étant passées par registre. Comme je ne connais pas du tout le domaine des environnements d'exécution sécurisés, j'ai considéré qu'il s'agissait d'une sorte d'ioctl vers l'environnement d'exécution sécurisé du système.

Pour les différents ioctl identifiés, voici les actions effectuées par le module noyau :

- 0xc0105300 : copie le tampon de l'espace utilisateur à l'espace noyau et appelle la fonction `__arm_smccc_smc` avec l'identifiant 0x83010004.
- 0xc0105301 : copie le tampon de l'espace utilisateur à l'espace noyau et appelle la fonction `__arm_smccc_smc` avec l'identifiant 0xf2005003.
- 0xc0105302 : appelle la fonction `__arm_smccc_smc` avec l'identifiant 0xf2005001.
- 0xc0105303 : appelle la fonction `__arm_smccc_smc` avec l'identifiant 0xf2005002.
- 0xc1105305 : appelle la fonction `__arm_smccc_smc` avec l'identifiant 0x83010005 et copie le tampon récupéré vers l'espace utilisateur.
- 0xc1105306 : copie le tampon de l'espace utilisateur à l'espace noyau et appelle la fonction `__arm_smccc_smc` avec l'identifiant 0x83010006.

Initialement, je ne constate aucun traitement effectué par le module noyau. En réalité, la seule opération qu'il fait qui le distingue d'un complet pass-plat est d'envoyer la clé en 8 morceaux de 4 octets chacun. Il ne fait donc pas un mais 8 appels à la fonction `__arm_smccc_smc` avec l'identifiant 0xf2005003. Étant donné que je n'ai aucune information supplémentaire, je fais l'hypothèse suivante :

Hyp : Il est possible de retrouver la clé en boîte noire.

Je cherche donc à récupérer des informations sur ce que fait le binaire. Trois pistes me viennent à l'esprit :

- Regarder si les ioctls inconnus ne sont pas utilisés par d'autres exécutables du système
- Regarder le blob et voir ce qu'il contient
- Étudier le nombre de tours que fait la boucle dans la deuxième étape de l'exécutable

Dans le noir

Je trouve assez vite une utilisation légitime des ioctls en question : ils sont utilisés par l'outil `/root/tools/keystore.py`. Par exemple, voici la fonction qui ajoute une clé au coffre à clés :

```
class key_container(ctypes.Structure):
    _fields_ = [
        ('key_id', ctypes.c_uint64),
        ('key_type', ctypes.c_int),
        ('key_len', ctypes.c_uint16),
        ('key_data', ctypes.c_byte*0x20),
    ]

def add_key(key_id, key_type, key_len, key_data):
    tee_key = key_container()
    tee_key.key_id = key_id
    tee_key.key_type = key_type
    tee_key.key_len = key_len
    tee_key.key_data = (ctypes.c_byte * 0x20).from_buffer_copy(key_data)

    fd = open('/dev/sstic', 'r')
    success = True

    try:
        fcntl.ioctl(fd, 0xC1105306, tee_key)
    except:
        success = False
```

```
finally:
    fd.close()
    return success
```

Le second ioctl inconnu, 0xC1105305 permet de récupérer une clé contenue dans le coffre à clé. J'ai récupéré toutes les clés du coffre, mais il ne contenait que les clés déjà trouvées, pas d'autre clé n'était contenue. J'abandonne donc cette piste.

Je décide maintenant de regarder le blob binaire contenu dans le fichier. Il fait un peu plus d'1Mo, ce qui pourrait signifier qu'il stocke des données intéressantes qui sont ensuite interprétées par le TEE. Après avoir extrait le blob, j'essaie de voir si certains éléments sont reconnaissables à l'intérieur.

J'utilise `strings`, puis `binwalk`²⁰ pour calculer l'entropie. Celle-ci est très élevée (0,98) signe que le contenu est pseudo-aléatoire (ce qui arrive s'il est compressé ou correctement chiffré). Je fais alors plusieurs hypothèses sur le format du blob par rapport notamment sa taille : $0x101010 = 0x100000 + 0x1000 + 0x10$.

Hyp : Les 16 octets supplémentaires du blob sont une clé ou un IV pour un AES.

J'ai essayé de déchiffrer le blob avec des modes classiques (ECB, CBC, CTR) en utilisant le premier bloc de l'algorithme comme clé, puis le dernier bloc, puis en les utilisant comme iv avec une clé déjà trouvée, mais rien n'a fonctionné. J'ai aussi tenté d'y reconnaître des données sérialisées comme des clés RSA, sans succès.

Je me tourne alors vers une méthode qui m'a déjà servi : mesurer le temps d'exécution. Ici, je cherche à évaluer le nombre de tours faits dans la seconde étape. Après avoir réimplémenté l'exécutable et ses différentes étapes en Python, je constate que cette seconde étape prend systématiquement 0x2496 tours de boucle pour s'exécuter. En faisant varier la valeur de l'argument, rien ne change : toujours 0x2496 tours.

Je cherche alors à faire varier le contenu du blob que j'envoie, pour déterminer si son contenu a un sens ou s'il est purement aléatoire. Pour ceci j'induis des inversions de bits dans le blob que j'envoie. Je tente d'inverser le premier bit, puis un bit au milieu du blob puis un bit à la fin. Dans le premier cas, le programme termine au bout d'un tour de boucle tandis que dans les deux autres cas, il termine en 0x2496 tours.

Je teste alors toutes les inversions de bits possibles sur les premiers blocs de 128 bits de données et j'observe l'évolution du nombre de tours qui sont accomplis. Dans les premiers blocs de données, une inversion de bits de traduit par l'un de ces trois effets :

- La boucle termine correctement en indiquant que l'argument n'est pas le bon.
- La boucle termine avec ce qui semble être une erreur.
- Le TEE panique, il faut arrêter de force qemu pour le relancer.

Dans les deux premiers cas il est possible de récupérer le nombre de tours de boucle qui ont été effectués et c'est suffisant (les cas de panique restent peu fréquents). Je peux observer alors la répartition du nombre de tours de boucle suivant la position du bit qui a été inversé :

- En inversant un bit dans le premier bloc, l'exécution termine en 1 tour (au minimum)
- En inversant un bit dans le second bloc, l'exécution termine en 6 tours (au minimum)
- En inversant un bit dans le troisième bloc, l'exécution termine en 11 tours (au minimum)
- ...

J'émet alors l'hypothèse suivante :

Hyp : Les données au début du blob sont chiffrées par bloc de 16 octets selon un procédé inconnu.

20. <https://github.com/ReFirmLabs/binwalk>

J'observe aussi qu'en inversant un bit après le 27e bloc, l'exécution termine toujours en 0x2496 tours de boucle. Je fais l'hypothèse suivante :

Hyp : Le début du blob est une sorte de code chiffré ; tout ce qui se trouve après le 27e bloc est de la donnée aléatoire.

Le script qui a permis de mesurer le nombre de tours de boucles en fonction du bit qui a été inversé se trouve en annexe.

Malgré cela, les informations que je parviens à récupérer en boîte noire sont plutôt maigres et je n'ai aucune idée de la suite des opérations. Comme qemu exécute le code du TEE, celui-ci doit bien se trouver quelque part. Je pars donc à la recherche du TEE afin de comprendre ce qu'il fait ²¹.

Au commencement, il y avait qemu

Bien amorcé

Lorsque j'évalue les possibilités pour retrouver le TEE, je pense au core dump ou à instrumenter le code de qemu pour retrouver le code exécuté. En discutant avec mes collègues, l'un d'eux me parle de quelque chose de beaucoup plus simple : lancer qemu en mode serveur gdb. Je fais l'hypothèse suivante :

Hyp : Le système n'implémente pas de mécanisme d'anti-débogage

Lorsque qemu est lancé avec le flag `-S`, il s'arrête au démarrage avant d'exécuter la première instruction. Lorsqu'il est lancé avec le flag `-s`, il lance un serveur gdb qui écoute à l'adresse `localhost:1234`. En combinant ces deux options il est possible de déboguer le système que qemu fait tourner.

```
# Lancement de qemu en mode serveur gdb
$ 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 -s -S
# Dans un autre terminal
$ aarch64-linux-gnu-gdb
(gdb) target remote localhost:1234
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000000000000000 in ?? ()
(gdb) p/x $pc
$1 = 0x0
(gdb) x/8i $pc
=> 0x0:  mov    x0, #0x830                      // #2096
    0x4:  movk   x0, #0x30c5, lsl #16
    0x8:  msr    sctlr_el3, x0
    0xc:  isb
    0x10: bl     0x385c
    0x14: cbz   x0, 0x1c
    0x18: br    x0
    0x1c: adr    x0, 0x4800
```

Avec ces commandes, je peux donc déboguer le processus d'amorçage du système. Le code exécuté en premier est le BIOS, contenu dans le fichier `rom.bin`. Le système lui-même est contenu dans le fichier

21. Lorsque je pars dans cette direction, je suis convaincu d'être en train de résoudre le challenge d'une manière 'non autorisée'. Il s'agissait en réalité de la bonne manière de procéder.

`flash.bin` mais celui-ci est chiffré. Néanmoins si `qemu` arrive à le déchiffrer alors je devrais aussi pouvoir y arriver d'une manière ou d'une autre.

Le fichier `flash.bin` dispose quand même d'un en-tête (reconnaissable car les données ne sont pas aléatoires). En affichant les premiers octets avec `xxd`²², voici le résultat.

```
00000000: 0100 64aa 7856 3412 0000 0000 0000 0000
00000010: 5ff9 ec0b 4d22 3e4d a544 c39d 81c7 3f0a
00000020: d800 0000 0000 0000 3094 0000 0000 0000
00000030: 0000 0000 0000 0000 47d4 086d 4cfe 9846
00000040: 9b95 2950 cbbd 5a00 0895 0000 0000 0000
00000050: a090 0000 0000 0000 0000 0000 0000 0000
00000060: 05d0 e189 53dc 1347 8d2b 500a 4b7a 3e38
00000070: a825 0100 0000 0000 7053 0000 0000 0000
00000080: 0000 0000 0000 0000 d6d0 eea7 fcea d54b
00000090: 9782 9934 f234 b6e4 1879 0100 0000 0000
000000a0: 2000 2202 0000 0000 0000 0000 0000 0000
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000
000000c0: 3879 2302 0000 0000 0000 0000 0000 0000
000000d0: 0000 0000 0000 0000 6431 5fc4 06cd ec49
000000e0: fcb7 25e0 2d8a 3df7 48eb a50b 410f 3d47
000000f0: 772f a330 6482 d503 3918 fc2f 5fcd 385f
....
```

Je charge donc le fichier `rom.bin` dans `ghidra`. Comme le fichier n'est pas dans un format exécutable reconnaissable, il n'arrive pas à deviner tout seul l'architecture. Il faut choisir manuellement l'architecture ciblée parmi la liste : `AArch64`, little endian.

En regardant les chaînes de caractère définies, je me rends compte que beaucoup de chaînes (notamment qui commencent par les mots "KEYSTORE") sont utilisées par la fonction `FUN_00000de8`. En regardant cette fonction, je me rends compte qu'elle cherche le coffre à clés. Pour rappel, pour déchiffrer le système lors de la première étape, le BIOS indiquait qu'il avait besoin de la clé AES stockée dans le coffre à clés.

À ce moment-là, je cherche un algorithme AES qui serait implémenté dans le BIOS. Une méthode pour le trouver est de chercher les occurrences de l'instruction `eor` (OU exclusif en ARM) car l'AES utilise cette opération à plusieurs endroits. Trois fonctions contiennent cette instruction :

- `FUN_000022b8`
- `FUN_0000286c`
- `FUN_00002c64`

Je commence par mettre un point d'arrêt sur la dernière pour voir les arguments qui lui sont passés. Le prototype de cette fonction, selon `ghidra` est :

```
void FUN_00002c64 (undefined8 uParm1, longlong lParm2, longlong lParm3,
    ulonglong uParm4, undefined8 uParm5, longlong lParm6)
```

Selon `gdb` :

```
uParm1 (x0) = "SSTIC{a947d6980ccf7b87cb8d7c246}"
lParm2 (x1) = 0xe00b010
lParm3 (x2) = 0xe00b000
uParm4 (x3) = 0x9440
uParm5 (x4) = 0x9440
```

22. <https://linux.die.net/man/1/xxd>

```
lParm6 (x5) = 0xe04fa70
```

L'argument "SSTIC{a947d6980ccf7b87cb8d7c246}" est la toute première clé trouvée. Il s'agit d'une chaîne de 32 caractères, j'ai donc à faire à un AES-256. De plus les arguments `uParm3` et `uParm4` pourraient être des longueurs de tampon. Enfin, les 32 premiers octets du tampon pointé par `x2` sont :

0xe00b000:	0x64	0x31	0x5f	0xc4	0x06	0xcd	0xec	0x49
0xe00b008:	0xfc	0xb7	0x25	0xe0	0x2d	0x8a	0x3d	0xf7
0xe00b010:	0x48	0xeb	0xa5	0x0b	0x41	0x0f	0x3d	0x47
0xe00b018:	0x77	0x2f	0xa3	0x30	0x64	0x82	0xd5	0x03

Je fais le rapprochement entre ces deux éléments et le début du fichier `flash.bin`. En effet, il est possible de voir les deux octets 3094, soit 0x9430 stocké en mode petit-boutiste, ce qui est quasiment la valeur stockée dans `x3` et `x4` (à 16 octets prêts). Aussi, je constate que `x2` pointe vers le début du blob chiffré, qui commence à l'adresse 0xd8 dans `flash.bin`.

Le code de la fonction décompilé par ghidra est :

```
void FUN_00002c64(undefined8 uParm1, longlong lParm2, longlong lParm3,
    ulonglong uParm4,
    undefined8 uParm5, longlong lParm6)
{
    longlong lVar1;
    ulonglong uVar2;
    undefined auStack256 [16];
    undefined auStack240 [240];

    uVar2 = uParm4 >> 4 & 0xffffffff;
    FUN_0000265c(auStack240, uParm1, 0x100);
    while ((int)uVar2 != 0) {
        FUN_00004174(auStack256, lParm2, 0x10);
        FUN_0000286c(auStack240, 0xe, lParm2, lParm3);
        lVar1 = 0;
        do {
            *(byte *)(lParm3 + lVar1) = *(byte *)(lParm3 + lVar1) ^ *(byte *)(
lParm6 + lVar1);
            lVar1 = lVar1 + 1;
        } while (lVar1 != 0x10);
        lParm2 = lParm2 + 0x10;
        FUN_00004174(lParm6, auStack256);
        lParm3 = lParm3 + 0x10;
        uVar2 = (ulonglong)((int)uVar2 - 1);
    }
    return;
}
```

La boucle `do...while` au milieu réalise le OU exclusif d'un bloc de 16 octets avec un autre, puis se décale de 16 octets dans chacun des tampons. Il pourrait donc s'agir d'un AES utilisé en mode CBC.

Je tente de déchiffrer la première partition avec ces paramètres :

- Algorithme : AES-256 en mode CBC
- Clé : "SSTIC{a947d6980ccf7b87cb8d7c246}"
- IV: 0x64315fc406cdec49fcb725e02d8a3df7 (le premier bloc de la partition chiffrée)
- Adresse de la partition : 0xe8 (le début des données de la partition chiffrée)
- Longueur de la partition : 0x9430

En affichant les derniers octets de la partition, je n'observe que des octets nuls : l'opération est un succès puisque si la clé ou le mode avaient été mauvais, le résultat aurait été des données aléatoires. Il faut maintenant que je sache où cette seconde partition est chargée en mémoire. Pour cela, j'ai continué le débogage avec gdb jusqu'à ce que le système exécute cette première partition : elle est chargée à l'adresse 0x0e00b000.

Au cours de ce processus, je constate que la fonction FUN_00002c64 n'est plus jamais appelée. A priori, le BIOS se contente donc de déchiffrer la première partition et de l'amorcer, mais ne déchiffre pas toute la mémoire flash. En réitérant l'analyse précédente pour essayer de voir comment sont déchiffrées les autres partitions, je retrouve la même fonction de déchiffrement, cette-fois ci à l'adresse 0x0e00d07c.

Je mets un point d'arrêt avec gdb pour voir avec quels arguments elle est appelée :

```
(gdb) dprintf *0x0e00d07c, "key: %s, length: 0x%x\n", (char*) $x0, $x3
Dprintf 3 at 0xe00d07c
(gdb) c
Continuing.
key: SSTIC{a947d6980ccf7b87cb8d7c246}, length: 0x90b0
key: SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}, length: 0x5380
key: SSTIC{a947d6980ccf7b87cb8d7c246}, length: 0x2220030
```

Vraisemblablement le TEE est la partition numéro 3, c'est la seule qui est déchiffrée avec la clé trouvée à l'étape précédente. Les autres sont déchiffrées avec la toute première clé trouvée.

Les partitions sont stockées les unes à la suite des autres dans le fichier **flash.bin**. De plus, l'IV est toujours le premier bloc de la partition. Le programme qui permet de déchiffrer toutes les partitions se trouve en annexe.

Je continue le débogage pour trouver les adresses de chargement des deux partitions suivantes : deuxième étage d'amorçage et TEE. Elles sont chargées respectivement aux adresses 0x0e030000 et 0x0e200000.

Une machine pas sympathique

Je cherche maintenant à savoir ce qui est fait lorsque l'instruction **smc** est exécutée. En utilisant gdb, je constate que chaque fois que cette instruction est exécutée le flot d'exécution passe à l'adresse 0x0e037404.

J'arrive aussi à mettre des points d'arrêt à l'intérieur du module noyau. Je peux donc voir à quels moments sont exécutées les instructions **smc** à l'intérieur du module noyau.

Contrairement à ce que j'imaginai initialement, l'instruction **smc** est exécutée plusieurs fois pour chaque exécution par le module noyau. En affichant l'identifiant de l'appel effectué, voici un début de trace généré avec gdb :

```
(gdb) dprintf *0xffff000008590198, "Sending blob\n"
(gdb) dprintf *0xffff000008590518, "Sending key part\n"
(gdb) dprintf *0xffff000008590414, "Loop call 1\n"
(gdb) dprintf *0xffff00000859022c, "Loop call 2\n"
(gdb) dprintf *0x0e037404, "SMC call id: %x\n", $x0
(gdb) c
Sending blob
```

```

SMC call id: 83010004
Sending key part
SMC call id: f2005003
SMC call id: f2001000
SMC call id: 83010002
SMC call id: f2005003
Sending key part
SMC call id: f2005003
SMC call id: f2001000
SMC call id: 83010002
SMC call id: f2005003
Sending key part
SMC call id: f2005003
SMC call id: f2001000
SMC call id: 83010002
SMC call id: f2005003
Sending key part
SMC call id: f2005003
SMC call id: f2001000
SMC call id: 83010003
SMC call id: 83010002
SMC call id: f2005003
Sending key part
SMC call id: f2005003
SMC call id: f2001000
SMC call id: 83010003
SMC call id: 83010002
SMC call id: f2005003
Sending key part
SMC call id: f2005003
SMC call id: f2001000
SMC call id: 83010003
SMC call id: 83010002
SMC call id: f2005003
Sending key part
SMC call id: f2005003
SMC call id: f2001000
SMC call id: 83010002
SMC call id: f2005003

```

Ce qui me semble plutôt incompréhensible pour le moment c'est que pour deux opérations identiques (envoyer quatre octets de clé), les appels effectués ne sont pas les mêmes : dans quatre cas un appel supplémentaire est fait (avec l'identifiant 0x83010003).

Je continue donc l'analyse statique et dynamique en parallèle, pour essayer d'identifier les fonctions qui réalisent des opérations intéressantes. Petit à petit, j'identifie une fonction qui semble pertinente, dont le comportement dépend de l'identifiant de l'appel SMC : 0x0e031034. Cette seule fonction fait un **switch** sur l'identifiant de l'instruction **smc** est occupée beaucoup de place à l'intérieur de la partition (un peu

plus de 10%). Elle a la particularité de beaucoup utiliser des instructions ARM NEON ²³, en incluant des opérations cryptographiques (`aesd`, `aese`).

J'identifie aussi deux fonctions dans la partition correspondant au TEE, qui utilisent aussi des instructions ARM NEON : la fonction `0x0e2005a4` fait elle aussi un `switch` sur quelque chose et utilise quelques instructions ARM NEON, tandis que la fonction à l'adresse `0x0e200e84` n'utilise quasiment que des instructions ARM NEON. Cette seconde fonction utilise aussi des instructions qui réalisent des opérations cryptographiques, à savoir `sm4ekey` et `sm4e`. Ces instructions sont respectivement l'algorithme de génération de clé de rondes de l'algorithme SM4 ²⁴ ainsi que les premiers tours de l'algorithme SM4.

Bien que je ne comprenne pas encore ce que font ces fonctions ni comment elles interagissent, je les note dans un coin de tête. Je cherche maintenant à avancer sur deux points critiques :

- Où est stocké le blob de données envoyé.
- Où est stockée la clé envoyée.

En passant du temps à déboguer le système, je constate déjà que le blob est copié à l'adresse `0x413000`. Pour la clé, plusieurs choses se passent :

- Seize octets de clé sont traités spécifiquement par la fonction `0x0e031034`
- Elle est envoyée par bouts de 4 octets à la fonction `0x0e200e84`

Pour rappel, la clé est envoyée par 8 paquets de quatre octets chacun. Les paquets numéro 3, 4, 5 et 6 (en numérotant de 0 à 7) font l'objet d'un traitement spécifique par la fonction à l'adresse `0x0e031034`. Ces quatre octets sont stockés dans des registres particuliers :

```
Envoi de la clé 0
  x1111111122222222333333334444444455555555666666667777777788888888

--
fpexc32_el2 = 0x44444444
dacr32_el2 = 0x55555555
ifsr32_el2 = 0x66666666
sder32_el3 = 0x77777777
```

Après leur initialisation, ces registres ne sont pas modifiés durant l'exécution. Ils sont lus à certains moments (par la fonction à l'adresse `0x0e031034`), mais pas plus.

Pour ce qui est du stockage de la clé en entier, elle passe par la fonction `0x0e200e84`. Là, elle est stockée chiffrée à l'adresse `0x513020`. Ce chiffrement est fait par quatre tours de l'algorithme SM4, avec une clé en dur dans le système, stockée à l'adresse `0x0e204048`. Je note aussi que le procédé de chiffrement est diversifié par l'adresse de stockage des données (qui est statique). Chaque bloc de 16 octets sera donc chiffré de manière différente suivant son adresse en mémoire.

Maintenant que ces éléments sont connus, même si j'ai encore une vision très incomplète des choses, je commence à m'intéresser à ce qui est fait par les deux appels au TEE à chaque tour de boucle du binaire de départ.

Ayant assisté au SSTIC, édition 2017 je me souviens très rapidement de la présentation "Désobfuscation binaire : Reconstruction de fonctions virtualisées" ²⁵. C'était la première fois que j'entendais parler du concept d'embarquer des machines virtuelles dans un binaire. Notamment, dans mes souvenirs l'une des caractéristiques d'une machine virtuelle embarquée dans un binaire est un grand `switch...case` qui choisit le traitement à effectuer pour une instruction virtuelle décodée. Il s'agit aussi du type d'implémentation faite de la machine virtuelle DWARF par la `libgcc`. J'ai pu reconnaître ici deux fonctions qui pourraient correspondre à cette description. Je pose donc l'hypothèse suivante :

23. <https://developer.arm.com/docs/ddi0596/b/simd-and-floating-point-instructions-alphabetic-order>

24. [https://en.wikipedia.org/wiki/SM4_\(cipher\)](https://en.wikipedia.org/wiki/SM4_(cipher))

25. https://www.sstic.org/2017/presentation/desobfuscation_binaire_reconstruction_de_fonctions_virtualisees/

Hyp : Je suis en train d'essayer de comprendre une machine virtuelle embarquée dans l'environnement d'exécution sécurisé.

Je continue à déboguer mon binaire, cette fois-ci j'essaye de corréler les appels faits aux fonctions 0x0e031034, 0x0e2005a4 avec les deux instructions `smc` exécutées à chaque tour de boucle. J'utilise donc gdb pour tracer les différents appels.

- Dans le premier cas, la première instruction `smc` exécutée se traduit systématiquement par l'appel de la fonction à l'adresse 0x0e031034 avec l'identifiant 0x83010001. Néanmoins, la deuxième instruction exécutée se traduit par plusieurs appels à cette fonction, avec des identifiants qui changent constamment. Les valeurs 0x83010001 et 0x83010002 reviennent régulièrement mais je ne distingue de motif clair.
- Dans le second cas, la première instruction `smc` exécutée ne provoque pas d'appel à la fonction à l'adresse 0x0e2005a4, en revanche la seconde instruction provoque systématiquement un seul et unique appel à cette fonction.

Par ailleurs, en observant rapidement cette fonction je constate qu'elle exécute plusieurs instructions `smc` avec un identifiant commençant par 0x8301. Ce préfixe se traduit par l'appel de la fonction 0x0e031034 avec l'identifiant donné. Pour chaque tour de boucle, j'en déduis le fonctionnement suivant :

1. La première instruction `smc` est exécutée avec l'identifiant 0xf2005001;
2. La fonction à l'adresse 0x0e031034 est appelée avec l'identifiant 0x83010001;
3. La seconde instruction `smc` est exécutée avec l'identifiant 0xf2005002;
4. La fonction à l'adresse 0x0e2005a4 est appelée avec un identifiant sur trois octets que je ne reconnais pas;
5. Suivant la valeur de cet identifiant, une ou plusieurs instructions `smc` sont exécutées avec un identifiant de format 0x8301XXXX;
6. Pour chaque instruction `smc` exécutée, la fonction à l'adresse 0x0e031034 est appelée avec l'identifiant 0x8301XXXX.

Maintenant que je sais très globalement dans quel ordre les opérations sont effectuées et les opérations appelées, j'essaye de voir un peu plus ce qu'il en est.

Toujours plus de chiffrement

Deux motifs reviennent souvent à l'intérieur de la fonction à l'adresse 0x0e031034 :

```
; Motif 1
; Chargement de 16 octets dans le registre v1
mrs      x0,fpexc32_el2
mov      v1.S[0x0],w0
mrs      x0,dacr32_el2
mov      v1.S[0x1],w0
mrs      x0,ifsr32_el2
mov      v1.S[0x2],w0
mrs      x0,sder32_el3
mov      v1.S[0x3],w0
; Chargement de 16 octets depuis une adresse vers v0
ld1      {v0.4S},{x22}
; Exécution d'un tour de l'algorithme de chiffrement AES-128 du bloc
; dans v0 avec la clé dans v1
aese     v0.16B,v1.16B
eor      v0.16B,v0.16B,v1.16B
st1      {v0.4S},{x20=>local_100}

; Motif 2
```

```

; Chargement de 16 octets dans le registre v1
mrs      x0,fpexc32_el2
mov      v1.S[0x0],w0
mrs      x0,dacr32_el2
mov      v1.S[0x1],w0
mrs      x0,ifsr32_el2
mov      v1.S[0x2],w0
mrs      x0,sder32_el3
mov      v1.S[0x3],w0
; Chargement de 16 octets depuis une adresse vers v0
ld1      {v0.4S},[x20=>local_100]
; Exécution d'un tour de l'algorithme de déchiffrement AES-128 du bloc
dans v0 avec la clé dans v1
aesd     v0.16B,v1.16B
eor      v0.16B,v0.16B,v1.16B
st1      {v0.4S},[x19]

```

Or, je sais que les quatre registres `fpexc32_el2`, `dacr32_el2`, `ifsr32_el2` et `sder32_el3` contiennent un bout (16 octets sur 32) de la clé donnée en entrée au binaire de départ. Ce fragment est utilisé comme clé pour chiffrer et déchiffrer des données à certaines adresses en utilisant l'algorithme AES-128²⁶. En effet, les instructions `aese` et `aesd` définies par ARM réalisent les opérations `AddRoundKey`, `SubBytes` et `ShiftRows` (ou les opérations inverses dans le cas du déchiffrement) pour un tour d'AES.

En regardant l'emplacement des données qui sont chiffrées et déchiffrées, elles se trouvent systématiquement entre les adresses `0x0e04a5d8` et `0x0e04a6c8`. Je peux donc tracer l'évolution de cette section à mesure que la boucle avance puisque je sais quelle clé et quel algorithme a été utilisé.

Avant de faire cela néanmoins, je cherche encore à savoir où est utilisé le blob de données envoyé. Je sais qu'il est lu quelque part et que sa valeur influe sur le déroulé de l'algorithme, puisque j'avais fait des tests par inversion de bits en ce sens. Comme pour l'étape 3, je positionne un point d'arrêt en lecture à l'adresse `0x413000` avec la commande `rwat`. J'ai gardé les points d'arrêt posés précédemment, ce qui me permet d'estimer quand est-ce que ce blob est lu.

Je constate que le bloc est lu par la fonction à l'adresse `0x0e200e84`, que 16 octets sont chargés et déchiffré avec l'algorithme SM4 (en utilisant là encore les instructions cryptographiques d'ARM NEON). Cette opération intervient entre les deux instructions `smc`. Le code de déchiffrement est le suivant :

```

; Chargement de l'adresse de la donnée à déchiffrer dans le registre v0
0e200ee8 ld1      {v0.4S},[x23=>local_30]
0e200eec rev32     v0.16B,v0.16B
; Chargement d'une constante de l'algorithme SM4 dans le registre v1
0e200ef0 ld1      {v1.4S},[x22=>DAT_0e204058]
; Chargement de la clé à l'adresse 0x0e204048 dans le registre v2
0e200ef4 mov      v2.S[0x0],w19
0e200ef8 mov      v2.S[0x1],w19
0e200efc mov      v2.S[0x2],w19
0e200f00 mov      v2.S[0x3],w19
; Diversification de la constante par l'adresse de chargement
0e200f04 eor      v1.16B,v1.16B,v2.16B
; Algorithme de génération des quatre premières clés de ronde, qui sont
mises dans le registre v1

```

26. Parler de chiffrement est un peu abusif puisqu'il s'agit d'un tour partiel d'AES (l'opération `MixColumns` n'est pas effectuée). Il s'agit plus d'une forme d'obfuscation que d'un chiffrement fort. J'ai choisi de conserver l'usage des termes 'chiffrement' et 'déchiffrement' néanmoins. De plus, comme la clé et les paramètres sont connus, l'usage d'un AES complet et non d'un seul tour aurait juste rallongé le code.

```

0e200f08  sm4ekey    v0.4S,v0.4S,v1.4S
0e200f0c  mov        v1.S[0x0],v0.S[0x3]
0e200f10  mov        v1.S[0x1],v0.S[0x2]
0e200f14  mov        v1.S[0x2],v0.S[0x1]
0e200f18  mov        v1.S[0x3],v0.S[0x0]
; Chargement du bloc de données à déchiffrer
0e200f1c  ld1        {v0.4S},[x0]
0e200f20  rev32      v0.16B,v0.16B
; Déchiffrement du bloc de données
0e200f24  sm4e       v0.4S,v1.4S
0e200f28  rev32      v0.16B,v0.16B
0e200f2c  mov        v1.S[0x0],v0.S[0x3]
0e200f30  mov        v1.S[0x1],v0.S[0x2]
0e200f34  mov        v1.S[0x2],v0.S[0x1]
0e200f38  mov        v1.S[0x3],v0.S[0x0]
0e200f3c  st1        {v1.4S},[x20=>local_20]

```

Je constate qu'après l'instruction à l'adresse 0x0e200f28, le registre v0 contient le bloc déchiffré. Je vais directement voir le résultat avec un point d'arrêt positionné à l'adresse 0x0e200f2c. Le premier bloc se déchiffre en 0x10404f202d10400f004d200010000d10. Je remarque aussi que le déchiffrement d'un bloc intervient de manière systématique entre les deux instructions `smc` exécutées par le module noyau, j'essaye donc de voir s'il n'y a pas un lien entre le bloc qui est déchiffré et l'identifiant de trois octets qui est utilisé par la fonction 0x0e2005a4.

Avec gdb et des points d'arrêt bien positionné, j'obtiens le début de trace suivante :

```

SMC call 1
Result 0x10404f202d10400f004d200010000d10
SMC call 2
id: 0x0d0010

SMC call 1
Result 0x10404f202d10400f004d200010000d10
SMC call 2
id: 0x4d0010

SMC call 1
Result 0x10404f202d10400f004d200010000d10
SMC call 2
id: 0x2d0020

SMC call 1
Result 0x10404f202d10400f004d200010000d10
SMC call 2
id: 0x0f4010

SMC call 1
Result 0x10404f202d10400f004d200010000d10
SMC call 2
id: 0x4f4010

```

L'identifiant ressemble assez fortement au blob déchiffré, même si l'ordre des octets n'a pas l'air d'être correct. En essayant plusieurs configurations, je tombe sur la valeur de premier bloc suivante : 0x10000d10004d20002d10400f10404f20. Les cinq premiers identifiants utilisés pour appeler la fonction 0x0e2005a4 sont quant à eux : 0x0d0010, 0x4d0010, 0x2d0020, 0x0f4010, 0x4f4010. Il s'agit des 15 premiers octets de ce bloc, lus trois par trois en mode petit-boutiste.

Avec ces informations, je fais l'hypothèse suivante :

Hyp : La première instruction smc exécutée dans un tour de boucle sert à déchiffrer l'instruction virtuelle courante, la seconde sert à décoder et exécuter cette instruction.

Si je récapitule les informations acquises jusqu'à présent, en utilisant les hypothèses précédentes :

- Le blob est copié à l'adresse 0x413000, il est déchiffré bloc par bloc par le TEE avec une clé en dur selon l'algorithme SM4
- La clé donnée en argument est chiffrée puis copiée à l'adresse 0x513020 (soit juste après le blob)
- Une partie de la clé est utilisée par la fonction à l'adresse 0x0e031034 pour chiffrer et déchiffrer successivement 16 blocs de 16 octets (registres de la machine virtuelle ?)
- Le TEE exécute une VM, dont chaque instruction est exécutée par la fonction à l'adresse 0x0e2005a4
- Chaque instruction se traduit par un ou plusieurs appels à la fonction à l'adresse 0x0e031034
- Le nombre de tours de boucle faits par le binaire de départ correspond au nombre d'instructions virtuelles exécutées
- Une instruction virtuelle prend trois octets.

On a donc un algorithme qui est exécuté par une machine virtuelle embarquée dans le TEE. Je peux tracer l'algorithme avec gdb, notamment récupérer toutes les instructions exécutées, ce que je fais. Le script en question se trouve en annexe.

Ingénierie à reculons et en musique

Son et lumières

Une fois que j'ai récupéré la trace des instructions virtuelles exécutées, j'essaye de trouver leur format. En étudiant la fonction 0x0e02005a4 je retrouve le format suivant :

```
opcode (4 bits) || flags (2 bits) || arg1 (4 bits) || arg2 (4 bits) ||
arg3 (10 bits)
```

Je suis donc face à un jeu d'instructions RISC sur trois octets. À noter aussi que le troisième argument semble assez peu utilisé. Maintenant que je sais globalement comment fonctionne la machine virtuelle, je cherche à comprendre l'algorithme qu'elle exécute. Quand j'en arrive à ce point, je cherche à limiter les heures passées à utiliser ghidra et gdb, je cherche donc à récupérer le maximum d'informations sur l'algorithme sans avoir à utiliser ces outils.

Essentiellement, mon idée est de faire exactement l'inverse de l'étape précédente. Dans l'étape précédente, j'ai fait la rétro-ingénierie du code DWARF puis j'ai retrouvé le flot de contrôle puis la structure de l'algorithme avec ses boucles, ses itérations etc. . . Ici, je souhaite retrouver la structure de l'algorithme à partir de la trace uniquement sans effectuer de rétro-ingénierie : je veux retrouver la structure sans avoir (pour le moment) à comprendre ce qu'il se passe.

L'idée de base est de changer la représentation de la trace. Plutôt que d'avoir une succession de quelques milliers d'instructions hexadécimales, je cherche une représentation qui soit pour moi plus facile à appréhender. Ma première idée est de transformer la trace en une image : si chaque instruction est encodée en une série de pixels, peut-être que l'image résultante laissera apparaître une structure évidente.

Je choisis de laisser de côté le troisième argument de chaque instruction qui est assez peu utilisé : l’opcode sera représenté par trois pixels, les flags par un pixel, arg1 par 3 pixels et arg2 par trois pixels pour un total de dix pixels (horizontalement) par instruction. Plutôt qu’avoir une longue colonne ou une longue ligne, je fixe un nombre maximum d’instructions par colonne que je fais varier.

Reste à choisir la palette de couleurs, sachant qu’il m’en faut 16 différentes. Je cherche à avoir une palette qui présente un maximum de contraste entre chaque couleur afin de pouvoir aisément discriminer les éventuels motifs. Après un tour du côté des communautés de CS: GO ²⁷ et de l’accessibilité aux malvoyants qui sont deux communautés pour lesquelles je sais que la notion de “palette de couleur contrastée” a une importance forte, je trouve une palette de 22 couleurs présentée par Kenneth Kelly dans “Twenty-Two Colors of Maximum Contrast” (1965).

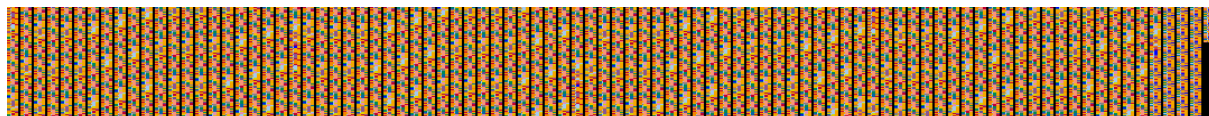


FIGURE 15 – Visualisation de la trace, 100 instructions par ligne

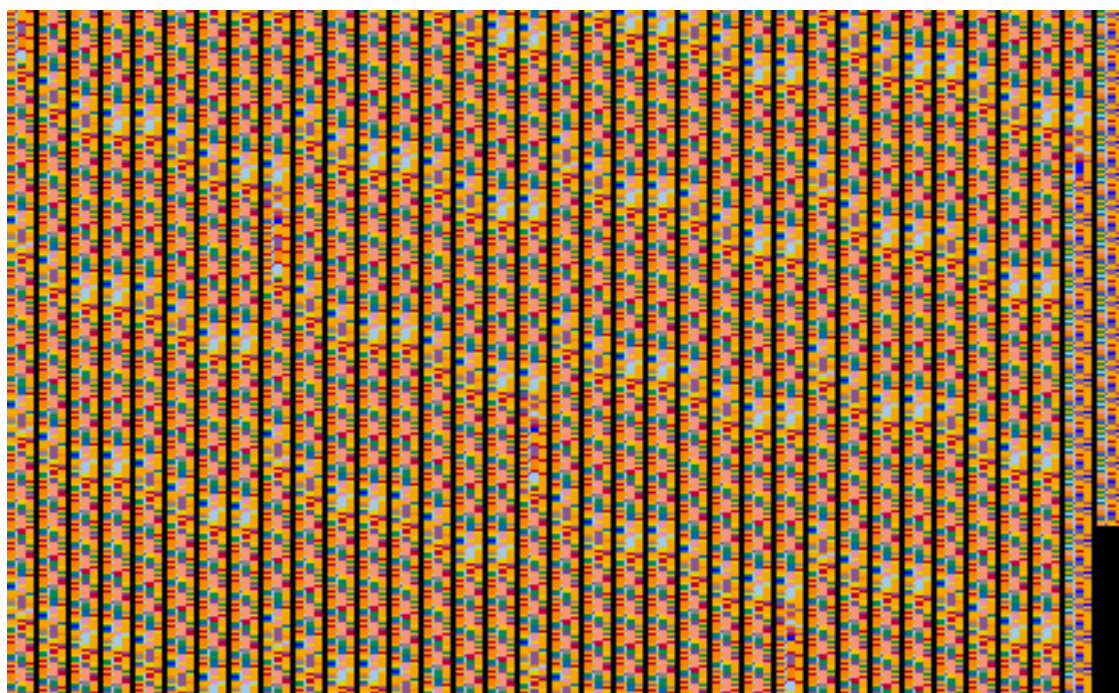


FIGURE 16 – Visualisation de la trace, 257 instructions par ligne

Quelques exemples de visualisations générées se trouvent sur les figure 15, 16 et 17. Le résultat est très encourageant puisqu’il est possible de distinguer assez nettement des motifs répétés qui pourraient être des boucles.

Toujours en cherchant à modifier la représentation de la trace, le second type d’analyse effectué est plus complexe à transmettre dans une solution écrite. Je fais de la musique depuis l’école primaire et pour moi, il est bien plus naturel de trouver la structure d’un morceau (lu et entendu) que de trouver la structure de code assembleur.

J’ai donc tenté de transformer la trace d’exécution en une représentation auditive afin de pouvoir l’écouter. J’ai conservé la même approche que précédemment, à savoir transformer l’opcode, les flags, arg1 et arg2 et laisser arg3 de côté.

27. Counter Strike: Global Offensive

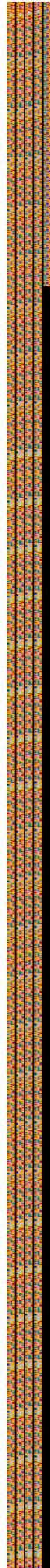


FIGURE 17 – Visualisation de la trace, 0x858 instructions par ligne

Générer un son ‘pur’ (mono-fréquentiel) de manière informatique n’est pas très compliqué avec quelques bases. Typiquement les cartes sons permettent de lire des sons échantillonnés à 44100Hz. De plus, Audacity²⁸ permet d’importer des fichiers bruts si on lui précise le format des échantillons qu’il contient (taux d’échantillonnage, représentation des échantillons, nombre de canaux, ...). Le fragment de code suivant en Golang génère un La (440Hz) qui dure deux secondes et le sauvegarde dans un fichier. Ce fichier n’a pas de format spécifique (e.g. ce n’est pas un .wav) donc il faut spécifier à Audacity que le taux d’échantillonnage est 44100Hz, qu’il n’y a qu’un canal et que le format des échantillons est “Signed 16-bit PCM” en mode gros-boutiste.

```
import (
    "math"
    "os"
)

const (
    sampleRate = 44100
    freq = 440
    duration = 2
)

func main() {
    buf := make([]byte, 0)

    _ = os.Remove("test.raw")
    fp, _ := os.Create("test.raw")

    nbSamples := duration * sampleRate

    for i := 0; i < nbSamples; i++ {
        dt := float64(i) / float64(sampleRate)
        sample := int16(math.Sin(dt*2*math.Pi*freq) * 32767)
        buf = append(buf, byte(sample>>8), byte(sample & 0xff))
    }

    fp.Write(buf)
    fp.Close()
}
```

Deux modes de représentations viennent logiquement en tête :

- Représenter chaque instruction par une succession de sons (mode ‘arpège’)
- Représenter chaque instruction par une superposition de sons (mode ‘accord’)

Il est aussi possible de faire varier la vitesse (le tempo) suivant les essais. Chaque valeur sur quatre bits doit être représentée par une note et je prend simplement les notes de la gamme de Do majeur (en commençant par le Do 4 à 261,63Hz). Je choisis d’utiliser le mode ‘arpège’ et cette fois-ci (arbitrairement) d’accorder un peu plus d’importance à l’opcode qu’aux arguments, ce qui se traduit par un rythme croche pointée (opcode) - double croche (flags) - croche (arg1) - croche (arg2).

Comme il est difficile de transmettre du son par écrit, les premières instructions sont encodées par la partition présentée en figure 18. La partition a été générée grâce à Lilypond²⁹ : un programme Go m’a permis d’écrire un fichier texte en .ly, que Lilypond a compilé en une partition au format pdf. Les

28. <https://www.audacityteam.org/>

29. <http://lilypond.org/>



FIGURE 18 – Visualisation des 30 premières instructions

programmes qui permettent de générer les fichiers de sons ainsi que les partitions à partir de la trace sont donnés en annexe. Je ne me suis pas servi de partitions telle que celle de la figure 18 au cours dans la résolution, son but premier est de servir au lecteur de cette solution.

L'objectif initial était de réussir à reconnaître la structure de l'algorithme à partir de ces deux modes de représentation. J'ai donc écouté la représentation sonore toute en parcourant la 'partition' formée par l'image générée au début. J'ai constaté qu'il était plus simple de voir la structure globale de l'algorithme en utilisant l'image (boucles externes), alors qu'il était plus simple d'entendre la structure de petits fragments de l'algorithme avec le son. Le son produit est très rapide (avec le tempo choisi) et vraiment pas plaisant, mais permet quand même de distinguer les motifs répétés, ce qui était l'objectif initial.

La combinaison de ces deux représentations m'a permis de déduire la structure d'algorithme suivante :

```
for i := 0; i < 4; i++ {
  # ...
  for j := 0; j < 8; j++ {
    for k := 0; k < 4; k++ {
      # ...
    }
    # ...
  }
  for j := 0; j < 8; j++ {
    for k := 0; k < 4; k++ {
      # ...
    }
    # ...
  }
}

# Ces deux boucles de taille 8 semblent être les même que celles d'
avant
for j := 0; j < 8; j++ {
  for k := 0; k < 4; k++ {
    # ...
  }
  # ...
}
for j := 0; j < 8; j++ {
  for k := 0; k < 4; k++ {
    # ...
  }
}
```

```

        # ...
    }
}
for i := 0; i < 32; i++ {
    # ...
}

```

Par ailleurs, les boucles les plus imbriquées (avec l'indice k) semblent être systématiquement les même. Je remarque aussi la toute dernière boucle, de longueur 32. L'argument passé en paramètre fait 32 octets, il est envisageable qu'il soit dérivé en une autre valeur de 32 octets, puis comparé caractère par caractère à une valeur de référence.

Hyp : La dernière boucle sert à comparer une valeur dérivée de l'argument avec une valeur de référence.

Les 32 dernières phrases

Je vais donc voir vers la fin de la trace, en sachant que je cherche une suite de quelques (environ 10) instructions qui se répète 32 fois. Je choisis de tracer une exécution complète de l'algorithme, cette fois en récupérant l'instruction exécutée, l'état de ce que je pense être les registres et les opérations de déchiffrement SM4 faites.

Je code un petit programme qui va déchiffrer l'état des supposés registres en utilisant la clé dérivée de l'entrée du programme (les scripts sont encore disponibles en annexe). Une fois cela fait, je constate que les registres contiennent au plus quatre octets et ne sont pas alignés. Une fois remises en place, je vois que la valeur du dernier registre s'incrémente de 3 en 3 : il doit s'agir du pointeur d'instruction virtuel. Quand aux autres, ils semblent contenir des valeurs intermédiaires, ce qui confirme que les 16 blocs de 16 octets chiffrés avec une partie de la clé d'entrée sont bien les registres de la machine virtuelle. À l'exception du dernier (que j'ai nommé 'pc' pour 'program counter'), je les nomme de $r0$ à $r14$.

En observant les états successifs des registres, je retrouve une valeur que je suppose être la valeur de référence. Cette valeur est la chaîne "pr.a.rfg.cnf.fv.snpvyr@ffgvp.bet", qui se décode avec un rot13 en "ce.n.est.pas.si.facile@sstic.org".

Je reprends cette fois-ci le débogage et la rétro-ingénierie pour continuer le challenge. Notamment, j'utilise gdb pour changer la valeur dérivée de mon argument et la remplacer par "pr.a.rfg.cnf.fv.snpvyr@ffgvp.bet". En faisant cela, le programme se termine en affichant "Win".

Cela confirme que les 32 opérations de la fin sont effectivement des comparaisons caractère par caractère et que cette chaîne de caractères est effectivement la valeur attendue qu'il faut produire.

Après avoir essayé un certain temps de retrouver l'algorithme en boîte noire en comparant les valeurs dérivées (récupérées en traçant l'exécution avec gdb) et les entrées, j'en arrive à la conclusion que chaque bloc de 8 octets de l'entrée est dérivé séparément. En envoyant quatre fois le même bloc, je devrais donc avoir un résultat composé de quatre fois la même valeur en sortie. Cela se produit rarement, même si j'observe que le nombre de valeurs produites est restreint.

En traçant l'algorithme deux fois, lorsque je lance deux fois le binaire avec le même argument, j'observe deux valeurs dérivées légèrement différentes. Souvent il n'y a qu'un bloc de 8 octets de différent, mais c'est plus que suffisant pour me perturber puisque je suis face à un algorithme non-déterministe, qui effectue une vérification par rapport à une valeur complètement déterminée.

Je ressors donc ghidra en plus de gdb pour aller voir ce que fait cet algorithme.

Retrouver les opérations perdues

Je connais la structure de l'algorithme que je cherche (ou une structure approchante) et je dispose de traces d'exécutions qui mettent en évidence l'effet de chaque instruction sur les registres de la machine

virtuelle. Cela me permet de simplifier d'autant le travail de rétro-ingénierie concret sur le code de la machine virtuelle.

Je retrouve d'abord une opération qui s'apparente à un `mov`, puis des opérations de rotation `lsl`, `lsr`, et ainsi de suite. Par ailleurs, je comprends un peu mieux toutes les instructions `smc` différentes exécutées. La fonction à l'adresse `0x0e2005a4` sert à implémenter chacune des instructions virtuelles. La fonction à l'adresse `0x0e031034` sert à implémenter des micro-instructions virtuelles qui sont appelées par la fonction à l'adresse `0x0e2005a4` via l'instruction `smc`.

Par exemple, pour implémenter l'addition de deux registres (e.g. `r0` et `r1`), les micro-instructions virtuelles suivantes sont utilisées :

1. Appel `smc` avec l'identifiant `0x83010001` et l'argument 0 (`r0`).
2. Appel `smc` avec l'identifiant `0x83010001` et l'argument 1 (`r0`).
3. Appel `smc` avec l'identifiant `0x83010002`, l'argument 0 (`r0`) et le résultat de l'addition.
4. Appel `smc` avec l'identifiant `0x83010001` et l'argument `0xf` (`pc`).
5. Appel `smc` avec l'identifiant `0x83010002`, l'argument `0xf` (`pc`) et le résultat de `pc + 3`.

Les deux micro-instructions les plus couramment utilisées (`0x83010001` et `0x83010002`) servent respectivement à lire la valeur d'un registre en le déchiffrant à la volée et à écrire une valeur dans un registre en le chiffrant à la volée.

Ceci me permet d'écrire un désassembleur pour les instructions virtuelles afin d'avoir une représentation plus simple de l'algorithme. Au final, les instructions que j'arrive à reconnaître sont :

opcode	mnémonique
0x0	mov
0x1	dec
0x2	add
0x3	sub
0x4	lsl
0x5	lsr
0x6	xor
0x7	and
0x8	jmp
0x9	jnz
0xa	quit
0xb	swap
0xc	str key
0xd	???
0xe	???

Pour récupérer l'entièreté du programme, je me suis contenté de tracer les appels à la fonction de déchiffrement du blob. Ceci m'a permis notamment de retrouver les premiers blocs du blob, qui contiennent le code du programme. Ce code est très court, puisqu'il fait 448 octets.

Parmi les opérations non standards, l'instruction `quit` n'est appelée qu'une fois : à la toute fin du programme. Elle termine l'exécution du programme. L'instruction `swap` permet d'échanger les deux octets de poids faible d'un registre : si le registre `r0` contient `0x1234`, après l'instruction `swap r0` il contiendra `0x3412`. L'opération 'str key' est utilisée pour chiffrer et stocker la valeur de référence à l'adresse `0x513040`. Cette valeur de référence est en réalité contenue dans la fonction `0x0e2005a4` par fragments de quatre octets. Enfin, je n'ai pas compris ce que faisaient exactement les instructions correspondant aux opcodes `0xd` et `0xe`. La première est exécutée 4 fois dans l'algorithme, tandis que la deuxième est un saut conditionnel dont je n'ai pas retrouvé la condition. Les flags servent à indiquer le type des arguments, à savoir si un registre contient une valeur directe ou une adresse vers une valeur.

Je réimplémente l'algorithme en Golang et ce faisant je trouve à quoi sert le reste du blob : il est utilisé

comme table de substitution. Il est déchiffré à la volée dès que nécessaire, toujours selon l'algorithme SM4 avec la clé identifiée. Ceci me permet de réaliser aussi cette opération de déchiffrement, en vérifiant que mes résultats sont cohérents avec ce que j'observe dans les traces.

En effectuant la rétro-ingénierie de l'algorithme, je constate que la structure que j'avais déjà identifiée est quasiment identique. La structure réelle est :

```
for i := 0; i < 4; i++ {
    for j = 0x1f; j >= 0; j-- {
        subst1()
        subst2()
        subst3()
        subst4()
        if j&0x08 == 0 {
            do_smth1()
        } else {
            do_smth2()
        }
    }
}
for i := 0; i < 32; i++ {
    verif_chr()
}
```

La boucle imbriquée change légèrement de moyen de recombinaison les résultats suivant la valeur de j . Elle va aller 8 fois dans une branche, puis 8 fois dans l'autre et recommencer. Ceci explique pourquoi j'avais compté 4 boucles de longueur 8 chacune initialement. Quand aux opérations de substitutions, elles sont quasiment identiques les unes aux autres, d'où l'impression qu'il s'agissait d'une unique boucle de longueur 4.

L'algorithme fonctionne en découpant l'entrée en 4 blocs de 8 octets et les traite indépendamment (ce qui avait déjà été déterminé). Ensuite, chaque bloc de 8 octets est découpé en 4 blocs de 2 octets. La deuxième boucle va modifier cet état pour finalement sortir la valeur dérivée finale. Chaque tour de boucle interne prend donc en entrée un état de quatre fois deux octets et sort un autre état de quatre fois deux octets.

Les modifications sont faites par les fonctions de substitution (nommées ici subst1 à subst4). Ensuite, la recombinaison de résultats se fait suivant la branche suivie (qui dépend de l'indice d'itération courant). Les deux recombinaisons ne sont pas exactement les mêmes, mais cela change peu de choses au final.

Pour réussir à inverser l'algorithme, il faut se rendre compte que le passage d'un état à un autre ne dépend que de deux octets de l'état initial (stockés dans le registre r1) et de la valeur de l'indice d'itération courant. Pour retrouver la valeur de r1 de l'état initial, il y a une valeur de vérification dans l'état final : r0 contient deux octets qui ne dépendent que de r1 et du numéro de tour. Il est donc possible de faire une recherche exhaustive sur r1 pour retrouver sa valeur. Ensuite, il est possible d'effectuer la combinaison inverse de résultats en ayant connaissance du r1 de départ pour retrouver l'état initial.

Néanmoins, un problème subsiste. Dans la manière de recombinaison les résultats, un second saut conditionnel est effectué, à la fois dans l'opération do_smth1 et do_smth2. Ce saut conditionnel a un opcode de 0xe et il s'agit d'une des deux instructions que je n'ai pas réussi à comprendre correctement. Par ailleurs, pour avoir généré beaucoup de traces en essayant d'inverser cette algorithme, j'ai constaté que pour un même bloc de 8 octets d'entrée, ce saut conditionnel était des fois effectué, et des fois non. C'est cette instruction qui est à l'origine du comportement non déterministe que j'ai pu observer.

Ultime tromperie

Afin d'avancer quand même et de commencer à inverser l'algorithme, je fais l'hypothèse suivante :

Hyp : Pour générer la valeur de référence, le saut conditionnel n'a jamais été pris.

En faisant tourner mon implémentation de l'algorithme sur des entrées choisies, je récupère des sorties déterministes (en utilisant cette hypothèse). Pour l'inversion, je constate que je n'ai pas d'inversion déterministe et que plusieurs entrées peuvent correspondre à une unique sortie. Néanmoins, l'entrée que je cherche fait bien partie de cet ensemble de possibles, ce qui me conforte dans mon approche.

Lorsque je tente d'inverser la valeur de référence cependant, je ne trouve aucune solution possible. J'ai vérifié plusieurs fois mon algorithme, les valeurs intermédiaires, changé l'ordre des octets mais cela n'a pas fonctionné. J'ai donc une implémentation d'un algorithme que j'arrive tant bien que mal à inverser lorsque je fais moi-même la dérivation, mais je n'arrive pas à inverser une valeur de référence générée selon cet algorithme. J'en déduis que l'algorithme que j'ai implémenté est différent de l'algorithme qui a été utilisé pour générer la valeur de référence, mais je n'arrive pas à déterminer la source ni la nature de cette différence.

Par ailleurs, lorsque le saut conditionnel n'est pas emprunté dans la fonction `do_smth1`, la machine virtuelle exécute 5 instructions de plus. Lorsque le saut conditionnel n'est pas emprunté dans la fonction `do_smth2`, la machine virtuelle exécute 2 instructions de plus. Cela signifie que si je compare deux exécutions 'extrêmes' : une où le saut conditionnel a systématiquement été emprunté et une où le saut conditionnel n'a jamais été emprunté, il y a une différence de $4 * (16 * 5 + 16 * 2)$ instructions soit 448 (0x1c0) instructions exécutées. Je n'ai pas fait les calculs exacts, néanmoins dans les traces que je génère, j'ai pu constater qu'il y avait environ 0x22e0 instructions exécutées (le résultat est non déterministe, mais mes observations montrent que le saut conditionnel est presque toujours pris et cette valeur est à peu de choses près exacte). En revanche sur les premiers tests effectués, qui se sont déroulés sans débogueur, j'observais systématiquement 0x2496 tours de boucle, correspondant à autant d'instructions exécutées par la machine virtuelle. Ces éléments apportent deux choses :

- Sur une exécution standard (sans débogueur) d'un programme qui réimplémente exactement le binaire, la machine virtuelle ne prend jamais (ou presque jamais) le saut conditionnel.
- La seule différence entre les tests qui ont généré ces résultats différents est la présence d'un débogueur (le blob et l'argument étant identiques).

Mon hypothèse de départ qui m'a servi pour toute l'analyse sur l'absence de mécanismes d'anti-débogage du système est sérieusement remise en question, mais pour le moment je n'ai rien observé de tel dans le code.

Je commence par chercher le sens des instructions dont je ne comprends pas tout à fait le fonctionnement, en continuant tout de même à utiliser gdb. En scrutant les valeurs de registres, les adresses et la mémoire, je trouve une chaîne de caractère que je n'avais pas remarquée avant. Stockée à l'adresse 0x0e20528d, elle contient "(ps -ef |grep -q 'qemu-system.* -[s] |qemu-system.* -[g]db') && exit 42". Lorsque je vérifie sa présence avec Ghidra, elle est effectivement stockée sous une forme obfusquée et semble décodée à l'amorçage de la partition contenant le TEE. La chaîne de caractères juste après celle-ci dans la mémoire est obfusquée aussi, il s'agit de "/proc/self/cmdline" qui est un fichier supposé contenir la commande ayant servi à lancer le programme.

La commande que je viens de trouver semble vérifier que les flags donnés à qemu au démarrage ne contiennent ni '-s', ni '-dbg', ni '-bg'. Il s'agit des différents flags qui permettent de lancer qemu en mode serveur gdb. Pour ce qui est de la fonction exit, je n'ai jamais observé le système ou le programme s'interrompre brutalement alors que j'étais en train de le déboguer. Même si je ne suis pas sûr que ces chaînes de caractères soient réellement utilisées, il est possible que d'autres éléments d'anti-débogage se basent sur les mêmes mécanismes, en utilisant les flags donnés à qemu au démarrage.

Je fais l'hypothèse suivante :

Hyp : Il existe un mécanisme d'anti-debogage dans le TEE qui se base sur les flags de lancement de qemu pour altérer l'exécution du système.

Je télécharge le code source de qemu 3.1.0 et je change le flag 'gdb' en 'totoro'. Deux fichiers ont été modifiés et le patch est le suivant :

```
--- a/qemu-options.def
+++ b/qemu-options.def
@@ -673,8 +673,8 @@ DEF("overcommit", HAS_ARG, QEMU_OPTION_overcommit,
 "
         cpu-pm=on|off controls cpu power management (default:
         off)\n",
 QEMU_ARCH_ALL)

-DEF("gdb", HAS_ARG, QEMU_OPTION_gdb, \
--gdb dev          wait for gdb connection on 'dev'\n", QEMU_ARCH_ALL)
+DEF("totoro", HAS_ARG, QEMU_OPTION_gdb, \
+--totoro dev       wait for gdb connection on 'dev'\n", QEMU_ARCH_ALL)

diff --git a/qemu-options.hx b/qemu-options.hx
index 269eda7..4d1db56 100644
--- a/qemu-options.hx
+++ b/qemu-options.hx
@@ -3293,8 +3293,8 @@ host CPU is not overcommitted. When used, host
 estimates of CPU cycle and power
 utilization will be incorrect, not taking into account guest idle time.
 ETEXTI

-DEF("gdb", HAS_ARG, QEMU_OPTION_gdb, \
-  "--gdb dev          wait for gdb connection on 'dev'\n", QEMU_ARCH_ALL
-)
+DEF("totoro", HAS_ARG, QEMU_OPTION_gdb, \
+  "--totoro dev       wait for gdb connection on 'dev'\n",
+  QEMU_ARCH_ALL)
```

Je lance la version modifiée qemu avec ce nouveau flag à la place du '-s' utilisé jusqu'alors et je régénère une trace. Le nombre d'instructions exécutées n'est toujours pas 0x2496, néanmoins j'observe une différence de taille entre les deux exécutions : la table de substitution utilisée par l'algorithme n'est plus la même.

Dans les exécution précédentes, lorsque le binaire allait chercher l'octet à la position 0x72111, il déchiffrait puis donnait l'octet '0xfa'. Dans cette exécution, il déchiffre puis donne l'octet '0x1c'. Il se trouve que le chargement de la table en mémoire a été décalé de 0x1000 octets entre les deux exécutions. Comme j'ai reproduit l'implémentation de l'algorithme et tout particulièrement le déchiffrement du blob en me basant sur les observations faites avec gdb, j'avais reproduit ce comportement.

Je refais des tests avec cette procédure de substitution corrigée : pour chaque bloc dérivé par l'algorithme je ne trouve plus un ensemble d'antécédents possible, mais un et un seul antécédent : le bon. Je tente donc d'inverser la valeur de référence en utilisant cette nouvelle table de substitution et j'obtiens la valeur acad8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e. En relançant Qemu sans serveur gdb et en donnant l'argument au binaire, il m'affiche la chaîne de caractère "Win".

Je peu ajouter cette clé et déchiffrer le dernier coffre :

```
./add_key.py
  acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e
[+] Key with key_id 00000004 ok
[+] Key added into keystore
[+] Envoyez le flag SSTIC{
  acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e} à l
  'adresse challenge2019@sstic.org pour valider votre avancée
[+] Container /root/safe_03/.encrypted decrypted to /root/safe_03/
  decrypted_file
```

Le mystère s'éclaircit

Encore une fois, je récupère le fichier nouvellement déchiffré pour voir ce qu'il en est sur mon poste de travail. La commande `file` m'indique qu'il s'agit d'une archive au format `gzip`. Après décompression, le fichier est une archive `tar`.

Les fichiers extraits semblent contenir des données appartenant à un téléphone Android : les dossiers extraits sont nommés `com.android.apps.tag`, `com.android.backupconfirm` etc. Je commence donc par chercher si l'un des fichiers extraits ne contiendrait pas par hasard la chaîne de caractères 'SSTIC'.

```
$ grep -i sstic * -R
Fichier binaire com.android.providers.contacts/databases/contacts2.db
correspondant
Fichier binaire com.android.providers.telephony/databases/mmssms.db
correspondant
Fichier binaire com.google.android.apps.messaging/databases/bugle_db
correspondant
Fichier binaire com.google.android.apps.messaging/databases/bugle_db-wal
correspondant
```

Je constate que la base de données de contacts ainsi que la base de données de SMS semblent contenir des informations intéressantes. En regardant le contenu de la base de données des contacts, je trouve six contacts différents dans la table `data` de la base, dont certains au nom plutôt évocateur :

```
Mister X03 : +33 6 12 34 56 78
Mister X02 : +33 6 01 02 03 04
SuperPharma : +33 6 12 34 56 79
Traiteur SSTIC : +33 6 23 23 23 23
Kevin Leserveur : +33 6 42 42 42 42
Mister X01 : +33 6 89 88 87 86
```

En regardant la base de données de SMS cette fois-ci, il est possible de reconstruire plusieurs fils de conversation. Voici les messages envoyés et reçus, chronologiquement :

```
De MisterX03: Reçu le 2018-05-15 11:05:12.370000 - Bonjour. Une action d
'ampleur est prévue pendant la conférence SSTIC le 13 juin après-midi
et dans la soirée. Pouvez-vous profiter du rassemblement d'un grand
nombre d'experts pour essayer de les mettre hors d'état de nuire

De SuperPharma: Reçu le 2018-05-25 01:06:29.336000 - Bonjour. Votre
commande pour les produits "DUPHALAC 1L" (10 articles) a bien été exp
édiée. Vous pouvez suivre votre commande à l'adresse suivante :
megasuivi.com/abXdSS
De SuperPharma: Reçu le 2018-05-28 18:06:12.360000 - Bonjour. Merci d'
avoir réceptionné vos produits. Vous pouvez noter votre article à l'
adresse superpharma.com/duphalac-1l.html

À MisterX03: Envoyé le 2018-06-01 09:18:17.947000 - OK, l'opération
suit son cours, je vous confirmerai les résultats.
```

À Kevin Leserveur: Envoyé le 2018-06-01 11:43:19.214000 - Bonjour Kévin . Désolé finalement nous n'avons plus besoin de vous le 13/06/2018, car nous avons déjà trop de personnel. Nous vous recontacterons pour une future mission. Agence Interim'expert

À Traiteur SSTIC: Envoyé le 2018-06-10 08:52:24.872000 - Bonjour, l'agence Interim'expert m'a contacté pour remplacer Kevin qui est souffrant pour le service du 13/06/2018 à la Halle Martenot. Pouvez-vous me préciser les horaires.

De Traiteur SSTIC: Reçu le 2018-06-10 12:11:12.535000 - Bonjour, merci d'être présent à partir de 9h30 le 13/06/2018. Vous serez affecté au poste des plats chauds.

À Traiteur SSTIC: Envoyé le 2018-06-10 12:11:26.545000 - Bien reçu. Bonne journée.

À MisterX03: Envoyé le 2018-06-13 23:52:30.418000 - Je vous confirme le semi-succès de l'opération "Ça coule de source". Je n'ai malheureusement pu avoir accès qu'à la moitié des préparations et je n'ai touché que la moitié du public.

De MisterX03: Reçu le 2018-06-14 09:22:26.503000 - OK, félicitations tout de même. À renouveler l'an prochain avec plusieurs infiltrés.

De MisterX02: Reçu le 2019-02-02 11:45:04.469000 - Bonjour, nous avons identifié un profil idéal pour l'usurpation d'identité dans le cadre du plan visant à focaliser l'attention des pentesteurs. Vous trouverez les identifiants du blog et du compte twitter sur notre outil interne.

À MisterX02: Envoyé le 2019-02-02 11:45:23.308000 - Bien reçu, je lance la campagne tout de suite.

De MisterX02: Reçu le 2019-02-13 16:50:04.515000 - Bravo. Ce billet "Pentest et pentesteur" nous a paru un peu gros lors de la publication, mais cela semble marcher au-delà de nos espérances. Continuez à alimenter la conversation, nous avons déjà constaté une perte d'attention chez la plupart des gens visés.

De MisterX01: Reçu le 2019-03-01 09:32:05.449000 - Bonjour, comme chaque année, l'opération "challenge SSTIC" se prépare avec des attaques ciblées courant la première semaine d'avril. Pouvez-vous faire en sorte de focaliser l'attention à ce moment-là.

À MisterX01: Envoyé le 2019-03-15 22:07:20.324000 - Mission accomplie, comme d'habitude la perspective d'une tournée de shooter au cactus a suffi à corrompre le CO et à choisir la date de publication du challenge. Pour être sûr que les experts sont toujours occupés, j'ai redirigé l'adresse 9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org vers votre boîte mail. Tant que vous ne voyez passer aucun mail, la voie est libre...

De MisterX03: Reçu le 2019-03-29 12:40:24.928000 - Bonjour, Cette année nous allons profiter du social event pour effectuer nos attaques ciblées. Merci de recruter en conséquence pour atteindre le maximum de personnes

À MisterX03: Envoyé le 2019-04-01 09:01:17.892000 - Bien noté, je pense cibler le stand des huîtres pour plus de discrétion et d'efficacité.

La première moitié des messages a été envoyée avant ou pendant le l'édition 2018 du SSTIC. Cette année-là,

une intoxication alimentaire avait touché une partie des participants, avec des effets qu'il est inutile de détailler (le Duphalac est un laxatif). Visiblement, il s'agit du téléphone de l'un des pépétreur de cette opération qui a été saisi³⁰.

Les messages suivants, envoyés aux alentours de Février 2019 mentionnent une campagne de détournement d'attention. Il y a effectivement eu, le 6 Février 2019, la publication d'un billet de blog intitulé "Pentest et Pentesteur". Ce billet a suscité des réactions plutôt nombreuses sur Twitter de la part de membres de la communauté sécurité informatique française, qui ont mené à trois autres billets de blog du même auteur quelques jours plus tard.

Enfin, les derniers messages font état de la corruption du Comité d'Organisation dans le but de maîtriser le calendrier du challenge³¹. Ces messages contiennent l'adresse mail tant convoitée : 9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org.

Attention tout de même aux huîtres lors du social event cette année.

Conclusion

Il s'agit de ma première participation au challenge SSTIC donc je n'ai pas d'élément de comparaison avec les éditions précédentes.

Ce fut très instructif et m'a permis de m'initier à l'utilisation d'outils de débogage et de rétro-ingénierie. J'ai pu expérimenter diverses méthodes de résolution et de récolte d'informations pour pallier à mon inexpérience dans ce domaine, ce qui fut au final très plaisant.

Remerciements

Je tiens à remercier tout particulièrement mes collègues pour avoir suivi mes pérégrinations dans les épreuves et qui m'ont supporté pendant la résolution, puis la rédaction de la solution.

Je remercie aussi mes amis qui m'ont encouragé au cours de ces cinq semaines de résolution puis ces deux semaines de rédaction.

Je remercie les concepteurs du challenge pour cette édition.

Enfin, je n'aurais pas pu résoudre ce challenge sans tous les outils en source ouverte utilisés en cours de route et donc je remercie les personnes les ayant conçus, développés, maintenus, documentés...

Annexe A : Script pour l'étape 1

Le programme Python suivant permet de résoudre la première étape et de générer les différents graphiques impliqués dans la solution :

```
# find_rsa_spa.py
import numpy as np
from matplotlib import pyplot as plt

data = np.load("arr_0.npy")

# Entire trace, then the start of the algorithm
plt.subplot(211)
plt.plot(data)
plt.subplot(212)
plt.plot(data[700000-4000:720000])
```

30. La Halle Martenot était le lieu de restauration de l'édition 2018, qui était gérée par un traiteur.

31. Le Cactus est un bar rennais à proximité du lieu de la conférence.

```

plt.show()

trace = data[700073:700073+788813]

# Detect holes
holes = np.array(np.zeros(len(trace)))
threshold = 0.145
high = 0.2
count = 0
first_idx = 0
count_thresh = 250
for i in range(len(trace)):
    sample = trace[i]
    # The sample is above the threshold
    if threshold < sample:
        # We just had more than count_thresh samples above, we just got another bit
        if count >= count_thresh:
            for j in range(count):
                holes[j+first_idx] = high
            # Else, we were in a normal operation
            count = 0
        # Else the sample is below
    else:
        if count == 1:
            first_idx = i
        count += 1

# Invert the signal
for i in range(len(holes)):
    holes[i] = abs(high-holes[i])

# Superposes the original signal and the work signal
plt.plot(trace[70000:95000], color="b")
plt.plot(holes[70000:95000], color="r")
plt.show()

sequences = []

# Count the number of samples per peak
old_val = holes[0]
count = 0
for t in holes:
    if t != old_val:
        if t == 0:
            sequences.append(count)
            count = 0
        if t != 0:
            count += 1
        old_val = t

if old_val != 0:
    sequences.append(count)

# Find the min and max timing to compute the median timing

```

```

bits = []
min_seq = len(trace)
max_seq = 0
for seq in sequences:
    if min_seq > seq:
        min_seq = seq
    if max_seq < seq:
        max_seq = seq

median = (max_seq+min_seq) / 2

plt.hist(sequences)
plt.axvline(x=median, color='red')
plt.xlabel("Nombre d'échantillons")
plt.ylabel("Nombre de pics")
plt.show()

# Decide wether we got a 0 or a 1 based on the median timing
for seq in sequences:
    if seq < median:
        bits.append(0)
    else:
        bits.append(1)

# Rebuild the key
d = 0
for b in bits[::-1]:
    d = (d << 1) + b

print("Private exponent is {:x}".format(d))

```

Annexe B : Scripts pour l'étape 2

Les programmes suivants m'ont permis de résoudre ce challenge :

- bf_safe_key1.py : Script Python qui complète le script initial et parcourt tout l'espace des clés pour retrouver la bonne valeur
- gen_keys.c : Programme C qui génère toutes les clés possibles et les stocke dans 16 fichiers
- bf_key.go : Programme Go qui parcourt les 16 fichiers et teste toutes les clés possibles

```

# bf_safe_key1.py
#!/usr/bin/python
# -*- coding: utf-8 -*-

import hashlib
import sys
import binascii

def info(msg):
    print("\033[34;1m\033[0m] %s" % (msg))

def ok(msg):
    print("\033[32;1m+\033[0m] %s" % (msg))

```

```

def warn(msg):
    print("\033[33;1mw\033[0m] %s" % (msg))

def error(msg):
    print("\033[31;1m!\033[0m] %s" % (msg))
def secure_device(a, b, op, button1=0, button2=0, button3=0, button4=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
    """
    # Get op
    op0 = op & 0x01
    op1 = (op & 0x02) >> 1

    if button3:
        ai = ((a << 1) | ((a & 0x80) >> 7)) & 0xff
    else:
        ai = a

    if button4:
        bi = ((b << 1) | ((b & 0x80) >> 7)) & 0xff
    else:
        bi = b

    opi0 = button1 ^ op0
    opi1 = button2 ^ op1

    opi = opi0 | (opi1 << 1)

    if opi == 3:
        out = ai+bi & 0xff
    elif opi == 2:
        out = ai ^ bi
    elif opi == 1:
        out = ai | bi
    elif opi == 0:
        out = ai & bi

    return out

def step1(key):
    but1 = key & 1
    but2 = (key & 2) >> 1
    but3 = (key & 4) >> 2
    but4 = (key & 8) >> 3
    r = secure_device(0x35, 0x27, 3, button1=but1,
                      button2=but2, button3=but3, button4=but4)

```

```

r = secure_device(0x7e, r, 3, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x66, r, 2, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x8, r, 1, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x13, r, 0, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x1f, r, 1, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0xa, r, 2, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0xd3, r, 0, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0xc6, r, 3, button1=but1, button2=but2,
                  button3=but3, button4=but4)

return r

def step2(key):
    but1 = key & 1
    but2 = (key & 2) >> 1
    but3 = (key & 4) >> 2
    but4 = (key & 8) >> 3

    r = secure_device(0xde, 0xab, 0, button1=but1,
                      button2=but2, button3=but3, button4=but4)
    r = secure_device(0x67, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x2a, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x6d, r, 1, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x4a, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xe7, r, 0, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x1c, r, 1, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x35, r, 0, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xde, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xf7, r, 0, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xda, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)

    return r

def step3(key):
    but1 = key & 1
    but2 = (key & 2) >> 1
    but3 = (key & 4) >> 2
    but4 = (key & 8) >> 3

```

```

r = secure_device(0x14, 0x23, 3, button1=but1,
                  button2=but2, button3=but3, button4=but4)
r = secure_device(0x72, r, 0, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x48, r, 3, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x53, r, 1, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0xa7, r, 0, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x5f, r, 1, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x3, r, 3, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0xb7, r, 3, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x73, r, 1, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x37, r, 3, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0xc5, r, 2, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0xa4, r, 1, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x30, r, 0, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0xdd, r, 2, button1=but1, button2=but2,
                  button3=but3, button4=but4)

return r

```

```

def step4(key):
    but1 = key & 1
    but2 = (key & 2) >> 1
    but3 = (key & 4) >> 2
    but4 = (key & 8) >> 3
    r = secure_device(0xb0, 0x42, 2, button1=but1,
                      button2=but2, button3=but3, button4=but4)
    r = secure_device(0xbc, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xfc, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x54, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x30, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x97, r, 1, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xe8, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xd6, r, 0, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x26, r, 0, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xeb, r, 0, button1=but1, button2=but2,
                      button3=but3, button4=but4)

```

```

r = secure_device(0x68, r, 1, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x26, r, 0, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x9, r, 3, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0x2a, r, 2, button1=but1, button2=but2,
                  button3=but3, button4=but4)
r = secure_device(0xa9, r, 3, button1=but1, button2=but2,
                  button3=but3, button4=but4)
return r

def step5(key):
    but1 = key & 1
    but2 = (key & 2) >> 1
    but3 = (key & 4) >> 2
    but4 = (key & 8) >> 3
    r = secure_device(0xff, 0x12, 0, button1=but1,
                      button2=but2, button3=but3, button4=but4)
    r = secure_device(0xfd, r, 1, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xe5, r, 1, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x26, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x85, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x63, r, 1, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x93, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xba, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x97, r, 0, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xab, r, 1, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x6e, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xfd, r, 0, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x4c, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x50, r, 0, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xa, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xfc, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xe3, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xa6, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x64, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)

```

```

    r = secure_device(0x8e, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xc1, r, 1, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    return r

def step6(key):
    but1 = key & 1
    but2 = (key & 2) >> 1
    but3 = (key & 4) >> 2
    but4 = (key & 8) >> 3

    r = secure_device(0x90, 0x77, 1, button1=but1,
                      button2=but2, button3=but3, button4=but4)
    r = secure_device(0x8e, r, 0, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xbd, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x39, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x4c, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xc5, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xb6, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x93, r, 1, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x9f, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xd6, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x6e, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x39, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x40, r, 1, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x14, r, 2, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0xe6, r, 3, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    return r

def step7(key):
    but1 = key & 1
    but2 = (key & 2) >> 1
    but3 = (key & 4) >> 2
    but4 = (key & 8) >> 3
    r = secure_device(0xf, 0xab, 3, button1=but1,
                      button2=but2, button3=but3, button4=but4)
    r = secure_device(0xa2, r, 1, button1=but1, button2=but2,
                      button3=but3, button4=but4)
    r = secure_device(0x7c, r, 0, button1=but1, button2=but2,

```



```

        button3=but3, button4=but4)
    r = secure_device(0xff, r, 1, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0x51, r, 0, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0x78, r, 1, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0x2f, r, 3, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0xed, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0x4b, r, 3, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0x4d, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
    return r

def step8(key):
    but1 = key & 1
    but2 = (key & 2) >> 1
    but3 = (key & 4) >> 2
    but4 = (key & 8) >> 3
    r = secure_device(0x88, 0x74, 0, button1=but1,
        button2=but2, button3=but3, button4=but4)
    r = secure_device(0x48, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0x11, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0x76, r, 0, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0x2b, r, 3, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0xf8, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
    return r

def init(key):
    but1 = key & 1
    but2 = (key & 2) >> 1
    but3 = (key & 4) >> 2
    but4 = (key & 8) >> 3
    r = secure_device(0x46, 0x92, 0, button1=but1,
        button2=but2, button3=but3, button4=but4)
    r = secure_device(0xdf, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0x3e, r, 0, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0x3a, r, 3, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0x36, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0x8e, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
    r = secure_device(0xc9, r, 3, button1=but1, button2=but2,

```

```

        button3=but3, button4=but4)
r = secure_device(0xe7, r, 1, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x29, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0xc2, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x79, r, 0, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x2a, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x4c, r, 3, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0xde, r, 0, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x88, r, 0, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x8b, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x97, r, 3, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x6a, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x60, r, 1, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x0f, r, 0, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x5b, r, 3, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0xd0, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0xa9, r, 1, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0xe3, r, 3, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0xd0, r, 1, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x27, r, 0, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x90, r, 0, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x3b, r, 1, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x66, r, 2, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0xe2, r, 0, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0x24, r, 3, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0xee, r, 1, button1=but1, button2=but2,
        button3=but3, button4=but4)
r = secure_device(0xf2, r, 3, button1=but1, button2=but2,
        button3=but3, button4=but4)
return r

```

```

def main():
    info("Brute-force de la clé\n")

    r = init(0xf)
    if r != 0xa1:
        print("0x{:0>2x}".format(r))
        error("Mauvaise initialisation, vérifiez l'état du secure é
lement")
        sys.exit(0)
    else:
        ok("Initialisation, check 1 OK")

    if init(0) != 0xe0:
        error("Mauvaise initialisation, vérifiez l'état du secure é
lement")
        sys.exit(0)
    else:
        ok("Initialisation, check 2 OK")

    # Generate the possible results for each step function
    count = 0
    poss1 = []
    poss2 = []
    poss3 = []
    poss4 = []
    poss5 = []
    poss6 = []
    poss7 = []
    poss8 = []

    a = ""
    b = ""
    c = ""
    d = ""
    e = ""
    f = ""
    g = ""
    h = ""

    for key in range(16):
        poss1.append(step1(key))
        poss2.append(step2(key))
        poss3.append(step3(key))
        poss4.append(step4(key))
        poss5.append(step5(key))
        poss6.append(step6(key))
        poss7.append(step7(key))
        poss8.append(step8(key))
        a += "{:02x}, ".format(step1(key))
        b += "{:02x}, ".format(step2(key))
        c += "{:02x}, ".format(step3(key))
        d += "{:02x}, ".format(step4(key))
        e += "{:02x}, ".format(step5(key))
        f += "{:02x}, ".format(step6(key))
        g += "{:02x}, ".format(step7(key))
        h += "{:02x}, ".format(step8(key))

```

```

# Print the possible results for each step function
print(a)
print(b)
print(c)
print(d)
print(e)
print(f)
print(g)
print(h)

print("Done generating all perms")
count = 0
for s1 in poss1:
    for s2 in poss2:
        for s3 in poss3:
            for s4 in poss4:
                count += 1
                print(count)
                for s5 in poss5:
                    for s6 in poss6:
                        for s7 in poss7:
                            for s8 in poss8:

                                key = bytearray(
                                    [s1, s2, s3, s4, s5, s6, s7, s8
                                ])

                                h = hashlib.sha256(key).hexdigest()

                                if "00
c8bb35d44dcbb2712a11799d8e1316045d64404f337f4ff653c27607f436ea" == h:
                                    # Display the key if found
                                    ok("Hash ok")
                                    print(s1)
                                    print(s2)
                                    print(s3)
                                    print(s4)
                                    print(s5)
                                    print(s6)
                                    print(s7)
                                    print(s8)

if __name__ == '__main__':
    main()

```

```

// gen_keys.c
#include "stdint.h"
#include "stdio.h"
#include "stdlib.h"

void gen_file(char* name, uint8_t s1_val, uint8_t* s2, uint8_t* s3,
    uint8_t* s4, uint8_t* s5, uint8_t* s6, uint8_t* s7, uint8_t* s8)
{

```

```

uint8_t k2, k3, k4, k5, k6, k7, k8;
FILE* fp = fopen(name, "w");

// Increment counter every 65536 keys
uint32_t count = 0;
for (k2 = 0; k2 < 16; k2++) {
    for (k3 = 0; k3 < 16; k3++) {
        for (k4 = 0; k4 < 16; k4++) {
            count++;
            printf("%d\n", count);
            for (k5 = 0; k5 < 16; k5++) {
                for (k6 = 0; k6 < 16; k6++) {
                    for (k7 = 0; k7 < 16; k7++) {
                        for (k8 = 0; k8 < 16; k8++) {
                            fprintf(fp, "%02x%02x%02x%02x%02x%02x%02x%02x\n", s1_val, s2[k2], s3[k3], s4[k4], s5[k5], s6[k6], s7[k7], s8[k8]);
                        }
                    }
                }
            }
        }
    }
}

fclose(fp);
}

int main()
{
    char filename[80];
    int key;
    uint8_t s1[16] = { 0xd7, 0x19, 0xdf, 0x40, 0xaf, 0x62, 0xbf, 0x81, 0x47, 0x39, 0xc7, 0xc2, 0x90, 0x72, 0x8f, 0x89 };
    uint8_t s2[16] = { 0x29, 0xd1, 0x08, 0xdb, 0x52, 0xa4, 0x10, 0xb5, 0x32, 0xd9, 0x90, 0xda, 0xf3, 0xb4, 0x20, 0xf5 };
    uint8_t s3[16] = { 0xed, 0x8d, 0xd9, 0xdf, 0xdb, 0x1c, 0xb0, 0xbf, 0xbd, 0x4f, 0xdc, 0xdd, 0x7b, 0x9f, 0x01, 0xbb };
    uint8_t s4[16] = { 0xaa, 0x38, 0xa9, 0x28, 0x55, 0x71, 0x53, 0x52, 0x1a, 0x4e, 0xad, 0x00, 0x35, 0x9c, 0x5b, 0x40 };
    uint8_t s5[16] = { 0xfd, 0x00, 0xaf, 0x47, 0xff, 0x01, 0x60, 0x8e, 0xed, 0x41, 0xde, 0xd4, 0xcb, 0x82, 0xbd, 0xa9 };
    uint8_t s6[16] = { 0x31, 0xb2, 0xf6, 0x64, 0x4e, 0x65, 0xed, 0xc8, 0x3d, 0xcf, 0xee, 0xe0, 0x1b, 0x9d, 0xdd, 0xc1 };
    uint8_t s7[16] = { 0xdd, 0x54, 0x4d, 0x4f, 0xbb, 0xa9, 0x9a, 0x9e, 0x60, 0x86, 0x05, 0xdf, 0xca, 0xe8, 0x0a, 0xbf };
    uint8_t s8[16] = { 0x83, 0x54, 0x78, 0xfb, 0x07, 0xaa, 0xf0, 0xf3, 0x97, 0x85, 0xd8, 0xfc, 0x4d, 0x48, 0xb0, 0xfd };

    for (key = 15; key >= 0; key--) {
        sprintf(filename, "keys%02d.db", key + 1);
        gen_file(filename, s1[key], s2, s3, s4, s5, s6, s7, s8);
    }
}

```

```

// bf_key.go
package main

import (
    "bufio"
    "bytes"
    "crypto/sha256"
    "encoding/hex"
    "fmt"
    "log"
    "os"

    "golang.org/x/crypto/scrypt"
)

func deriveAesKey(key []byte) {
    salt := []byte("sup3r_s3cur3_k3y_d3r1v4t10n_s4lt")
    n := 1 << 0xd
    r := 1 << 3
    p := 1 << 1
    dklen := 32
    derived, _ := scrypt.Key(key, salt, n, r, p, dklen)
    encoded := hex.EncodeToString(derived)

    fmt.Println("AES key : ")
    fmt.Println(encoded)
}

func tryKeysInFile(filename string, check []byte) bool {
    file, err := os.Open(filename)
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()

    scanner := bufio.NewScanner(file)
    count := 0
    dst := make([]byte, 8)
    for scanner.Scan() {
        count++
        if count%65536 == 0 {
            fmt.Println(count / 65536)
        }
        buf := scanner.Bytes()
        _, _ = hex.Decode(dst, buf)
        sum := sha256.Sum256(dst)
        if bytes.Equal(sum[:], check) {
            fmt.Printf("%02x%02x%02x%02x%02x%02x%02x%02x\n", dst[0], dst
[1], dst[2], dst[3], dst[4], dst[5], dst[6], dst[7])
            fmt.Println("ok")
            deriveAesKey(dst)
            return true
        }
    }
}

```

```

    return false
}

func main() {
    // Prepare check value
    check := []byte("00
c8bb35d44dcbb2712a11799d8e1316045d64404f337f4ff653c27607f436ea")
    decoded := make([]byte, hex.DecodedLen(len(check)))
    hex.Decode(decoded, check)

    for i := 15; i >= 0; i-- {
        filename := fmt.Sprintf("keys%02d.db", i+1)
        if tryKeysInFile(filename, decoded) {
            break
        }
    }
}

```

Annexe C : Programmes pour l'étape 3

Pour cette étape, les programmes utilisés ont été :

- `compute_prog_exec_time.py` : Ce script Python a permis de mesurer le temps d'exécution du programme en fonction de la longueur de l'argument
- `timestamp.py` : Ce script Python affiche l'horodatage courant
- `trace_except.gdb` : Ce script gdb génère une trace qui permet de voir quand les différentes fonctions impliquées dans la gestion d'exception ont été appelées
- `display_trace_except.py` : Ce script Python affiche deux traces d'exécutions côte à côte avec matplotlib. Les deux tableaux d'horodatage correspondent à des traces précédentes, mais peuvent être modifiés
- `trace_dwarf_vm.gdb` : Ce script gdb permet de tracer toute l'exécution de la VM dwarf, c'est à dire l'instruction courante et l'état de la pile avant l'exécution de chaque instruction. En l'occurrence, il n'active la trace que lorsque la première instruction est lue. Il est possible de changer cette adresse afin de pouvoir tracer des segments spécifiques.
- `extract_dwarf.go` : Ce programme Go extrait le code objet DWARF depuis le binaire
- `dwarf_const.go` : Ce fichier Go contient les constantes DWARF
- `disass_dwarf.go` : Ce programme Go prend en entrée un fichier binaire DWARF et le désassemble. Ici, il ne désassemble spécifiquement pas la section de données. Il se lance avec la commande `go run disass_dwarf.go dwarf_const.go program.dw`
- `stack_machine.go` : Ce programme Go implémente une machine à pile minimaliste utilisant un pseudo-langage
- `parse_shuffle8.go` : Ce programme Go parcourt du code dwarf désassemblé et essaye de générer du code Python suivant l'automate présenté dans la solution
- `solve_step3.py` : Ce script Python réimplémente l'algorithme. Il implémente aussi l'inversion de cet algorithme, qui forme la solution de l'étape.

```

# compute_prog_exec_time.py
import os
import time

samples = 50
res = []

```

```

for i in range(257):
    print(i)
    total = 0
    for j in range(samples):
        start = time.time_ns()
        os.system("/root/safe_01/decrypted_file " + "C" * i)
        total += time.time_ns()-start
    res.append(total/samples/1000000)

for i in range(len(res)):
    print("{}: {:.4f}".format(i, res[i]))

```

```

# timestamp.py
#!/usr/bin/python
import time

print("TIMESTAMP {}".format(int(time.time_ns()/1000000)))

```

```

# trace_except.gdb
set args AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
set pagination off

# Appelle le script qui affiche l'horodatage courant
define timestamp
shell /tmp/timestamp.py
end

# Place un point d'arrêt sur __libc_start_main
b __libc_start_main
# Lance le programme
r

# Il faut que le programme ait déjà chargé les bibliothèques partagées
# pour pouvoir placer des points d'arrêt sur les symboles de celles-ci

break __cxa_begin_catch
commands 2
printf "__cxa_begin_catch: "
timestamp
continue
end

break __cxa_end_catch
commands 3
printf "__cxa_end_catch: "
timestamp
continue
end

break __cxa_get_globals

```

```

commands 4
printf "__cxa_get_globals: "
timestamp
continue
end

break __cxa_get_globals_fast
commands 5
printf "__cxa_get_globals_fast: "
timestamp
continue
end

break __cxa_get_exception_ptr
commands 6
printf "__cxa_get_exception_ptr: "
timestamp
continue
end

break __gxx_personality_v0
commands 7
printf "__gxx_personality_v0: "
timestamp
continue
end

break _Unwind_RaiseException
commands 8
printf "_Unwind_RaiseException: "
timestamp
continue
end

break _Unwind_FindEnclosingFunction
commands 9
printf "_Unwind_FindEnclosingFunction: "
timestamp
continue
end

break _Unwind_GetDataRelBase
commands 10
printf "_Unwind_GetDataRelBase: "
timestamp
continue
end

break _Unwind_GetTextRelBase
commands 11
printf "_Unwind_GetTextRelBase: "
timestamp
continue
end

break _Unwind_GetIP

```

```

commands 12
printf "_Unwind_GetIP: "
timestamp
continue
end

break _Unwind_Resume
commands 13
printf "_Unwind_Resume: "
timestamp
continue
end

break _Unwind_SetGR
commands 14
printf "_Unwind_SetGR: "
timestamp
continue
end

break _Unwind_SetIP
commands 15
printf "_Unwind_SetIP: "
timestamp
continue
end

break _Unwind_GetIPInfo
commands 16
printf "_Unwind_GetIPInfo: "
timestamp
continue
end

break _Unwind_GetGR
commands 17
printf "_Unwind_GetGR: "
timestamp
continue
end

break _Unwind_GetRegionStart
commands 18
printf "_Unwind_GetRegionStart: "
timestamp
continue
end

break _Unwind_GetLanguageSpecificData
commands 19
printf "_Unwind_GetLanguageSpecificData: "
timestamp

```

```

continue
end

break _Unwind_GetCFA
commands 20
printf "_Unwind_GetCFA: "
timestamp
continue
end

break _Unwind_Find_FDE
commands 21
printf "_Unwind_FindFDE: "
timestamp
continue
end

break get_unexpected
commands 22
printf "get_unexpected: "
timestamp
continue
end

break get_terminate
commands 23
printf "get_terminate: "
timestamp
continue
end

b *0x402f20

c

```

```

# display_trace_except.py
from matplotlib import pyplot as plt

# Chaque fonction dispose d'une couleur
colors = {
    "__cxa_get_globals": 'b',
    "get_unexpected": 'g',
    "get_terminate": 'r',
    "_Unwind_RaiseException": 'c',
    "_Unwind_FindFDE": 'm',
    "_Unwind_GetGR": 'y',
    "__gxx_personality_v0": 'k',
    "_Unwind_GetLanguageSpecificData": 'gold',
    "_Unwind_GetRegionStart": 'orange',
    "_Unwind_GetIPInfo": 'purple',

```



```

1557423755086, 1557423755660, 1557423756370, 1557423757059,
1557423757633, 1557423758206, 1557423758785, 1557423759357,
1557423759937, 1557423760523, 1557423761096, 1557423761673,
1557423762252, 1557423763014, 1557423763594, 1557423764179,
1557423764770, 1557423765349, 1557423765925, 1557423766538,
1557423767133]

# Deuxième trace d'exécution
trace_2 = [1557421049654, 1557421050240, 1557421050814, 1557421051399,
1557421051974, 1557421052550, 1557421053139, 1557421053726,
1557421054306, 1557421054881, 1557421055468, 1557421056042,
1557421056620, 1557421057193, 1557421057777, 1557421058371,
1557421058950, 1557421059595, 1557421060254, 1557421060841,
1557421061408, 1557421061973, 1557421062539, 1557421063114,
1557421063681, 1557421064254, 1557421064825, 1557421065409,
1557421065979, 1557421066546, 1557421067133, 1557421067698,
1557421068268, 1557421068841, 1557421069418, 1557421069986,
1557421070562]

# Transforme les horodatages en une durée, relativement au premier
  horodatage enregistré
x_start = trace_2[0]
t_start = trace_1[0]
for i in range(len(trace_2)):
    trace_2[i] = trace_2[i]-x_start
    trace_1[i] = trace_1[i]-t_start

# Il faut ruser un peu pour afficher des points de couleur différente
  sur la même ligne. En l'occurrence, il y a un graphique différent par
  fonction par trace.
to_plot = {}

for i in range(len(trace_2)):
    label = labels[i]
    if label not in to_plot:
        to_plot[label] = {"x1s": [], "x2s": []}
    to_plot[label]["x1s"].append(trace_2[i])
    to_plot[label]["x2s"].append(trace_1[i])

# Affichage de tous les graphiques
for label, elt in to_plot.items():
    plt.plot(elt["x1s"], [1]*len(elt["x1s"]),
              color=colors[label], linestyle='None',
              marker='o')
    plt.plot(elt["x2s"], [2]*len(elt["x2s"]),
              color=colors[label], linestyle='None', marker='o', label=
              label.lstrip('_'))

plt.legend(loc="upper right")
plt.ylim(ymin=0, ymax=10)

plt.show()

```

```

# trace_dwarf_vm.gdb
set args AAAAAAAAABBBBBBBCCCCCCCCDDDDDDDD
set pagination off
b __libc_start_main
r

# DWARF VM main loop is at _Unwind_Resume-3392
# DWARF VM function is at _Unwind_Resume-3392-124
break *_Unwind_Resume-3392
commands 2
printf "dw_op 0x%x\n", $w7
# Il s'agit du haut de la pile
x/2x $x26+$x19*8-8
c
end

disable 2

# La première instruction lue est à l'adresse 0x400258
rwatch *(0x400258+0x00)
commands 3
enable 2
c
end

c

```

```

// extract_dwarf.go
package main

import (
    "io/ioutil"
)

func main() {
    content, _ := ioutil.ReadFile("decrypted_file")
    offsetFirst := 0x258
    offsetLast := 0x1982 + 1
    bytecode := content[offsetFirst:offsetLast]
    ioutil.WriteFile("program.dw", bytecode, 0644)
}

```

```

// dwarf_const.go
package main

const (
    DwOpAddr          = 0x03
    DwOpDeref         = 0x06
    DwOpConst1u       = 0x08
    DwOpConst1s       = 0x09

```

DwOpConst2u	= 0x0a
DwOpConst2s	= 0x0b
DwOpConst4u	= 0x0c
DwOpConst4s	= 0x0d
DwOpConst8u	= 0x0e
DwOpConst8s	= 0x0f
DwOpConstu	= 0x10
DwOpConsts	= 0x11
DwOpDup	= 0x12
DwOpDrop	= 0x13
DwOpOver	= 0x14
DwOpPick	= 0x15
DwOpSwap	= 0x16
DwOpRot	= 0x17
DwOpXderef	= 0x18
DwOpAbs	= 0x19
DwOpAnd	= 0x1a
DwOpDiv	= 0x1b
DwOpMinus	= 0x1c
DwOpMod	= 0x1d
DwOpMul	= 0x1e
DwOpNeg	= 0x1f
DwOpNot	= 0x20
DwOpOr	= 0x21
DwOpPlus	= 0x22
DwOpPlusUconst	= 0x23
DwOpShl	= 0x24
DwOpShr	= 0x25
DwOpShra	= 0x26
DwOpXor	= 0x27
DwOpSkip	= 0x2f
DwOpBra	= 0x28
DwOpEq	= 0x29
DwOpGe	= 0x2a
DwOpGt	= 0x2b
DwOpLe	= 0x2c
DwOpLt	= 0x2d
DwOpNe	= 0x2e
DwOpLit0	= 0x30
DwOpLit1	= 0x31
DwOpLit2	= 0x32
DwOpLit3	= 0x33
DwOpLit4	= 0x34
DwOpLit5	= 0x35
DwOpLit6	= 0x36
DwOpLit7	= 0x37
DwOpLit8	= 0x38
DwOpLit9	= 0x39
DwOpLit10	= 0x3a
DwOpLit11	= 0x3b
DwOpLit12	= 0x3c
DwOpLit13	= 0x3d
DwOpLit14	= 0x3e
DwOpLit15	= 0x3f
DwOpLit16	= 0x40
DwOpLit17	= 0x41

DwOpLit18	= 0x42
DwOpLit19	= 0x43
DwOpLit20	= 0x44
DwOpLit21	= 0x45
DwOpLit22	= 0x46
DwOpLit23	= 0x47
DwOpLit24	= 0x48
DwOpLit25	= 0x49
DwOpLit26	= 0x4a
DwOpLit27	= 0x4b
DwOpLit28	= 0x4c
DwOpLit29	= 0x4d
DwOpLit30	= 0x4e
DwOpLit31	= 0x4f
DwOpReg0	= 0x50
DwOpReg1	= 0x51
DwOpReg2	= 0x52
DwOpReg3	= 0x53
DwOpReg4	= 0x54
DwOpReg5	= 0x55
DwOpReg6	= 0x56
DwOpReg7	= 0x57
DwOpReg8	= 0x58
DwOpReg9	= 0x59
DwOpReg10	= 0x5a
DwOpReg11	= 0x5b
DwOpReg12	= 0x5c
DwOpReg13	= 0x5d
DwOpReg14	= 0x5e
DwOpReg15	= 0x5f
DwOpReg16	= 0x60
DwOpReg17	= 0x61
DwOpReg18	= 0x62
DwOpReg19	= 0x63
DwOpReg20	= 0x64
DwOpReg21	= 0x65
DwOpReg22	= 0x66
DwOpReg23	= 0x67
DwOpReg24	= 0x68
DwOpReg25	= 0x69
DwOpReg26	= 0x6a
DwOpReg27	= 0x6b
DwOpReg28	= 0x6c
DwOpReg29	= 0x6d
DwOpReg30	= 0x6e
DwOpReg31	= 0x6f
DwOpBreg0	= 0x70
DwOpBreg1	= 0x71
DwOpBreg2	= 0x72
DwOpBreg3	= 0x73
DwOpBreg4	= 0x74
DwOpBreg5	= 0x75
DwOpBreg6	= 0x76
DwOpBreg7	= 0x77
DwOpBreg8	= 0x78
DwOpBreg9	= 0x79

```

DwOpBreg10      = 0x7a
DwOpBreg11      = 0x7b
DwOpBreg12      = 0x7c
DwOpBreg13      = 0x7d
DwOpBreg14      = 0x7e
DwOpBreg15      = 0x7f
DwOpBreg16      = 0x80
DwOpBreg17      = 0x81
DwOpBreg18      = 0x82
DwOpBreg19      = 0x83
DwOpBreg20      = 0x84
DwOpBreg21      = 0x85
DwOpBreg22      = 0x86
DwOpBreg23      = 0x87
DwOpBreg24      = 0x88
DwOpBreg25      = 0x89
DwOpBreg26      = 0x8a
DwOpBreg27      = 0x8b
DwOpBreg28      = 0x8c
DwOpBreg29      = 0x8d
DwOpBreg30      = 0x8e
DwOpBreg31      = 0x8f
DwOpRegx        = 0x90
DwOpFbreg       = 0x91
DwOpBregx       = 0x92
DwOpPiece       = 0x93
DwOpDerefSize   = 0x94
DwOpXderefSize  = 0x95
DwOpNop         = 0x96
DwOpPushObjectAddress = 0x97
DwOpCall12      = 0x98
DwOpCall14      = 0x99
DwOpCallRef     = 0x9a
DwOpFormTlsAddress = 0x9b
DwOpCallFrameCfa = 0x9c
DwOpBitPiece    = 0x9d
DwOpLoUser      = 0xe0
DwOpHiUser      = 0xff
)

```

```

// disass_dwarf.go
package main

import (
    "fmt"
    "io/ioutil"
    "os"
)

func varint(data []byte, ip *int) (uint64, uint) {
    c := uint64(0)
    bits := uint(0)
    for *ip < len(data) {

```

```

        byte := data[*ip]
        *ip++
        c |= uint64(byte&0x7F) << bits
        bits += 7
        if byte&0x80 == 0 {
            return c, bits
        }
    }
    return 0, 0
}

func sleb128(data []byte, ip *int) int64 {
    c, bits := varint(data, ip)
    res := int64(c)
    if res&(1<<(bits-1)) != 0 {
        res |= -1 << bits
    }
    return res
}

func uleb128(data []byte, ip *int) uint64 {
    c, _ := varint(data, ip)
    return c
}

func const1u(data []byte, ip *int) uint8 {
    res := data[*ip]
    *ip++
    return res
}

func const1s(data []byte, ip *int) int8 {
    res := int8(data[*ip])
    *ip++
    return res
}

func const2u(data []byte, ip *int) uint16 {
    res := uint16(data[*ip])
    *ip++
    res |= uint16(data[*ip]) << 8
    *ip++
    return res
}

func const2s(data []byte, ip *int) int16 {
    res := uint16(data[*ip])
    *ip++
    res |= uint16(data[*ip]) << 8
    *ip++
    return int16(res)
}

func const4u(data []byte, ip *int) uint32 {
    res := uint32(data[*ip])
    *ip++

```

```

    res |= uint32(data[*ip]) << 8
    *ip++
    res |= uint32(data[*ip]) << 16
    *ip++
    res |= uint32(data[*ip]) << 24
    *ip++
    return res
}

func const4s(data []byte, ip *int) int32 {
    res := int32(data[*ip])
    *ip++
    res |= int32(data[*ip]) << 8
    *ip++
    res |= int32(data[*ip]) << 16
    *ip++
    res |= int32(data[*ip]) << 24
    *ip++
    return res
}

func const8u(data []byte, ip *int) uint64 {
    res := uint64(data[*ip])
    *ip++
    res |= uint64(data[*ip]) << 8
    *ip++
    res |= uint64(data[*ip]) << 16
    *ip++
    res |= uint64(data[*ip]) << 24
    *ip++
    res |= uint64(data[*ip]) << 32
    *ip++
    res |= uint64(data[*ip]) << 40
    *ip++
    res |= uint64(data[*ip]) << 48
    *ip++
    res |= uint64(data[*ip]) << 56
    *ip++
    return res
}

func const8s(data []byte, ip *int) int64 {
    res := int64(data[*ip])
    *ip++
    res |= int64(data[*ip]) << 8
    *ip++
    res |= int64(data[*ip]) << 16
    *ip++
    res |= int64(data[*ip]) << 24
    *ip++
    res |= int64(data[*ip]) << 32
    *ip++
    res |= int64(data[*ip]) << 40
    *ip++
    res |= int64(data[*ip]) << 48
    *ip++

```

```

    res |= int64(data[*ip]) << 56
    *ip++
    return res
}

func constu(data []byte, ip *int) uint64 {
    return uleb128(data, ip)
}

func readAndDisass(prog []byte, ip *int) {
    opcode := prog[*ip]
    fmt.Printf("0x400258+%04x: ", *ip)
    *ip++
    switch opcode {
    case DwOpAddr:
        addr := const8u(prog, ip)
        fmt.Printf("DW_OP_addr 0x%x\n", addr)
        break
    case DwOpDeref:
        fmt.Printf("DW_OP_deref\n")
        break
    case DwOpConst1u:
        cst := const1u(prog, ip)
        fmt.Printf("DW_OP_const1u 0x%x\n", cst)
        break
    case DwOpConst1s:
        cst := const1s(prog, ip)
        fmt.Printf("DW_OP_const1s 0x%x\n", cst)
        break
    case DwOpConst2u:
        cst := const2u(prog, ip)
        fmt.Printf("DW_OP_const2u 0x%x\n", cst)
        break
    case DwOpConst2s:
        cst := const2s(prog, ip)
        fmt.Printf("DW_OP_const2s 0x%x\n", cst)
        break
    case DwOpConst4u:
        cst := const4u(prog, ip)
        fmt.Printf("DW_OP_const4u 0x%x\n", cst)
        break
    case DwOpConst4s:
        cst := const4s(prog, ip)
        fmt.Printf("DW_OP_const4s 0x%x\n", cst)
        break
    case DwOpConst8u:
        cst := const8u(prog, ip)
        fmt.Printf("DW_OP_const8u 0x%x\n", cst)
        break
    case DwOpConst8s:
        cst := const8s(prog, ip)
        fmt.Printf("DW_OP_const8s 0x%x\n", cst)
        break
    case DwOpConstu:
        cst := uleb128(prog, ip)
        fmt.Printf("DW_OP_constu 0x%x\n", cst)
        break
    }
}

```

```

case DwOpConsts:
    cst := sleb128(prog, ip)
    fmt.Printf("DW_OP_consts 0x%x\n", cst)
    break
case DwOpDup:
    fmt.Printf("DW_OP_dup\n")
    break
case DwOpDrop:
    fmt.Printf("DW_OP_drop\n")
    break
case DwOpOver:
    fmt.Printf("DW_OP_over\n")
    break
case DwOpPick:
    cst := const1u(prog, ip)
    fmt.Printf("DW_OP_pick 0x%x\n", cst)
    break
case DwOpSwap:
    fmt.Printf("DW_OP_swap\n")
    break
case DwOpRot:
    fmt.Printf("DW_OP_rot\n")
    break
case DwOpXderef:
    fmt.Printf("DW_OP_xderef\n")
    break
case DwOpAbs:
    fmt.Printf("DW_OP_abs\n")
    break
case DwOpAnd:
    fmt.Printf("DW_OP_and\n")
    break
case DwOpDiv:
    fmt.Printf("DW_OP_div\n")
    break
case DwOpMinus:
    fmt.Printf("DW_OP_minus\n")
    break
case DwOpMod:
    fmt.Printf("DW_OP_mod\n")
    break
case DwOpMul:
    fmt.Printf("DW_OP_mul\n")
    break
case DwOpNeg:
    fmt.Printf("DW_OP_neg\n")
    break
case DwOpNot:
    fmt.Printf("DW_OP_not\n")
    break
case DwOpOr:
    fmt.Printf("DW_OP_or\n")
    break
case DwOpPlus:
    fmt.Printf("DW_OP_plus\n")
    break

```

```

case DwOpPlusUconst:
    cst := uleb128(prog, ip)
    fmt.Printf("DW_OP_plus_uconst 0x%x\n", cst)
    break
case DwOpShl:
    fmt.Printf("DW_OP_shl\n")
    break
case DwOpShr:
    fmt.Printf("DW_OP_shr\n")
    break
case DwOpShra:
    fmt.Printf("DW_OP_shra\n")
    break
case DwOpXor:
    fmt.Printf("DW_OP_xor\n")
    break
case DwOpSkip:
    off := const2s(prog, ip)
    fmt.Printf("DW_OP_skip 0x%x\n", off)
    break
case DwOpBra:
    off := const2s(prog, ip)
    fmt.Printf("DW_OP_bra 0x%x\n", off)
    break
case DwOpEq:
    fmt.Printf("DW_OP_eq\n")
    break
case DwOpGe:
    fmt.Printf("DW_OP_ge\n")
    break
case DwOpGt:
    fmt.Printf("DW_OP_gt\n")
    break
case DwOpLe:
    fmt.Printf("DW_OP_le\n")
    break
case DwOpLt:
    fmt.Printf("DW_OP_lt\n")
    break
case DwOpNe:
    fmt.Printf("DW_OP_ne\n")
    break
case DwOpLit0:
    fmt.Printf("DW_OP_lit0\n")
    break
case DwOpLit1:
    fmt.Printf("DW_OP_lit1\n")
    break
case DwOpLit2:
    fmt.Printf("DW_OP_lit2\n")
    break
case DwOpLit3:
    fmt.Printf("DW_OP_lit3\n")
    break
case DwOpLit4:
    fmt.Printf("DW_OP_lit4\n")

```

```

        break
    case DwOpLit5:
        fmt.Printf("DW_OP_lit5\n")
        break
    case DwOpLit6:
        fmt.Printf("DW_OP_lit6\n")
        break
    case DwOpLit7:
        fmt.Printf("DW_OP_lit7\n")
        break
    case DwOpLit8:
        fmt.Printf("DW_OP_lit8\n")
        break
    case DwOpLit9:
        fmt.Printf("DW_OP_lit9\n")
        break
    case DwOpLit10:
        fmt.Printf("DW_OP_lit10\n")
        break
    case DwOpLit11:
        fmt.Printf("DW_OP_lit11\n")
        break
    case DwOpLit12:
        fmt.Printf("DW_OP_lit12\n")
        break
    case DwOpLit13:
        fmt.Printf("DW_OP_lit13\n")
        break
    case DwOpLit14:
        fmt.Printf("DW_OP_lit14\n")
        break
    case DwOpLit15:
        fmt.Printf("DW_OP_lit15\n")
        break
    case DwOpLit16:
        fmt.Printf("DW_OP_lit16\n")
        break
    case DwOpLit17:
        fmt.Printf("DW_OP_lit17\n")
        break
    case DwOpLit18:
        fmt.Printf("DW_OP_lit18\n")
        break
    case DwOpLit19:
        fmt.Printf("DW_OP_lit19\n")
        break
    case DwOpLit20:
        fmt.Printf("DW_OP_lit20\n")
        break
    case DwOpLit21:
        fmt.Printf("DW_OP_lit21\n")
        break
    case DwOpLit22:
        fmt.Printf("DW_OP_lit22\n")
        break
    case DwOpLit23:

```

```

        fmt.Printf("DW_OP_lit23\n")
        break
    case DwOpLit24:
        fmt.Printf("DW_OP_lit24\n")
        break
    case DwOpLit25:
        fmt.Printf("DW_OP_lit25\n")
        break
    case DwOpLit26:
        fmt.Printf("DW_OP_lit26\n")
        break
    case DwOpLit27:
        fmt.Printf("DW_OP_lit27\n")
        break
    case DwOpLit28:
        fmt.Printf("DW_OP_lit28\n")
        break
    case DwOpLit29:
        fmt.Printf("DW_OP_lit29\n")
        break
    case DwOpLit30:
        fmt.Printf("DW_OP_lit30\n")
        break
    case DwOpLit31:
        fmt.Printf("DW_OP_lit31\n")
        break
    case DwOpReg0:
        fmt.Printf("DW_OP_reg0\n")
        break
    case DwOpReg1:
        fmt.Printf("DW_OP_reg1\n")
        break
    case DwOpReg2:
        fmt.Printf("DW_OP_reg2\n")
        break
    case DwOpReg3:
        fmt.Printf("DW_OP_reg3\n")
        break
    case DwOpReg4:
        fmt.Printf("DW_OP_reg4\n")
        break
    case DwOpReg5:
        fmt.Printf("DW_OP_reg5\n")
        break
    case DwOpReg6:
        fmt.Printf("DW_OP_reg6\n")
        break
    case DwOpReg7:
        fmt.Printf("DW_OP_reg7\n")
        break
    case DwOpReg8:
        fmt.Printf("DW_OP_reg8\n")
        break
    case DwOpReg9:
        fmt.Printf("DW_OP_reg9\n")
        break

```

```

case DwOpReg10:
    fmt.Printf("DW_OP_reg10\n")
    break
case DwOpReg11:
    fmt.Printf("DW_OP_reg11\n")
    break
case DwOpReg12:
    fmt.Printf("DW_OP_reg12\n")
    break
case DwOpReg13:
    fmt.Printf("DW_OP_reg13\n")
    break
case DwOpReg14:
    fmt.Printf("DW_OP_reg14\n")
    break
case DwOpReg15:
    fmt.Printf("DW_OP_reg15\n")
    break
case DwOpReg16:
    fmt.Printf("DW_OP_reg16\n")
    break
case DwOpReg17:
    fmt.Printf("DW_OP_reg17\n")
    break
case DwOpReg18:
    fmt.Printf("DW_OP_reg18\n")
    break
case DwOpReg19:
    fmt.Printf("DW_OP_reg19\n")
    break
case DwOpReg20:
    fmt.Printf("DW_OP_reg20\n")
    break
case DwOpReg21:
    fmt.Printf("DW_OP_reg21\n")
    break
case DwOpReg22:
    fmt.Printf("DW_OP_reg22\n")
    break
case DwOpReg23:
    fmt.Printf("DW_OP_reg23\n")
    break
case DwOpReg24:
    fmt.Printf("DW_OP_reg24\n")
    break
case DwOpReg25:
    fmt.Printf("DW_OP_reg25\n")
    break
case DwOpReg26:
    fmt.Printf("DW_OP_reg26\n")
    break
case DwOpReg27:
    fmt.Printf("DW_OP_reg27\n")
    break
case DwOpReg28:
    fmt.Printf("DW_OP_reg28\n")

```

```

        break
    case DwOpReg29:
        fmt.Printf("DW_OP_reg29\n")
        break
    case DwOpReg30:
        fmt.Printf("DW_OP_reg30\n")
        break
    case DwOpReg31:
        fmt.Printf("DW_OP_reg31\n")
        break
    case DwOpBreg0:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg0 0x%x\n", cst)
        break
    case DwOpBreg1:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg1 0x%x\n", cst)
        break
    case DwOpBreg2:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg2 0x%x\n", cst)
        break
    case DwOpBreg3:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg3 0x%x\n", cst)
        break
    case DwOpBreg4:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg4 0x%x\n", cst)
        break
    case DwOpBreg5:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg5 0x%x\n", cst)
        break
    case DwOpBreg6:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg6 0x%x\n", cst)
        break
    case DwOpBreg7:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg7 0x%x\n", cst)
        break
    case DwOpBreg8:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg8 0x%x\n", cst)
        break
    case DwOpBreg9:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg9 0x%x\n", cst)
        break
    case DwOpBreg10:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg10 0x%x\n", cst)
        break
    case DwOpBreg11:
        cst := sleb128(prog, ip)

```

```

        fmt.Printf("DW_OP_breg11 0x%x\n", cst)
        break
    case DwOpBreg12:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg12 0x%x\n", cst)
        break
    case DwOpBreg13:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg13 0x%x\n", cst)
        break
    case DwOpBreg14:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg14 0x%x\n", cst)
        break
    case DwOpBreg15:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg15 0x%x\n", cst)
        break
    case DwOpBreg16:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg16 0x%x\n", cst)
        break
    case DwOpBreg17:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg17 0x%x\n", cst)
        break
    case DwOpBreg18:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg18 0x%x\n", cst)
        break
    case DwOpBreg19:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg19 0x%x\n", cst)
        break
    case DwOpBreg20:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg20 0x%x\n", cst)
        break
    case DwOpBreg21:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg21 0x%x\n", cst)
        break
    case DwOpBreg22:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg22 0x%x\n", cst)
        break
    case DwOpBreg23:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg23 0x%x\n", cst)
        break
    case DwOpBreg24:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg24 0x%x\n", cst)
        break
    case DwOpBreg25:
        cst := sleb128(prog, ip)

```

```

        fmt.Printf("DW_OP_breg25 0x%x\n", cst)
        break
    case DwOpBreg26:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg26 0x%x\n", cst)
        break
    case DwOpBreg27:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg27 0x%x\n", cst)
        break
    case DwOpBreg28:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg28 0x%x\n", cst)
        break
    case DwOpBreg29:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg29 0x%x\n", cst)
        break
    case DwOpBreg30:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg30 0x%x\n", cst)
        break
    case DwOpBreg31:
        cst := sleb128(prog, ip)
        fmt.Printf("DW_OP_breg31 0x%x\n", cst)
        break
    case DwOpRegx:
        cst := uleb128(prog, ip)
        fmt.Printf("DW_OP_regx 0x%x\n", cst)
        break
    case DwOpFbreg:
        off := sleb128(prog, ip)
        fmt.Printf("DW_OP_fbreg 0x%x\n", off)
        break
    case DwOpBregx:
        cst := uleb128(prog, ip)
        off := sleb128(prog, ip)
        fmt.Printf("DW_OP_bregx 0x%x 0x%x\n", cst, off)
        break
    case DwOpPiece:
        size := uleb128(prog, ip)
        fmt.Printf("DW_OP_piece 0x%x\n", size)
        break
    case DwOpDerefSize:
        size := const1u(prog, ip)
        fmt.Printf("DW_OP_deref_size 0x%x\n", size)
        break
    case DwOpXderefSize:
        size := const1u(prog, ip)
        fmt.Printf("DW_OP_xderef_size 0x%x\n", size)
        break
    case DwOpNop:
        fmt.Printf("DW_OP_nop\n")
        break
    case DwOpPushObjectAddress:
        fmt.Printf("DW_OP_push_object_address\n")

```

```

        break
    case DwOpCall12:
        size := const2u(prog, ip)
        fmt.Printf("DW_OP_call12 0x%x\n", size)
        break
    case DwOpCall14:
        size := const4u(prog, ip)
        fmt.Printf("DW_OP_call14\n", size)
        break
    case DwOpCallRef:
        size := constu(prog, ip)
        fmt.Printf("DW_OP_call_ref\n", size)
        break
    case DwOpFormTlsAddress:
        fmt.Printf("DW_OP_form_tls_address\n")
        break
    case DwOpCallFrameCfa:
        fmt.Printf("DW_OP_call_frame_cfa\n")
        break
    case DwOpBitPiece:
        fmt.Printf("DW_OP_bit_piece\n")
        break
    case DwOpLoUser:
        fmt.Printf("DW_OP_lo_user\n")
        break
    case DwOpHiUser:
        fmt.Printf("DW_OP_hi_user\n")
        break
    }
}

func main() {
    filename := os.Args[1]
    content, err := ioutil.ReadFile(filename)
    if err != nil {
        fmt.Printf("Couldn't open the file: %s\n", err)
        os.Exit(1)
    }

    i := 0
    for i < len(content) {
        if i >= (0x400640-0x400258) && i <= (0x400ab5-0x400258) {
            i = 0x400ab5 - 0x400258
        }
        readAndDisass(content, &i)
    }
}

```

```

// stack_machine.go
package main

import (

```

```

    "bufio"
    "fmt"
    "os"
    "strings"
)

var memory = [...]uint32{0x489dddde, 0x67990f1, 0x95bf74a9, 0x77941ee7,
    0xe6d80e3, 0x2dedaf8b, 0xfb92cd42, 0xd0e867c0, 0xf2b3a3fb, 0x6c39ce47,
    0xe74f99e0, 0x5a24f221, 0xd6378fea,
    0xe23ca8c4, 0x84e3b1bc, 0xce5e10bf, 0xa2b364da, 0x41f250f0, 0xfe97040, 0
    x1cc05266, 0x16f87e4b, 0x515e26b7, 0xeea48dcb, 0x62b357e4, 0x39bd2041
    , 0x72cd387a, 0xf37aac8b, 0x5963b39b,
    0x30f75add, 0x103fecbc, 0x392e7a, 0x35df7adf, 0xe19abd13, 0xdaf8b35c, 0
    xf8798214, 0xb30c2305, 0x67980e9, 0x6900d940, 0x35e876f, 0xa3857014,
    0x56c8e162, 0xe9748f56, 0x91d4e409,
    0xdcc75a09, 0xac65f52f, 0x8571dd07, 0x19edcf6, 0x51cef9eb, 0x1eb1b17d, 0
    xabe446f, 0x3b277cfe, 0x843869bc, 0xb23ea298, 0x7c296f51, 0xcd799972,
    0x62180a64, 0xac052d5, 0xf0076205,
    0x13183193, 0xb908cb94, 0x4bf4cd3c, 0xdee5d48a, 0xf64f9a74, 0xd64a15d0,
    0xb2cad434, 0x64e9013b, 0xf46cc1d2, 0x9d78e9db, 0x11789216, 0
    x335689e6, 0x74c7edb, 0xe6eb6185, 0xd020170b,
    0xf304aa15, 0xbcfc2b69e, 0x4eb3d2ea, 0xd78d4d5c, 0x7ed2bfc4, 0x58ebf0f3,
    0x8b591c3f, 0xd3041f6b, 0x5cae88, 0xba696f5c, 0xc16c8ede, 0x9abcbb27,
    0x56d78d77, 0x765b3e20, 0xcf37212d,
    0x192e2dcf, 0x8caf2806, 0xbc9a575b, 0x776421ce, 0x527fb9eb, 0x69f84340,
    0xadbc7bd7, 0x73f2c329, 0x737f8a7f, 0xe301d3e4, 0x57eb2eb2, 0x5859b858
    , 0x2cc41979, 0xec69a639, 0x53b0d523,
    0x39a2f532, 0x8b29e35d, 0x44e2ce81, 0xcc10a16d, 0x44d9ff58, 0x77102c14,
    0xfb57817d, 0x3cf7c8c8, 0x1222868a, 0x4173d5d1, 0x3529ee32, 0
    x7a9df58e, 0x513525ac, 0x81954bac, 0xce53ccf5,
    0x79168728, 0xa2d660f8, 0xf30cc9ce, 0xf0b89c76, 0x89fb3a9, 0xc919dba8, 0
    x1f9e4dc3, 0xa2594e0c, 0x34ffe178, 0xb04414fb, 0xd31fb33a, 0x184d0278
    , 0x2c816a9a, 0xb993f2f2, 0xe4d8601c,
    0x49e2eede, 0x9cd50ce1, 0xc03e1e77, 0xa901869e, 0x7579de50, 0x726ac4ab,
    0x38d04840, 0xeabe1270, 0x8c40812d, 0xe84976b7, 0x172b04ad, 0
    x756606c4, 0x66258491, 0xb5a0bef8, 0x6bcc5cf3,
    0xa535ae94, 0xc97a87aa, 0x9103a8f6, 0xcc3b9e5f, 0xbb20be1f, 0xffcfef97,
    0x90954f16, 0x501ae1a6, 0x6ed589cd, 0x6826b02b, 0x565ff263, 0
    x8e8c369b, 0x6990be7a, 0x3525b840, 0x1847d7bb,
    0x355a40c7, 0xa3579f10, 0xe9edecae, 0xd0337ab1, 0x6355e5ba, 0x88975355,
    0x5ec0f3cf, 0xa0d6213d, 0x75389387, 0xe40216f0, 0xd980cce0, 0
    x6c88c67c, 0x829d419c, 0x3bf6451b, 0x11f07bfa,
    0xc4c1154e, 0xbd0735eb, 0x9cf8df9d, 0xe457be75, 0x63a6bd18, 0xefe77fd3,
    0x83421b63, 0x7f83072d, 0x44940f61, 0xf8bdcdf7, 0x61c802ca, 0xa30f9a8
    , 0x7ff03b37, 0xa26cc5a9, 0xe10e570d,
    0x95ea0c16, 0xa05e6b02, 0xc81d5384, 0x7785db05, 0x92c84c5f, 0x5584617, 0
    x82bcfe8d, 0x559ea1da, 0x4fd5cdb0, 0x9d871fed, 0xdd6f5539, 0x4ed1ef26
    , 0xfe6813c4, 0x1cfa71d5, 0xd5613aea,
    0xf1c9b8c, 0x2bcac45d, 0x65d00f41, 0x689be0d8, 0x68b01100, 0x635bd280, 0
    x954d5d4b, 0x72887f79, 0xce027a75, 0xfc01c66, 0x6a1bd3, 0x199a1c8e,
    0x87d6ee25, 0x938e9f08, 0xd8a11d4d,
    0x2b9a4d81, 0xb6f5d2e5, 0xd15c325a, 0x64eaafc1, 0xfd33b61c, 0x43c1bd57,
    0x37b8f048, 0x5cba7cf2, 0x72810cd0, 0xabfef454, 0xa76384ba, 0
    xd8861440, 0x36de5837, 0xf6a03f1, 0x10d48fa1,
    0x5883ec2f, 0xa8c00c9b, 0x618ffea4, 0xa05da206, 0xffb9e97a, 0x8a376781,
    0x3156b479, 0xe4af5ecd, 0x87d9e06f, 0xb4d4d459, 0xeb9a7d25, 0

```

```

    x59dffeaa, 0xdc8bf553, 0x6dce3c3a, 0x2162970e,
    0xe8c9929d, 0x6c3a9bf4, 0x45da5392, 0x9ceee7b0, 0x3f68d4eb, 0xcd29434f,
    0xe4df712, 0xb1a8c69a, 0x1c190f46, 0x2b45873c, 0x46afdfc9, 0x61e8883f
    , 0x979118c7, 0x70f991b1, 0x1f82604d,
    0xc18bf48f, 0xb327f4ff, 0x519a7508, 0xfa619b0d, 0x268d1490, 0x567e37c2,
    0x25a07691, 0x424359c0, 0x13320c53, 0xe7f742fd, 0x48b945ba, 0
    xcfa8e711, 0x8f5fb519, 0x2b7332a5, 0x10aa767c, 0x689f0cc0}

var memory6b4 = memory[27:]
var memory678 = memory[12:]

func pp64(elt uint64, i int) string {
    return fmt.Sprintf("%d : 0x%08x %08x", i, uint32((elt&0
    xffffffff00000000)>>32), uint32(elt&0xffffffff))
}

type Stack struct {
    data [64]uint64
    i    int
}

func (s *Stack) peek(index int) uint64 {
    return s.data[s.i-index-1]
}

func (s *Stack) push(elt uint64) {
    s.data[s.i] = elt
    s.i++
}

func (s *Stack) pop() uint64 {
    s.i--
    return s.data[s.i]
}

func (s *Stack) String() string {
    res := make([]string, 0)
    for i := 0; i < s.i; i++ {
        res = append(res, pp64(s.data[i], i))
    }
    return strings.Join(res, "\n")
}

func main() {
    reader := bufio.NewReader(os.Stdin)
    vm := &Stack{}
    for {
        fmt.Print("> ")
        line, _ := reader.ReadString('\n')
        words := strings.Fields(line)
        if len(words) == 0 {
            break
        }
        switch words[0] {
        case "--":
            break
        }
    }
}

```

```

    case "memory":
        if len(words) > 1 && words[1] == "8" {
            index := vm.pop()
            vm.push(uint64(memory[index])<<32 | uint64(memory[index
+1])))
        } else {
            vm.push(uint64(memory[vm.pop()]))
        }
        break
    case "memory6b4":
        if len(words) > 1 && words[1] == "8" {
            index := vm.pop()
            vm.push(uint64(memory6b4[index])<<32 | uint64(memory6b4[
index+1]))
        } else {
            vm.push(uint64(memory6b4[vm.pop()]))
        }
        break
    case "memory678":
        vm.push(uint64(memory678[vm.pop()]))
        break
    case "print":
        fmt.Println(vm)
        break
    case "push":
        push(vm, words)
        break
    case "pop":
        pop(vm)
        break
    case "plus":
        plus(vm)
        break
    case "minus":
        minus(vm)
        break
    case "mul":
        mul(vm)
        break
    case "or":
        or(vm)
        break
    case "and":
        and(vm)
        break
    case "shl":
        shl(vm)
        break
    case "shr":
        shr(vm)
        break
    case "xor":
        xor(vm)
        break
    case "pick":
        pick(vm, words)

```

```

        break
    case "rot":
        rot(vm)
        break
    case "swap":
        swap(vm)
        break
    case "dup":
        pick(vm, []string{"", "0"})
        break
    case "init":
        vm.push(0x4949494949494949)
        vm.push(0x4848484848484848)
        vm.push(0x4747474747474747)
        vm.push(0x4646464646464646)
        vm.push(0x4545454545454545)
        vm.push(0x4444444444444444)
        vm.push(0x4343434343434343)
        vm.push(0x4242424242424242)
        vm.push(0x4141414141414141)
        break
    default:
        fmt.Println("Not understood")
}
}

func push(vm *Stack, words []string) {
    data := words[1]
    var elt uint64
    if strings.HasPrefix(data, "0x") {
        fmt.Sscanf(data, "0x%x", &elt)
    } else {
        fmt.Sscanf(data, "%d", &elt)
    }
    vm.push(elt)
}

func pop(vm *Stack) {
    fmt.Println(pp64(vm.pop(), 0))
}

func plus(vm *Stack) {
    vm.push(vm.pop() + vm.pop())
}

func minus(vm *Stack) {
    first := vm.pop()
    second := vm.pop()
    vm.push(second - first)
}

func and(vm *Stack) {
    vm.push(vm.pop() & vm.pop())
}

```

```

func or(vm *Stack) {
    vm.push(vm.pop() | vm.pop())
}

func shl(vm *Stack) {
    shift := vm.pop()
    data := vm.pop()
    vm.push(data << shift)
}

func shr(vm *Stack) {
    shift := vm.pop()
    data := vm.pop()
    vm.push(data >> shift)
}

func mul(vm *Stack) {
    vm.push(vm.pop() * vm.pop())
}

func xor(vm *Stack) {
    vm.push(vm.pop() ^ vm.pop())
}

func pick(vm *Stack, words []string) {
    var elt int
    fmt.Sscanf(words[1], "%d", &elt)
    vm.push(vm.peek(elt))
}

func rot(vm *Stack) {
    first := vm.pop()
    second := vm.pop()
    third := vm.pop()
    vm.push(first)
    vm.push(third)
    vm.push(second)
}

func swap(vm *Stack) {
    first := vm.pop()
    second := vm.pop()
    vm.push(first)
    vm.push(second)
}

```

```

// parse_shuffle8.go
package main

import (
    "bufio"
    "fmt"
    "os"

```

```

    "strings"
)

const (
    STATE_INIT = iota
    STATE_1_0
    STATE_1_1
    STATE_1_2
    STATE_2_0
    STATE_2_1
    STATE_2_2
    STATE_2_3
    STATE_2_4
    STATE_2_5
    STATE_2_6
    STATE_2_7
    STATE_2_2_0
    STATE_2_2_1
    STATE_2_2_2
    STATE_3_0
    STATE_3_1
    STATE_3_2
    STATE_3_3
    STATE_3_4
    STATE_3_5
    STATE_3_6
    STATE_3_7
    STATE_3_8
    STATE_3_9
    STATE_3_9_1
    STATE_3_9_2
)

var parsed []string
var varIndex int = 0

func main() {
    fp, _ := os.Open(os.Args[1])
    reader := bufio.NewReader(fp)
    state := STATE_INIT
    parsed = make([]string, 6)
    for {
        line, _, err := reader.ReadLine()
        if err != nil || len(line) == 0 {
            return
        }
        fmt.Println(string(line))
        tokens := strings.Fields(string(line))[2:]
        switch state {
        case STATE_INIT:
            state_init(tokens, &state)
            break
        case STATE_1_0:
            state_1_0(tokens, &state)
            break
        case STATE_1_1:

```

```

        state_1_1(tokens, &state)
        break
    case STATE_1_2:
        state_1_2(tokens, &state)
        break
    case STATE_2_0:
        state_2_0(tokens, &state)
        break
    case STATE_2_1:
        state_2_1(tokens, &state)
        break
    case STATE_2_2:
        state_2_2(tokens, &state)
        break
    case STATE_2_3:
        state_2_3(tokens, &state)
        break
    case STATE_2_4:
        state_2_4(tokens, &state)
        break
    case STATE_2_5:
        state_2_5(tokens, &state)
        break
    case STATE_2_6:
        state_2_6(tokens, &state)
        break
    case STATE_2_7:
        state_2_7(tokens, &state)
        break
    case STATE_2_2_1:
        state_2_2_1(tokens, &state)
        break
    case STATE_2_2_2:
        state_2_2_2(tokens, &state)
        break
    case STATE_3_1:
        state_3_1(tokens, &state)
        break
    case STATE_3_2:
        state_3_2(tokens, &state)
        break
    case STATE_3_3:
        state_3_3(tokens, &state)
        break
    case STATE_3_4:
        state_3_4(tokens, &state)
        break
    case STATE_3_5:
        state_3_5(tokens, &state)
        break
    case STATE_3_6:
        state_3_6(tokens, &state)
        break
    case STATE_3_7:
        state_3_7(tokens, &state)
        break

```

```

        case STATE_3_8:
            state_3_8(tokens, &state)
            break
        case STATE_3_9:
            state_3_9(tokens, &state)
            break
        case STATE_3_9_1:
            state_3_9_1(tokens, &state)
            break
        case STATE_3_9_2:
            state_3_9_2(tokens, &state)
            break
        default:
            fmt.Println("What happened ???")
            break
    }
}

func state_init(tokens []string, state *int) {
    switch tokens[0] {
        case "DW_OP_const4u":
            parsed[0] = tokens[1]
            *state = STATE_1_0
            break
        case "DW_OP_dup":
            *state = STATE_2_0
            break
        default:
            *state = STATE_INIT
            fmt.Println("Invalid")
            return
    }
    return
}

func state_1_0(tokens []string, state *int) {
    switch tokens[0] {
        case "DW_OP_plus":
            parsed[1] = "+"
            *state = STATE_1_1
        case "DW_OP_minus":
            parsed[1] = "-"
            *state = STATE_1_1
        case "DW_OP_xor":
            *state = STATE_INIT
            complete_chain_1_1()
            break
        default:
            *state = STATE_INIT
            fmt.Println("Invalid")
            return
    }
    return
}

func state_1_1(tokens []string, state *int) {
    switch tokens[0] {

```

```

        case "DW_OP_const4u":
            if tokens[1] == "0xffffffff" {
                *state = STATE_1_2
            } else {
                *state = STATE_INIT
                fmt.Println("Invalid")
            }
        default:
            *state = STATE_INIT
            fmt.Println("Invalid")
            return
        }
    }
    return
}

func state_1_2(tokens []string, state *int) {
    switch tokens[0] {
        case "DW_OP_and":
            *state = STATE_INIT
            complete_chain_1()
            break
        default:
            *state = STATE_INIT
            fmt.Println("Invalid")
            return
        }
    }
    return
}

func state_2_0(tokens []string, state *int) {
    switch tokens[0] {
        case "DW_OP_const1u":
            *state = STATE_2_1
            parsed[0] = tokens[1]
            break
        case "DW_OP_dup":
            *state = STATE_3_1
            break
        default:
            *state = STATE_INIT
            fmt.Println("Invalid")
            return
        }
    }
    return
}

func state_2_1(tokens []string, state *int) {
    switch tokens[0] {
        case "DW_OP_shl":
            *state = STATE_2_2
            parsed[1] = "<<"
            break
        case "DW_OP_shr":
            *state = STATE_2_2
            parsed[1] = ">>"
            break
        default:
            *state = STATE_INIT
            fmt.Println("Invalid")
    }
}

```

```

        return
    }
    return
}
func state_2_2(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_swap":
        *state = STATE_2_3
        break
    case "DW_OP_xor":
        *state = STATE_INIT
        complete_chain_2_2_1()
        break
    case "DW_OP_plus":
        *state = STATE_2_2_1
        parsed[2] = "+"
        break
    case "DW_OP_minus":
        *state = STATE_2_2_1
        parsed[2] = "-"
        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")
        return
    }
    return
}
func state_2_3(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_const1u":
        *state = STATE_2_4
        parsed[2] = tokens[1]
        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")
        return
    }
    return
}
func state_2_4(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_shl":
        *state = STATE_2_5
        parsed[3] = "<<"
        break
    case "DW_OP_shr":
        *state = STATE_2_5
        parsed[3] = ">>"
        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")
        return
    }
}

```

```

        return
    }
    func state_2_5(tokens []string, state *int) {
        switch tokens[0] {
        case "DW_OP_or":
            *state = STATE_2_6
            break
        default:
            *state = STATE_INIT
            fmt.Println("Invalid")
            return
        }
        return
    }
    func state_2_6(tokens []string, state *int) {
        switch tokens[0] {
        case "DW_OP_const4u":
            *state = STATE_2_7
            break
        default:
            *state = STATE_INIT
            fmt.Println("Invalid")
            return
        }
        return
    }
    func state_2_7(tokens []string, state *int) {
        switch tokens[0] {
        case "DW_OP_and":
            *state = STATE_INIT
            complete_chain_2()
            break
        default:
            *state = STATE_INIT
            fmt.Println("Invalid")
            return
        }
        return
    }
    func state_2_2_1(tokens []string, state *int) {
        switch tokens[0] {
        case "DW_OP_const4u":
            *state = STATE_2_2_2
            break
        default:
            *state = STATE_INIT
            fmt.Println("Invalid")
            return
        }
        return
    }
    func state_2_2_2(tokens []string, state *int) {
        switch tokens[0] {
        case "DW_OP_and":
            *state = STATE_INIT
            complete_chain_2_2_2()

```

```

        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")
        return
    }
    return
}

func state_3_1(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_const1u":
        *state = STATE_3_2
        parsed[0] = tokens[1]
        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")
        return
    }
    return
}

func state_3_2(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_shl":
        *state = STATE_3_3
        parsed[1] = "<<"
        break
    case "DW_OP_shr":
        *state = STATE_3_3
        parsed[1] = ">>"
        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")
        return
    }
    return
}

func state_3_3(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_swap":
        *state = STATE_3_4
        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")
        return
    }
    return
}

func state_3_4(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_const1u":
        *state = STATE_3_5
        parsed[2] = tokens[1]
        break

```

```

        default:
            *state = STATE_INIT
            fmt.Println("Invalid")
            return
        }
    }
    return
}

func state_3_5(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_shl":
        *state = STATE_3_6
        parsed[3] = "<<"
        break
    case "DW_OP_shr":
        *state = STATE_3_6
        parsed[3] = ">>"
        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")
        return
    }
    return
}

func state_3_6(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_or":
        *state = STATE_3_7
        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")
        return
    }
    return
}

func state_3_7(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_const4u":
        *state = STATE_3_8
        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")
        return
    }
    return
}

func state_3_8(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_and":
        *state = STATE_3_9
        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")

```

```

        return
    }
    return
}
func state_3_9(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_xor":
        *state = STATE_INIT
        complete_chain_3()
        break
    case "DW_OP_plus":
        *state = STATE_3_9_1
        parsed[4] = "+"
        break
    case "DW_OP_minus":
        *state = STATE_3_9_1
        parsed[4] = "-"
        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")
        return
    }
    return
}
func state_3_9_1(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_const4u":
        *state = STATE_3_9_2
        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")
        return
    }
    return
}
func state_3_9_2(tokens []string, state *int) {
    switch tokens[0] {
    case "DW_OP_and":
        *state = STATE_INIT
        complete_chain_3_9()
        break
    default:
        *state = STATE_INIT
        fmt.Println("Invalid")
        return
    }
    return
}

func complete_chain_1_1() {
    oldVar := fmt.Sprintf("%t%d", varIndex)
    varIndex++
    newVar := fmt.Sprintf("%t%d", varIndex)
}

```

```

    expr := "%s = %s ^ %s\n"
    fmt.Printf(expr, newVar, oldVar, parsed[0])
}

func complete_chain_1() {
    oldVar := fmt.Sprintf("t%d", varIndex)
    varIndex++
    newVar := fmt.Sprintf("t%d", varIndex)

    expr := "%s = (%s %s %s) & 0xffffffff\n"
    fmt.Printf(expr, newVar, oldVar, parsed[1], parsed[0])
}

func complete_chain_2() {
    oldVar := fmt.Sprintf("t%d", varIndex)
    varIndex++
    newVar := fmt.Sprintf("t%d", varIndex)

    expr := "%s = ((%s %s %s) | (%s %s %s)) & 0xffffffff\n"
    fmt.Printf(expr, newVar, oldVar, parsed[1], parsed[0], oldVar,
        parsed[3], parsed[2])
}

func complete_chain_2_2_1() {
    oldVar := fmt.Sprintf("t%d", varIndex)
    varIndex++
    newVar := fmt.Sprintf("t%d", varIndex)

    expr := "%s = %s ^ (%s %s %s)\n"
    fmt.Printf(expr, newVar, oldVar, oldVar, parsed[1], parsed[0])
}

func complete_chain_2_2_2() {
    oldVar := fmt.Sprintf("t%d", varIndex)
    varIndex++
    newVar := fmt.Sprintf("t%d", varIndex)

    expr := "%s = (%s %s (%s %s %s)) & 0xffffffff\n"
    fmt.Printf(expr, newVar, oldVar, parsed[2], oldVar, parsed[1],
        parsed[0])
}

func complete_chain_3() {
    oldVar := fmt.Sprintf("t%d", varIndex)
    varIndex++
    newVar := fmt.Sprintf("t%d", varIndex)

    expr := "%s = %s ^ (((%s %s %s) | (%s %s %s)) & 0xffffffff)\n"
    fmt.Printf(expr, newVar, oldVar, oldVar, parsed[1], parsed[0],
        oldVar, parsed[3], parsed[2])
}

func complete_chain_3_9() {
    oldVar := fmt.Sprintf("t%d", varIndex)
    varIndex++
    newVar := fmt.Sprintf("t%d", varIndex)

    expr := "%s = (%s %s (((%s %s %s) | (%s %s %s)) & 0xffffffff)) & 0
        xffffffff\n"
    fmt.Printf(expr, newVar, oldVar, parsed[4], oldVar, parsed[1],

```

```

    parsed[0], oldVar, parsed[3], parsed[2])
}

```

```

# solve_step3.py
from binascii import unhexlify
import sys

memory = [0x489dddde, 0x67990f1, 0x95bf74a9, 0x77941ee7, 0xe6d80e3, 0
x2dedaf8b, 0xfb92cd42, 0xd0e867c0, 0xf2b3a3fb, 0x6c39ce47, 0xe74f99e0
, 0x5a24f221, 0xd6378fea, 0xe23ca8c4, 0x84e3b1bc, 0xce5e10bf, 0
xa2b364da, 0x41f250f0, 0xfe97040, 0x1cc05266, 0x16f87e4b, 0x515e26b7,
0xeea48dcb, 0x62b357e4, 0x39bd2041, 0x72cd387a, 0xf37aac8b, 0
x5963b39b, 0x30f75add, 0x103fecbc, 0x392e7a, 0x35df7adf, 0xe19abd13,
0xdaf8b35c, 0xf8798214, 0xb30c2305, 0x67980e9, 0x6900d940, 0x35e876f,
0xa3857014, 0x56c8e162, 0xe9748f56, 0x91d4e409, 0xdcc75a09, 0
xac65f52f, 0x8571dd07, 0x19edcf6, 0x51cef9eb, 0x1eb1b17d, 0xabe446f,
0x3b277cfe, 0x843869bc, 0xb23ea298, 0x7c296f51, 0xcd799972, 0
x62180a64, 0xac052d5, 0xf0076205, 0x13183193, 0xb908cb94, 0x4bf4cd3c,
0xdee5d48a, 0xf64f9a74, 0xd64a15d0, 0xb2cad434, 0x64e9013b, 0
xf46cc1d2, 0x9d78e9db, 0x11789216, 0x335689e6, 0x74c7edb, 0xe6eb6185,
0xd020170b, 0xf304aa15, 0xbcfc2b69e, 0x4eb3d2ea, 0xd78d4d5c, 0
x7ed2bfc4, 0x58ebf0f3, 0x8b591c3f, 0xd3041f6b, 0x5cae88, 0xba696f5c,
0xc16c8ede, 0x9abcb27, 0x56d78d77, 0x765b3e20, 0xcf37212d, 0
x192e2dcf, 0x8caf2806, 0xbc9a575b, 0x776421ce, 0x527fb9eb, 0x69f84340
, 0xadb7bd7, 0x73f2c329, 0x737f8a7f, 0xe301d3e4, 0x57ebeb2, 0
x5859b858, 0x2cc41979, 0xec69a639, 0x53b0d523, 0x39a2f532, 0x8b29e35d
, 0x44e2ce81, 0xcc10a16d, 0x44d9ff58, 0x77102c14, 0xfb57817d, 0
x3cf7c8c8, 0x1222868a, 0x4173d5d1, 0x3529ee32, 0x7a9df58e, 0x513525ac
, 0x81954bac, 0xce53ccf5, 0x79168728, 0xa2d660f8, 0xf30cc9ce, 0
xf0b89c76, 0x89fb3a9, 0xc919dba8, 0x1f9e4dc3, 0xa2594e0c, 0x34ffe178,
0xb04414fb, 0xd31fb33a, 0x184d0278, 0x2c816a9a, 0xb993f2f2, 0
xe4d8601c, 0x49e2eede, 0x9cd50ce1, 0xc03e1e77, 0xa901869e, 0x7579de50
, 0x726ac4ab, 0x38d04840, 0xeabe1270, 0x8c40812d,
    0xe84976b7, 0x172b04ad, 0x756606c4, 0x66258491, 0xb5a0bef8, 0
x6bcc5cf3, 0xa535ae94, 0xc97a87aa, 0x9103a8f6, 0xcc3b9e5f, 0xbb20be1f
, 0xffcfef97, 0x90954f16, 0x501ae1a6, 0x6ed589cd, 0x6826b02b, 0
x565ff263, 0x8e8c369b, 0x6990be7a, 0x3525b840, 0x1847d7bb, 0x355a40c7
, 0xa3579f10, 0xe9edecae, 0xd0337ab1, 0x6355e5ba, 0x88975355, 0
x5ec0f3cf, 0xa0d6213d, 0x75389387, 0xe40216f0, 0xd980cce0, 0x6c88c67c
, 0x829d419c, 0x3bf6451b, 0x11f07bfa, 0xc4c1154e, 0xbd0735eb, 0
x9cf8df9d, 0xe457be75, 0x63a6bd18, 0xefef77fd3, 0x83421b63, 0x7f83072d
, 0x44940f61, 0xf8bdcdf7, 0x61c802ca, 0xa30f9a8, 0x7ff03b37, 0
xa26cc5a9, 0xe10e570d, 0x95ea0c16, 0xa05e6b02, 0xc81d5384, 0x7785db05
, 0x92c84c5f, 0x5584617, 0x82bcfe8d, 0x559ea1da, 0x4fd5cdb0, 0
x9d871fed, 0xdd6f5539, 0x4ed1ef26, 0xfe6813c4, 0x1cfa71d5, 0xd5613aea
, 0xf1c9b8c, 0x2bcac45d, 0x65d00f41, 0x689be0d8, 0x68b01100, 0
x635bd280, 0x954d5d4b, 0x72887f79, 0xce027a75, 0xfcf01c66, 0x6a1bd3,
0x199a1c8e, 0x87d6ee25, 0x938e9f08, 0xd8a11d4d, 0x2b9a4d81, 0
xb6f5d2e5, 0xd15c325a, 0x64eaafc1, 0xfd33b61c, 0x43c1bd57, 0x37b8f048
, 0x5cba7cf2, 0x72810cd0, 0xabfef454, 0xa76384ba, 0xd8861440, 0
x36de5837, 0xf6a03f1, 0x10d48fa1, 0x5883ec2f, 0xa8c00c9b, 0x618ffea4,
    0xa05da206, 0xffb9e97a, 0x8a376781, 0x3156b479, 0xe4af5ecd, 0
x87d9e06f, 0xb4d4d459, 0xeb9a7d25, 0x59dffeaa, 0xdc8bf553, 0xdce3c3a

```

```

, 0x2162970e, 0xe8c9929d, 0x6c3a9bf4, 0x45da5392, 0x9ceee7b0, 0
x3f68d4eb, 0xcd29434f, 0xe4df712, 0xb1a8c69a, 0x1c190f46, 0x2b45873c,
0x46afdfc9, 0x61e8883f, 0x979118c7, 0x70f991b1, 0x1f82604d, 0
xc18bf48f, 0xb327f4ff, 0x519a7508, 0xfa619b0d, 0x268d1490, 0x567e37c2
, 0x25a07691, 0x424359c0, 0x13320c53, 0xe7f742fd, 0x48b945ba, 0
xcfa8e711, 0x8f5fb519, 0x2b7332a5, 0x10aa767c, 0x689f0cc0]
memory6b4 = memory[27:]
memory678 = memory[12:]

def pp_stack(stack):
    for i in range(len(stack)):
        print("{:d} : 0x{:08x} {:08x}".format(
            i, (stack[i] >> 32) & 0xffffffff, stack[i] & 0xffffffff))

def pp_elt(elt):
    print("0x{:08x} {:08x}".format(elt >> 32, elt & 0xffffffff))

def mem8(arr, index):
    return arr[index+1] << 32 | arr[index]

def shift_left_32(word, shift):
    return (word << shift | word >> (32-shift)) & 0xffffffff

def shift_right_32(word, shift):
    return (word >> shift | word << (32-shift)) & 0xffffffff

def pick(stack, index):
    return stack[len(stack)-index-1]

def swap(stack):
    first = stack.pop()
    second = stack.pop()
    stack.append(first)
    stack.append(second)

def rot(stack):
    first = stack.pop()
    second = stack.pop()
    third = stack.pop()
    stack.append(first)
    stack.append(third)
    stack.append(second)

def init(stack):
    stack.append(pick(stack, 3))
    stack.append(pick(stack, 3))
    do_hash(stack)

```

```

def do_hash(stack):
    for i in range(4):
        stack.append(i)

        prepare_step1(stack)
        shuffle(stack)
        stack.append(pick(stack, 4))
        stack.append(pick(stack, 4))
        diversify(stack)

        prepare_step2(stack)
        shuffle(stack)
        stack.append(pick(stack, 7))
        stack.append(pick(stack, 7))
        diversify(stack)

        prepare_next_round(stack)
        stack.append(pick(stack, 3))
        stack.append(pick(stack, 3))
        verif(stack)

def prepare_step1(stack):
    rot(stack)
    stack.append(0)
    stack.append(pick(stack, 5))
    stack.append(pick(stack, 5))

def shuffle(stack):
    s0 = stack.pop()
    s1 = stack.pop()
    for i in range(4):
        v1 = s0 & 0xffffffff
        v2 = s0 >> 32
        v3 = s1 & 0xffffffff
        v4 = s1 >> 32

        t1 = v1
        t2 = v2
        t3 = v4 & v1
        t4 = (v3 ^ (v1 + v4)) & 0xffffffff

        t5 = (t1 - t4) & 0xffffffff
        t4, t5 = (t5, t4)

        t5 = (t5 + memory6b4[t2 & 0xff]) & 0xffffffff
        t6 = t5 ^ (t2 >> 8)
        u1 = (t3 << 32) | t5
        u2 = (t6 << 32) | t4
        s1 = u1
        s0 = u2
    stack.append(s1)
    stack.append(s0)

```

```

def diversify(stack):
    swap(stack)
    stack.append(0)
    rot(stack)
    for i in range(15):
        diversify_step1(stack)
        diversify_prepare(stack)
        diversify_step_prepare_stack(stack)
        diversify_step2(stack)
        diversify_step_prepare_stack(stack)
        diversify_step3(stack)
    t0 = stack.pop()
    t1 = stack.pop()
    stack.pop() # t2
    stack.pop() # t3
    stack.pop() # t4
    s8 = stack.pop()
    stack.append(t0)
    stack.append(t1)

def diversify_step1(stack):
    counter = pick(stack, 2)
    stack.append(counter)
    stack.append(pick(stack, 5))
    stack.append(pick(stack, 5))
    for i in range(counter+1):
        stack.append(i)
        inner_step1(stack)
        shuffle7(stack)
        inner_step3(stack)
        inner_step4(stack)

def inner_step1(stack):
    rot(stack)
    stack.append(pick(stack, 2))

def shuffle7(stack):
    global memory

    c = 0x45786532

    s0 = stack.pop()
    s1 = stack.pop()
    s2 = stack.pop()
    t1 = (s1 >> 32) & 0xffffffff
    t2 = s1 & 0xffffffff
    t3 = (s2 >> 32) & 0xffffffff
    t4 = s2 & 0xffffffff
    t5 = s0

    v2 = memory678[t5] ^ t4

```

```

v1 = (t3 + c) & 0xffffffff
stack.append(t5)
stack.append(t4)
stack.append(t3)
stack.append(t2)
stack.append(t1)
stack.append(v2)
stack.append(v1 ^ t2)
stack.append(shift_left_32(t1, 4))
stack.append(v1)
stack.append((v2-v1) & 0xffffffff)

test = (v1 ^ t2) & 0x80000000
if test == 0:
    # From 0x17246549
    stack.append(0x818f694a)
else:
    # From 0x84653217
    stack.append(0x60bf080f)

def inner_step3(stack):
    # Save five first elements
    s0 = stack.pop()
    s1 = stack.pop()
    s2 = stack.pop()
    s3 = stack.pop()
    s4 = stack.pop()
    # Forget the six others
    stack.pop()
    stack.pop()
    stack.pop()
    stack.pop()
    stack.pop()
    stack.pop()
    s0 = ((s0 ^ s2 ^ s3) << 32) | s1
    s4 = (s3 << 32) | s4
    stack.append(s0)
    stack.append(s4)

def inner_step4(stack):
    s0 = stack.pop()
    s1 = stack.pop()
    s2 = stack.pop()
    s3 = stack.pop()
    stack.append(s3)
    stack.append(s1)
    stack.append(s0)

def diversify_prepare(stack):
    rot(stack)
    rot(stack)
    stack.pop()
    swap(stack)

```

```

third = pick(stack, 3)
stack.append(0)
stack.append(third & 0xffffffff)
stack.append((third >> 32) & 0xffffffff)

def diversify_step_prepare_stack(stack):
    stack.append(pick(stack, 0))
    rot(stack)
    stack.append(0)
    for i in range(6):
        shuffle9(stack)
    # We're done with the loop
    stack.pop()
    stack.pop()
    swap(stack)
    rot(stack)
    rot(stack)
    stack.pop()

def shuffle9(stack):
    # Internal function, does not need to return
    s0 = stack.pop()
    s1 = stack.pop()
    s2 = stack.pop()
    s3 = stack.pop()

    m1 = memory[s0*2]
    m2 = mem8(memory, ((s0*2)+1))

    t1 = ((m1 + s1) & 0xffffffff) ^ s2
    t2 = (((m2 | t1) ^ s3) << 32) & 0xffffffff00000000

    stack.append(t2 >> 32)
    stack.append(t1)
    stack.append(t2 >> 32)
    stack.append(s0+1)

def diversify_step2(stack):
    s0 = stack.pop()
    s1 = stack.pop()
    s2 = stack.pop()
    s3 = stack.pop()
    s4 = stack.pop()

    t1 = (s4 & 0xffffffff)
    t0 = ((s4 & 0xffffffff00000000) >> 32)
    u1 = (shift_left_32(t1 ^ s1, 4) ^ s0)
    u0 = (shift_left_32(t0 ^ s0, 0xe) ^ s1)
    v1 = ((u0 << 32) | u1) ^ s2
    stack.append(v1)
    stack.append(s3)
    stack.append(1)
    stack.append(v1 & 0xffffffff)

```

```

stack.append((v1 & 0xffffffff00000000) >> 32)

def diversify_step3(stack):
    s0 = stack.pop()
    s1 = stack.pop()
    s2 = stack.pop()
    s3 = stack.pop()
    s4 = stack.pop()
    s5 = stack.pop()

    t0 = (s4 & 0xffffffff00000000) >> 32
    t1 = s4 & 0xffffffff
    u0 = (shift_left_32(t0 ^ s0 ^ s1, 0x12))
    u1 = (shift_left_32(t1 ^ s0, 0x1a))
    v1 = ((u0 << 32) | u1) ^ s2
    stack.append(s5+1)
    stack.append(v1)
    stack.append(s3)

def prepare_step2(stack):
    s0 = stack.pop()
    s1 = stack.pop()
    stack.pop() # Forget s2
    stack.pop() # Forget s3
    stack.append(s1)
    stack.append(s0)
    stack.append(1)
    stack.append(s1)
    stack.append(s0)

def prepare_next_round(stack):
    s0 = stack.pop()
    s1 = stack.pop()
    s2 = stack.pop()
    s3 = pick(stack, 0)
    s4 = pick(stack, 1)
    stack.append(s1)
    stack.append(s0)
    stack.append(s3)
    stack.append(s2)

def verif(stack):
    # On vérifie que chacun des paramètres soit égal à la valeur
    attendue
    a0 = stack.pop()
    a1 = stack.pop()
    a2 = stack.pop()
    a3 = stack.pop()
    c0 = 0x658302a68e8e1c24
    c1 = 0xdc7564f1612e5347
    c2 = 0xd9c69b74a86ec613
    c3 = 0x65850b36e76aaed5

```

```

result = ((a0 ^ c0) + (a1 ^ c1) + (a2 ^ c2) +
          (a3 ^ c3)) & 0xffffffffffffffff == 0
if result:
    print("Success")
else:
    print("Not good")
return result

def prepare_step2_recover(stack):
    s0 = stack.pop()
    s1 = stack.pop()
    stack.append(s1)
    stack.append(s0)
    stack.append(1)
    stack.append(s1)
    stack.append(s0)

def recover_step(state, iterations):
    forwarded = [state[0], state[1]]

    prepare_step2_recover(forwarded)
    shuffle(forwarded)
    for i in range(iterations):
        forwarded.append(i)
        inner_step1(forwarded)
        shuffle7(forwarded)
        inner_step3(forwarded)
        inner_step4(forwarded)
    to_shuffle9 = [state[2] >> 32,
                   state[2] & 0xffffffff, state[2] >> 32, 0]
    for i in range(6):
        shuffle9(to_shuffle9)

    s0 = to_shuffle9[2]
    s1 = to_shuffle9[1]
    v1 = state[3]
    s2 = forwarded[-1]
    dec = v1 ^ s2
    u0 = (shift_right_32(dec >> 32, 0x12) ^ s0 ^ s1)
    u1 = (shift_right_32(dec & 0xffffffff, 0x1a) ^ s0)

    new_s3 = u0 << 32 | u1
    to_shuffle9 = [u0, u1, u0, 0]
    for i in range(6):
        shuffle9(to_shuffle9)
    s0 = to_shuffle9[2]
    s1 = to_shuffle9[1]
    s2 = forwarded[-2]
    v1 = state[2]
    dec = v1 ^ s2
    u0 = (shift_right_32((dec >> 32) ^ s1, 0xe) ^ s0)
    u1 = (shift_right_32((dec & 0xffffffff) ^ s0, 0x4) ^ s1)
    new_s2 = u0 << 32 | u1
    state[2] = new_s2

```

```

state[3] = new_s3

def solve():
    expected = [0x65850b36e76aaed5, 0xd9c69b74a86ec613,
                0xdc7564f1612e5347, 0x658302a68e8e1c24]

    state = expected
    for i in range(4):
        for i in range(15, 0, -1):
            recover_step(state, i)

        state = [state[2], state[3], state[0], state[1]]
        for i in range(15, 0, -1):
            recover_step(state, i)
        state = [state[2], state[3], state[0], state[1]]

    print("{:x}{:x}{:x}{:x}".format(state[0], state[1], state[2], state
[3]))
    flag_1 = unhexlify("{:x}".format(state[0]))[::-1]
    flag_2 = unhexlify("{:x}".format(state[1]))[::-1]
    flag_3 = unhexlify("{:x}".format(state[2]))[::-1]
    flag_4 = unhexlify("{:x}".format(state[3]))[::-1]
    print("{}".format(flag_1 + flag_2 + flag_3 + flag_4))

if __name__ == "__main__":
    solve()

```

Annexe D : Programmes pour l'étape 4

La résolution de l'étape 4 s'est appuyée sur beaucoup de programmes différents :

- `extract_blob.go` : Ce programme Go extrait le blob du fichier de départ
- `try_bitflips.py` : Ce script Python lit le blob et compte le nombre de tours de boucles faits par le programme lorsque des inversions de bits sont faites dans le début du blob. Ce genre de test fait paniquer le TEE, auquel cas il faut arrêter qemu.
- `decrypt_flash.go` : Ce programme Go déchiffre les différentes partitions de la flash
- `vm.trace.zip` : Ce fragment de texte contient le fichier `vm.trace.zip` encodé en base64. Il peut être décodé pour être décompressé, ce qui permettra de récupérer la trace utilisée par les programmes suivants
- `draw_image.go` : Ce programme Go parse la trace et en produit une représentation graphique
- `produce_sound.go` : Ce programme Go parse la trace et en produit une représentation acoustique
- `gen_lilypond.go` : Ce programme Go parse la trace et produit un fichier lilypond, qui peut être compilé en une partition de musique.
- `trace_tee_vm.gdb` : Ce script gdb génère une trace de l'exécution de la VM en sauvegardant les registres et l'instruction exécutée à chaque tour
- `aes.go` : Une implémentation d'AES en Go, pour pouvoir déchiffrer les registres de la VM
- `decrypt_state.go` : À partir d'une trace d'exécution (sortie gdb brute) et de la clé, ce programme Go crée un nouveau fichier qui aura déchiffré tous les registres de la VM
- `prog.zip` : Ce fragment de texte contient l'encodage en base64 du fichier `prog.zip`. Ce fichier contient le code de la machine virtuelle déchiffré
- `disass_prog.go` : Ce programme Go lit le code de la machine virtuelle et affiche une version désassemblée

- sm4.go : Ce programme Go contient une implémentation des premiers tours de l'algorithme SM4. L'implémentation initiale a été faite par Suzhou Tongji, que j'ai modifiée.
- solve_step4.go : Ce programme Go contient le code de l'algorithme, le code de déchiffrement du blob et l'algorithme d'inversion. Il résout le challenge.

```
// extract_glob.go
package main

import (
    "io/ioutil"
)

func main() {
    content, _ := ioutil.ReadFile("./decrypted_file")
    startAddr := 0x4dbd8
    length := 0x101010
    blob := content[startAddr : startAddr+length+1]
    ioutil.WriteFile("blob.bin", blob, 0644)
}
```

```
# try_bitflips.py
import ctypes
import fcntl
import binascii
import os
import time

class data_container(ctypes.Structure):
    _fields_ = [
        ('data', ctypes.POINTER(ctypes.c_byte)),
        ('data_len', ctypes.c_uint64),
    ]

def main():
    flips = [0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80]

    with open("blob.bin", "rb") as fp:
        content = fp.read()

    payload = data_container()
    payload.data_len = len(content)
    payload.data = (ctypes.c_byte * len(content)).from_buffer_copy(
        content)

    key = data_container()
    key.data_len = 0x20
    key.data = (ctypes.c_byte * 0x20).from_buffer_copy(b"a"*0x20)

    fd = open('/dev/sstic', 'w')
    # Itère sur les 32 premiers blocs
```

```

for i in range(32):
    # Itère sur les 16 octets de chaque bloc
    for j in range(16):
        # Itère sur les 8 bits de chaque octet
        for k in range(8):
            payload.data[i*16+j] = payload.data[i*16+j] ^ flips[k]
            ret = 0
            count = 0
            try:
                fcntl.ioctl(fd, 0xC0105300, payload)
                fcntl.ioctl(fd, 0xC0105301, key)
                while ret & 0xffff != 0xffff:
                    fcntl.ioctl(fd, 0xC0105302, 0)
                    ret = fcntl.ioctl(fd, 0xC0105303, 0)
                    count += 1
                    if ret & 0xffff != 0:
                        break
            except:
                ret = 0xffff
            finally:
                if ret & 0xffff == 0xffff:
                    status = "Erreur"
                else:
                    status = "Echec"
                print("Bloc {}, bit {} : {} en 0x{:x} tours".format(
                    i, j*8+k, status, count))
                payload.data[i*16+j] = payload.data[i*16+j] ^ flips[
k]

if __name__ == "__main__":
    main()

```

```

// decrypt_flash.go
package main

import (
    "crypto/aes"
    "crypto/cipher"
    "fmt"
    "io/ioutil"
)

func main() {
    content, _ := ioutil.ReadFile("flash.bin")
    cipherLength1 := 0x9430
    cipherLength2 := 0x90a0
    cipherLength3 := 0x5370
    elt := content[0xd8 : 0xd8+cipherLength1]
    iv := elt[:0x10]
    elt = elt[0x10:]
    key := []byte("SSTIC{a947d6980ccf7b87cb8d7c246}")
    fmt.Println(len(elt))
}

```

```

prim, _ := aes.NewCipher(key)
cbc := cipher.NewCBCDecrypter(prim, iv)
cbc.CryptBlocks(elt, elt)
ioutil.WriteFile("flash1.dec", elt, 0644)

elt = content[0xd8+cipherLength1 : 0xd8+cipherLength1+cipherLength2]
iv = elt[:0x10]
elt = elt[0x10:]
key = []byte("SSTIC{a947d6980ccf7b87cb8d7c246}")
fmt.Println(len(elt))
prim, _ = aes.NewCipher(key)
cbc = cipher.NewCBCDecrypter(prim, iv)
cbc.CryptBlocks(elt, elt)
ioutil.WriteFile("flash2.dec", elt, 0644)

elt = content[0xd8+cipherLength1+cipherLength2 : 0xd8+cipherLength1+
cipherLength2+cipherLength3]
iv = elt[:0x10]
elt = elt[0x10:]
key = []byte("SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}")
fmt.Println(len(elt))
prim, _ = aes.NewCipher(key)
cbc = cipher.NewCBCDecrypter(prim, iv)
cbc.CryptBlocks(elt, elt)
ioutil.WriteFile("flash3.dec", elt, 0644)

elt = content[0xd8+cipherLength1+cipherLength2+cipherLength3-0x30:]
iv = elt[:0x10]
elt = elt[0x10:]
key = []byte("SSTIC{a947d6980ccf7b87cb8d7c246}")
fmt.Println(len(elt))
prim, _ = aes.NewCipher(key)
cbc = cipher.NewCBCDecrypter(prim, iv)
cbc.CryptBlocks(elt, elt)
ioutil.WriteFile("flash4.dec", elt, 0644)
}

```

```

-- vm.trace.zip
UESDBBQAAAAIAF0wsE7InvUWIwQAAFIvAQAKABwAdm0udHJhY2UuMlVUCQAD8eDcXPjg3Fx1eAsA

AQQAAAAABAAAAADt3NFyqyAQg0H78zSAGMnpjKjPOMc/IIttciYz7XjGWeS/kkYEql+2
W5JZ8zkb
Y80f8+lrw6Wgy43Vy5nacKlRzhhfD5jUyMfo1w71saHMeVUHs243GesfX3t62tf/9
r3XvuG2jfu
vuG171z7xto31r6xnFpDWfacfp7ycZbF2UFGs3kJ+SiLHHOPkBpTaqzr1vCl4ebS1S+
xdHH3KC/I
Gp171Feck9/DLdIYH2E7TnMoo93TmsIjz2zjt1/sx+0H1/FvdXz/Nf5jeR4//GJ8/zK+q+
Obr/Fn
9zz+/74/6+1p/LGu38v4Th6aG8qZcUk/D3mU3LCpEVD5zvfFWLc9xcqzLHaS+5
hIbhfcKsnUKKeW
6K3PDyqUtS8yYBp5nKDUEqUcBp7vE6QgRXSCEpS+UeoRz79/
GkAEol8iAg94usWjLrkGUXuIwAMe

```

```

PXjKEtaqJlY1Ms0SY79YSJhBQ4QBC1hOT4hBc100YAELWDRiIeEFDREGLGBhBxhE7UQc8IAHPcFg
IUEGOWFE4AFPt3hIoEFOGBF4wAOeFvGQQL9BVJbApiERByxdY+
  ErFaAhwoAFLJfAQsILGiIMWMCi
P+Hd6/rsRXv2ej70iKMPOZPLBW1lgvaiPXtFnrlYjzNSseBDtn72i+
  wgKgeZ8b6m1Y35Yqo08cZr
NErLNb/eCKWuR+uU1H2qB6nWSamhBB7wgIctDxCdiQg840kWj7pOGkTtIQIPePTgKUtgB5+
  EGTRE
GLCARUVCDJrrogELWMCiEQsJL2iIMGABCzvAIGon4oAHP0A5AQ8JMogOIwIPeLrFQwINosOIwAMe
8
  LSIhwT6DaKyBDYnITHg6RoLX6kADREGLGC5BBYSXtAQYcACFv0JL3WGqDPUORtPzUYoRWFap6Tu
Uz1ItU5KDSXwgAc8bHmA6ExE4AFPt3jUpdMgag8ReMCjB09ZAjv4JMygIcKABSwqEmLQXBcNWMAC
Fo1YSHhBQ4QBC1jYAQZR0xEHP0ABzwl4SJBBDBgReMDTLR4SaBAdRgQe8ICnRTwk0G8QlSWwaUjE
AUvXWPhKBWiIMGAByyWwkPCChggDFrDoT3ipM0SdoY7eeGo2QikK0zoldZ/
  qQap1UlCC0tUo9YiH
DRAQHUYEHvBOiOddcg2i9hCBBzx68JQlsJ9PwgwaIgxYwKIiIQbNddGABSxg0YiFhBcORBiwgIUd
YBC1E3HAAx7wnICHBB1EhxGBBzdz4iGBBtFhROABD3haxEMC/
  QZRWQKbkhQcsHSNha9UgIYIAxaw
XAILCS9oiDBgAYv+hFdD1aFVagz52kg3e6vvM6RrS6EfucTLGF0UJ+YnIy8IkaEu0jOn+7
  BybT6G
/MJUV2S3hppyLtY7GW24MxETMRETmRETmRETmRETmRET6Z9Iyrc+xu0/
  tr9QSwECHgMUAACABd
MLBOyJ71FiMEAABSLwEACgAYAAAAAABAAAA/4
  EAAAAAdmOudHJhY2UuMlVUBQAD8eDcXHV4CwAB
BAAAAAAEAAAAAFBLBQYAAAAAAQABAFAAAAABnBAAAAAA=

```

```

// draw_image.go
package main

import (
    "bufio"
    "fmt"
    "image"
    "image/color"
    "image/png"
    "os"
)

const LINES = 257
const FILENAME = "trace_257.png"

var BLACK = color.RGBA{0, 0, 0, 255}

var kellyPalette []color.Color = []color.Color{
    color.RGBA{0xF6, 0xA6, 0x00, 0xff},
    color.RGBA{0x87, 0x56, 0x92, 0xff},
    color.RGBA{0xF3, 0x84, 0x00, 0xff},
    color.RGBA{0xA1, 0xCA, 0xF1, 0xff},
}

```

```

    color.RGBA{0xBE, 0x00, 0x32, 0xff},
    color.RGBA{0xF3, 0xC3, 0x00, 0xff},
    color.RGBA{0x84, 0x84, 0x82, 0xff},
    color.RGBA{0x00, 0x88, 0x56, 0xff},
    color.RGBA{0xE6, 0x8F, 0xAC, 0xff},
    color.RGBA{0x00, 0x67, 0xA5, 0xff},
    color.RGBA{0xF9, 0x93, 0x79, 0xff},
    color.RGBA{0x60, 0x4E, 0x97, 0xff},
    color.RGBA{0x00, 0x00, 0xff, 0xff},
    color.RGBA{0xB3, 0x44, 0x6C, 0xff},
    color.RGBA{0xDC, 0xD3, 0x00, 0xff},
    color.RGBA{0x88, 0x2D, 0x17, 0xff},
    color.RGBA{0x8D, 0xB6, 0x00, 0xff},
    color.RGBA{0x65, 0x45, 0x22, 0xff},
    color.RGBA{0xE2, 0x58, 0x22, 0xff},
    color.RGBA{0x2B, 0x3D, 0x26, 0xff},
    color.RGBA{0xF2, 0xF3, 0xF4, 0xff},
    color.RGBA{0x22, 0x22, 0x22, 0xff},
}
var stateFPRegs []uint32
var state4a618 []uint32

type instruction struct {
    opcode uint8
    flags  uint8
    arg1   uint8
    arg2   uint8
    arg3   uint16
}

func (ins *instruction) String() string {
    return fmt.Sprintf("opcode: 0x%x, flags: 0b%02b, arg1: 0x%x, arg2: 0x%x, arg3: 0x%03x", ins.opcode, ins.flags, ins.arg1, ins.arg2, ins.arg3)
}

func decode(encoded uint) *instruction {
    op := uint8(encoded >> 0x14 & 0xf)
    flags := uint8(encoded >> 0x12 & 3)
    arg1 := uint8(encoded >> 0xe & 0xf)
    arg2 := uint8(encoded >> 0xa & 0xf)
    arg3 := uint16(encoded & 0x3ff)
    return &instruction{op, flags, arg1, arg2, arg3}
}

func (instr *instruction) draw() []color.Color {
    return []color.Color{
        kellyPalette[instr.opcode],
        kellyPalette[instr.flags],
        kellyPalette[instr.arg1],
        kellyPalette[instr.arg2],
    }
}

func drawInstruction(im *image.RGBA, instr *instruction, pos int) {
    x := pos % LINES

```

```

    y := pos / LINES * 12
    colorInstr := instr.draw()
    im.Set(y+0, x, colorInstr[0])
    im.Set(y+1, x, colorInstr[0])
    im.Set(y+2, x, colorInstr[0])
    im.Set(y+3, x, colorInstr[1])
    im.Set(y+4, x, colorInstr[2])
    im.Set(y+5, x, colorInstr[2])
    im.Set(y+6, x, colorInstr[2])
    im.Set(y+7, x, colorInstr[3])
    im.Set(y+8, x, colorInstr[3])
    im.Set(y+9, x, colorInstr[3])
}

func main() {
    program := make([]*instruction, 0)
    var instr uint
    fp, _ := os.Open("vm.trace")
    scanner := bufio.NewScanner(fp)
    for scanner.Scan() {
        fmt.Sscanf(scanner.Text(), "0x%x", &instr)
        program = append(program, decode(instr))
    }
    columns := ((len(program) / LINES) + 1) * 12
    rect := image.Rect(0, 0, columns, LINES)
    im := image.NewRGBA(rect)
    dx := rect.Dx()
    dy := rect.Dy()
    for x := 0; x < dx; x++ {
        for y := 0; y < dy; y++ {
            im.Set(x, y, BLACK)
        }
    }
    for i, instr := range program[7:] {
        drawInstruction(im, instr, i)
    }

    _ = os.Remove(FILENAME)
    wr, _ := os.Create(FILENAME)
    png.Encode(wr, im)
}

```

```

// produce_sound.go

package main

import (
    "bufio"
    "math"
    "fmt"
    "os"
)

```

```

const sampleRate = 44100
# Durée d'une croche en millisecondes
const tempo = 10

var freqs []float64 = []float64{
    261.6,
    277.2,
    293.7,
    311.1,
    329.6,
    349.2,
    370.0,
    392.0,
    415.3,
    440.0,
    466.2,
    493.9,
    523.3,
    554.4,
    587.3,
    622.3,
    659.3,
    698.5,
    740.0,
    784.0,
    830.6,
    880.0,
    932.3,
    987.8,
}

type soundWriter struct {
    sampleRate int
    wr          io.Writer
    samples     []byte
}

func (snd *soundWriter) Write(freq float64, durationMs int) {
    nbSamples := durationMs * snd.sampleRate / 1000
    for i := 0; i < nbSamples; i++ {
        dt := float64(i) / float64(snd.sampleRate)
        sample := int16(math.Sin(dt*2*math.Pi*freq) * 32272)
        snd.samples = append(snd.samples, byte(sample>>8), byte(sample&0
xff))
    }
}

func (snd *soundWriter) Dump() {
    snd.wr.Write(snd.samples)
}

type instruction struct {
    opcode uint8
    flags  uint8
    arg1   uint8

```

```

    arg2    uint8
    arg3    uint16
}

func (ins *instruction) String() string {
    return fmt.Sprintf("opcode: 0x%x, flags: 0b%02b, arg1: 0x%x, arg2: 0x%x, arg3: 0x%03x", ins.opcode, ins.flags, ins.arg1, ins.arg2, ins.arg3)
}

func decode(encoded uint) *instruction {
    op := uint8(encoded >> 0x14 & 0xf)
    flags := uint8(encoded >> 0x12 & 3)
    arg1 := uint8(encoded >> 0xe & 0xf)
    arg2 := uint8(encoded >> 0xa & 0xf)
    arg3 := uint16(encoded & 0x3ff)
    return &instruction{op, flags, arg1, arg2, arg3}
}

func (instr *instruction) freq() []float64 {
    return []float64{
        freqs[instr.opcode],
        freqs[instr.flags],
        freqs[instr.arg1],
        freqs[instr.arg2],
    }
}

func synthesizeInstruction(snd *soundWriter, instr *instruction) {
    freq := instr.freq()
    snd.Write(freq[0], tempo+tempo/2)
    snd.Write(freq[1], tempo/2)
    snd.Write(freq[2], tempo)
    snd.Write(freq[3], tempo)
}

func main() {
    program := make([]*instruction, 0)
    var instr uint
    fp, _ := os.Open("vm.trace")
    scanner := bufio.NewScanner(fp)
    for scanner.Scan() {
        fmt.Sscanf(scanner.Text(), "0x%x", &instr)
        program = append(program, decode(instr))
    }

    _ = os.Remove("trace.raw")
    wr, _ := os.Create("trace.raw")
    snd := &soundWriter{sampleRate, wr, make([]byte, 0)}

    for i, instr := range program {
        synthesizeInstruction(snd, instr)
        if i%500 == 0 {
            fmt.Println(i)
        }
    }
}

```

```
    snd.Dump()
}
```

```
// gen_lilypond.go
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
)

const SAMPLERATE = 44100
const tempo = 10
const BUFLIMIT = 1024 * 1024
const nbInstruction = 30

var lilyHdr = []byte("\\version \"2.18.2\"\\n{\\n")

var notes []string = []string{
    "c'",
    "d'",
    "e'",
    "f'",
    "g'",
    "a'",
    "b'",
    "c'",
    "d'",
    "e'",
    "f'",
    "g'",
    "a'",
    "b'",
    "c'",
    "d'",
    "e'",
    "f'",
    "g'",
}

type lilyWriter struct {
    wr      io.Writer
    buffer []byte
}

func newWriter(wr io.Writer) *lilyWriter {
    lily := &lilyWriter{wr, make([]byte, 0)}
    lily.wr.Write(lilyHdr)
    return lily
}
```

```

func (lily *lilyWriter) Write(data []byte) {
    lily.buffer = append(lily.buffer, data...)
    if len(lily.buffer) > BUFLIMIT {
        lily.wr.Write(lily.buffer)
        lily.buffer = make([]byte, 0)
    }
}

func (lily *lilyWriter) Close() {
    lily.wr.Write(lily.buffer)
    lily.wr.Write([]byte("{}\n"))
}

type instruction struct {
    opcode uint8
    flags  uint8
    arg1   uint8
    arg2   uint8
    arg3   uint16
}

func (ins *instruction) String() string {
    return fmt.Sprintf("opcode: 0x%x, flags: 0b%02b, arg1: 0x%x, arg2: 0x%x, arg3: 0x%03x", ins.opcode, ins.flags, ins.arg1, ins.arg2, ins.arg3)
}

func decode(encoded uint) *instruction {
    op := uint8(encoded >> 0x14 & 0xf)
    flags := uint8(encoded >> 0x12 & 3)
    arg1 := uint8(encoded >> 0xe & 0xf)
    arg2 := uint8(encoded >> 0xa & 0xf)
    arg3 := uint16(encoded & 0x3ff)
    return &instruction{op, flags, arg1, arg2, arg3}
}

func (instr *instruction) notes() []string {
    return []string{
        notes[instr.opcode],
        notes[instr.flags],
        notes[instr.arg1],
        notes[instr.arg2],
    }
}

func synthesizeInstruction(lily *lilyWriter, instr *instruction) {
    note := instr.notes()
    lily.Write([]byte(note[0] + "8."))
    lily.Write([]byte("[ "))
    lily.Write([]byte(note[1] + "16"))
    lily.Write([]byte(" "))
    lily.Write([]byte(note[2] + "8"))
    lily.Write([]byte(" "))
    lily.Write([]byte(note[3] + "8"))
    lily.Write([]byte("]"))
}

```

```

    lily.Write([]byte("\n"))
}

func main() {
    program := make([]*instruction, 0)
    var instr uint
    fp, _ := os.Open("vm.trace")
    scanner := bufio.NewScanner(fp)
    for scanner.Scan() {
        fmt.Sscanf(scanner.Text(), "0x%x", &instr)
        program = append(program, decode(instr))
    }

    _ = os.Remove("sheet.ly")
    wr, _ := os.Create("sheet.ly")
    lily := newWriter(wr)

    for _, instr := range program[:nbInstruction] {
        synthesizeInstruction(lily, instr)
    }
    lily.Close()
}

```

```

// trace_vm_tee.gdb
target remote localhost:1234
set pagination off

dprintf *0x0e00b000, "Booting BL1\n"

break *0x39ac
dprintf *0x0e030000, "Booting BL2\n"
c
si

# Affiche les registres chiffrés de la VM
break *0x00000000e031038
command 4
x/32gx 0x0e04a5d8
disable 4
c
end

disable 4

b *0x0e032028

dprintf *0x0e200000, "Booting Secure-OS\n"
break *0x0e200004

c

```

```

dprintf *0x0e200f20, "Decrypting 0x%08x%08x%08x%08x with 0x%08x%08x%08x
%08x: \n", $v0.s.u[3], $v0.s.u[2], $v0.s.u[1], $v0.s.u[0], $v1.s.u
[3], $v1.s.u[2], $v1.s.u[1], $v1.s.u[0]
# Dump Decryption result
dprintf *0x0e200f2c, "Result 0x%02x%02x%02x%02x%02x%02x%02x%02x%02x
%02x%02x%02x%02x%02x%02x\n", $v0.b.u[12], $v0.b.u[13], $v0.b.u[14],
$v0.b.u[15], $v0.b.u[8], $v0.b.u[9], $v0.b.u[10], $v0.b.u[11], $v0.b.
u[4], $v0.b.u[5], $v0.b.u[6], $v0.b.u[7], $v0.b.u[0], $v0.b.u[1], $v0
.b.u[2], $v0.b.u[3]

# Affiche l'instruction courante
break *0x000000000e2005a4
command 10
printf "instruction: 0x%x\n", $x0
enable 4
# Active le point d'arrêt 13 pour qu'au prochain passage dans 0x0e031034
, les registres soient affichés
c
end

dprintf *0xffff000008590198, "Sending blob\n"
dprintf *0xffff000008590518, "Sending key nibble\n"
dprintf *0xffff000008590414, "Exec instr\n"
dprintf *0xffff00000859022c, "Decrypt instr\n"

c

```

```

// aes.go
package main

// Pre-computed multiplication by 2 in GF(256)
var gmul2 = [256]byte{
    0x00, 0x02, 0x04, 0x06, 0x08, 0x0A, 0x0C, 0x0E, 0x10, 0x12, 0x14, 0
    x16, 0x18, 0x1A, 0x1C, 0x1E,
    0x20, 0x22, 0x24, 0x26, 0x28, 0x2A, 0x2C, 0x2E, 0x30, 0x32, 0x34, 0
    x36, 0x38, 0x3A, 0x3C, 0x3E,
    0x40, 0x42, 0x44, 0x46, 0x48, 0x4A, 0x4C, 0x4E, 0x50, 0x52, 0x54, 0
    x56, 0x58, 0x5A, 0x5C, 0x5E,
    0x60, 0x62, 0x64, 0x66, 0x68, 0x6A, 0x6C, 0x6E, 0x70, 0x72, 0x74, 0
    x76, 0x78, 0x7A, 0x7C, 0x7E,
    0x80, 0x82, 0x84, 0x86, 0x88, 0x8A, 0x8C, 0x8E, 0x90, 0x92, 0x94, 0
    x96, 0x98, 0x9A, 0x9C, 0x9E,
    0xA0, 0xA2, 0xA4, 0xA6, 0xA8, 0xAA, 0xAC, 0xAE, 0xB0, 0xB2, 0xB4, 0
    xB6, 0xB8, 0xBA, 0xBC, 0xBE,
    0xC0, 0xC2, 0xC4, 0xC6, 0xC8, 0xCA, 0xCC, 0xCE, 0xD0, 0xD2, 0xD4, 0
    xD6, 0xD8, 0xDA, 0xDC, 0xDE,
    0xE0, 0xE2, 0xE4, 0xE6, 0xE8, 0xEA, 0xEC, 0xEE, 0xF0, 0xF2, 0xF4, 0
    xF6, 0xF8, 0xFA, 0xFC, 0xFE,
    0x1B, 0x19, 0x1F, 0x1D, 0x13, 0x11, 0x17, 0x15, 0x0B, 0x09, 0x0F, 0
    x0D, 0x03, 0x01, 0x07, 0x05,
    0x3B, 0x39, 0x3F, 0x3D, 0x33, 0x31, 0x37, 0x35, 0x2B, 0x29, 0x2F, 0
    x2D, 0x23, 0x21, 0x27, 0x25,

```

```

    0x5B, 0x59, 0x5F, 0x5D, 0x53, 0x51, 0x57, 0x55, 0x4B, 0x49, 0x4F, 0
    x4D, 0x43, 0x41, 0x47, 0x45,
    0x7B, 0x79, 0x7F, 0x7D, 0x73, 0x71, 0x77, 0x75, 0x6B, 0x69, 0x6F, 0
    x6D, 0x63, 0x61, 0x67, 0x65,
    0x9B, 0x99, 0x9F, 0x9D, 0x93, 0x91, 0x97, 0x95, 0x8B, 0x89, 0x8F, 0
    x8D, 0x83, 0x81, 0x87, 0x85,
    0xBB, 0xB9, 0xBF, 0xBD, 0xB3, 0xB1, 0xB7, 0xB5, 0xAB, 0xA9, 0xAF, 0
    xAD, 0xA3, 0xA1, 0xA7, 0xA5,
    0xDB, 0xD9, 0xDF, 0xDD, 0xD3, 0xD1, 0xD7, 0xD5, 0xCB, 0xC9, 0xCF, 0
    xCD, 0xC3, 0xC1, 0xC7, 0xC5,
    0xFB, 0xF9, 0xFF, 0xFD, 0xF3, 0xF1, 0xF7, 0xF5, 0xEB, 0xE9, 0xEF, 0
    xED, 0xE3, 0xE1, 0xE7, 0xE5,
}

// Pre-computed multiplication by 3 in GF(256)
var gmul3 = [256]byte{
    0x00, 0x03, 0x06, 0x05, 0x0C, 0x0F, 0x0A, 0x09, 0x18, 0x1B, 0x1E, 0
    x1D, 0x14, 0x17, 0x12, 0x11,
    0x30, 0x33, 0x36, 0x35, 0x3C, 0x3F, 0x3A, 0x39, 0x28, 0x2B, 0x2E, 0
    x2D, 0x24, 0x27, 0x22, 0x21,
    0x60, 0x63, 0x66, 0x65, 0x6C, 0x6F, 0x6A, 0x69, 0x78, 0x7B, 0x7E, 0
    x7D, 0x74, 0x77, 0x72, 0x71,
    0x50, 0x53, 0x56, 0x55, 0x5C, 0x5F, 0x5A, 0x59, 0x48, 0x4B, 0x4E, 0
    x4D, 0x44, 0x47, 0x42, 0x41,
    0xC0, 0xC3, 0xC6, 0xC5, 0xCC, 0xCF, 0xCA, 0xC9, 0xD8, 0xDB, 0xDE, 0
    xDD, 0xD4, 0xD7, 0xD2, 0xD1,
    0xF0, 0xF3, 0xF6, 0xF5, 0xFC, 0xFF, 0xFA, 0xF9, 0xE8, 0xEB, 0xEE, 0
    xED, 0xE4, 0xE7, 0xE2, 0xE1,
    0xA0, 0xA3, 0xA6, 0xA5, 0xAC, 0xAF, 0xAA, 0xA9, 0xB8, 0xBB, 0xBE, 0
    xBD, 0xB4, 0xB7, 0xB2, 0xB1,
    0x90, 0x93, 0x96, 0x95, 0x9C, 0x9F, 0x9A, 0x99, 0x88, 0x8B, 0x8E, 0
    x8D, 0x84, 0x87, 0x82, 0x81,
    0x9B, 0x98, 0x9D, 0x9E, 0x97, 0x94, 0x91, 0x92, 0x83, 0x80, 0x85, 0
    x86, 0x8F, 0x8C, 0x89, 0x8A,
    0xAB, 0xA8, 0xAD, 0xAE, 0xA7, 0xA4, 0xA1, 0xA2, 0xB3, 0xB0, 0xB5, 0
    xB6, 0xBF, 0xBC, 0xB9, 0xBA,
    0xFB, 0xF8, 0xFD, 0xFE, 0xF7, 0xF4, 0xF1, 0xF2, 0xE3, 0xE0, 0xE5, 0
    xE6, 0xEF, 0xEC, 0xE9, 0xEA,
    0xCB, 0xC8, 0xCD, 0xCE, 0xC7, 0xC4, 0xC1, 0xC2, 0xD3, 0xD0, 0xD5, 0
    xD6, 0xDF, 0xDC, 0xD9, 0xDA,
    0x5B, 0x58, 0x5D, 0x5E, 0x57, 0x54, 0x51, 0x52, 0x43, 0x40, 0x45, 0
    x46, 0x4F, 0x4C, 0x49, 0x4A,
    0x6B, 0x68, 0x6D, 0x6E, 0x67, 0x64, 0x61, 0x62, 0x73, 0x70, 0x75, 0
    x76, 0x7F, 0x7C, 0x79, 0x7A,
    0x3B, 0x38, 0x3D, 0x3E, 0x37, 0x34, 0x31, 0x32, 0x23, 0x20, 0x25, 0
    x26, 0x2F, 0x2C, 0x29, 0x2A,
    0x0B, 0x08, 0x0D, 0x0E, 0x07, 0x04, 0x01, 0x02, 0x13, 0x10, 0x15, 0
    x16, 0x1F, 0x1C, 0x19, 0x1A,
}

// Pre-computed multiplication by 9 in GF(256)
var gmul9 = [256]byte{
    0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F, 0x48, 0x41, 0x5A, 0
    x53, 0x6C, 0x65, 0x7E, 0x77,
    0x90, 0x99, 0x82, 0x8B, 0xB4, 0xBD, 0xA6, 0xAF, 0xD8, 0xD1, 0xCA, 0
    xC3, 0xFC, 0xF5, 0xEE, 0xE7,

```

```

    0x3B, 0x32, 0x29, 0x20, 0x1F, 0x16, 0x0D, 0x04, 0x73, 0x7A, 0x61, 0
    x68, 0x57, 0x5E, 0x45, 0x4C,
    0xAB, 0xA2, 0xB9, 0xB0, 0x8F, 0x86, 0x9D, 0x94, 0xE3, 0xEA, 0xF1, 0
    xF8, 0xC7, 0xCE, 0xD5, 0xDC,
    0x76, 0x7F, 0x64, 0x6D, 0x52, 0x5B, 0x40, 0x49, 0x3E, 0x37, 0x2C, 0
    x25, 0x1A, 0x13, 0x08, 0x01,
    0xE6, 0xEF, 0xF4, 0xFD, 0xC2, 0xCB, 0xD0, 0xD9, 0xAE, 0xA7, 0xBC, 0
    xB5, 0x8A, 0x83, 0x98, 0x91,
    0x4D, 0x44, 0x5F, 0x56, 0x69, 0x60, 0x7B, 0x72, 0x05, 0x0C, 0x17, 0
    x1E, 0x21, 0x28, 0x33, 0x3A,
    0xDD, 0xD4, 0xCF, 0xC6, 0xF9, 0xF0, 0xEB, 0xE2, 0x95, 0x9C, 0x87, 0
    x8E, 0xB1, 0xB8, 0xA3, 0xAA,
    0xEC, 0xE5, 0xFE, 0xF7, 0xC8, 0xC1, 0xDA, 0xD3, 0xA4, 0xAD, 0xB6, 0
    xBF, 0x80, 0x89, 0x92, 0x9B,
    0x7C, 0x75, 0x6E, 0x67, 0x58, 0x51, 0x4A, 0x43, 0x34, 0x3D, 0x26, 0
    x2F, 0x10, 0x19, 0x02, 0x0B,
    0xD7, 0xDE, 0xC5, 0xCC, 0xF3, 0xFA, 0xE1, 0xE8, 0x9F, 0x96, 0x8D, 0
    x84, 0xBB, 0xB2, 0xA9, 0xA0,
    0x47, 0x4E, 0x55, 0x5C, 0x63, 0x6A, 0x71, 0x78, 0x0F, 0x06, 0x1D, 0
    x14, 0x2B, 0x22, 0x39, 0x30,
    0x9A, 0x93, 0x88, 0x81, 0xBE, 0xB7, 0xAC, 0xA5, 0xD2, 0xDB, 0xC0, 0
    xC9, 0xF6, 0xFF, 0xE4, 0xED,
    0x0A, 0x03, 0x18, 0x11, 0x2E, 0x27, 0x3C, 0x35, 0x42, 0x4B, 0x50, 0
    x59, 0x66, 0x6F, 0x74, 0x7D,
    0xA1, 0xA8, 0xB3, 0xBA, 0x85, 0x8C, 0x97, 0x9E, 0xE9, 0xE0, 0xFB, 0
    xF2, 0xCD, 0xC4, 0xDF, 0xD6,
    0x31, 0x38, 0x23, 0x2A, 0x15, 0x1C, 0x07, 0x0E, 0x79, 0x70, 0x6B, 0
    x62, 0x5D, 0x54, 0x4F, 0x46,
}

// Pre-computed multiplication by 11 in GF(256)
var gmul11 = [256]byte{
    0x00, 0x0B, 0x16, 0x1D, 0x2C, 0x27, 0x3A, 0x31, 0x58, 0x53, 0x4E, 0
    x45, 0x74, 0x7F, 0x62, 0x69,
    0xB0, 0xBB, 0xA6, 0xAD, 0x9C, 0x97, 0x8A, 0x81, 0xE8, 0xE3, 0xFE, 0
    xF5, 0xC4, 0xCF, 0xD2, 0xD9,
    0x7B, 0x70, 0x6D, 0x66, 0x57, 0x5C, 0x41, 0x4A, 0x23, 0x28, 0x35, 0
    x3E, 0x0F, 0x04, 0x19, 0x12,
    0xCB, 0xC0, 0xDD, 0xD6, 0xE7, 0xEC, 0xF1, 0xFA, 0x93, 0x98, 0x85, 0
    x8E, 0xBF, 0xB4, 0xA9, 0xA2,
    0xF6, 0xFD, 0xE0, 0xEB, 0xDA, 0xD1, 0xCC, 0xC7, 0xAE, 0xA5, 0xB8, 0
    xB3, 0x82, 0x89, 0x94, 0x9F,
    0x46, 0x4D, 0x50, 0x5B, 0x6A, 0x61, 0x7C, 0x77, 0x1E, 0x15, 0x08, 0
    x03, 0x32, 0x39, 0x24, 0x2F,
    0x8D, 0x86, 0x9B, 0x90, 0xA1, 0xAA, 0xB7, 0xBC, 0xD5, 0xDE, 0xC3, 0
    xC8, 0xF9, 0xF2, 0xEF, 0xE4,
    0x3D, 0x36, 0x2B, 0x20, 0x11, 0x1A, 0x07, 0x0C, 0x65, 0x6E, 0x73, 0
    x78, 0x49, 0x42, 0x5F, 0x54,
    0xF7, 0xFC, 0xE1, 0xEA, 0xDB, 0xD0, 0xCD, 0xC6, 0xAF, 0xA4, 0xB9, 0
    xB2, 0x83, 0x88, 0x95, 0x9E,
    0x47, 0x4C, 0x51, 0x5A, 0x6B, 0x60, 0x7D, 0x76, 0x1F, 0x14, 0x09, 0
    x02, 0x33, 0x38, 0x25, 0x2E,
    0x8C, 0x87, 0x9A, 0x91, 0xA0, 0xAB, 0xB6, 0xBD, 0xD4, 0xDF, 0xC2, 0
    xC9, 0xF8, 0xF3, 0xEE, 0xE5,
    0x3C, 0x37, 0x2A, 0x21, 0x10, 0x1B, 0x06, 0x0D, 0x64, 0x6F, 0x72, 0
    x79, 0x48, 0x43, 0x5E, 0x55,

```

```

    0x01, 0x0A, 0x17, 0x1C, 0x2D, 0x26, 0x3B, 0x30, 0x59, 0x52, 0x4F, 0
    x44, 0x75, 0x7E, 0x63, 0x68,
    0xB1, 0xBA, 0xA7, 0xAC, 0x9D, 0x96, 0x8B, 0x80, 0xE9, 0xE2, 0xFF, 0
    xF4, 0xC5, 0xCE, 0xD3, 0xD8,
    0x7A, 0x71, 0x6C, 0x67, 0x56, 0x5D, 0x40, 0x4B, 0x22, 0x29, 0x34, 0
    x3F, 0x0E, 0x05, 0x18, 0x13,
    0xCA, 0xC1, 0xDC, 0xD7, 0xE6, 0xED, 0xF0, 0xFB, 0x92, 0x99, 0x84, 0
    x8F, 0xBE, 0xB5, 0xA8, 0xA3,
}

// Pre-computed multiplication by 13 in GF(256)
var gmul13 = [256]byte{
    0x00, 0x0D, 0x1A, 0x17, 0x34, 0x39, 0x2E, 0x23, 0x68, 0x65, 0x72, 0
    x7F, 0x5C, 0x51, 0x46, 0x4B,
    0xD0, 0xDD, 0xCA, 0xC7, 0xE4, 0xE9, 0xFE, 0xF3, 0xB8, 0xB5, 0xA2, 0
    xAF, 0x8C, 0x81, 0x96, 0x9B,
    0xBB, 0xB6, 0xA1, 0xAC, 0x8F, 0x82, 0x95, 0x98, 0xD3, 0xDE, 0xC9, 0
    xC4, 0xE7, 0xEA, 0xFD, 0xF0,
    0x6B, 0x66, 0x71, 0x7C, 0x5F, 0x52, 0x45, 0x48, 0x03, 0x0E, 0x19, 0
    x14, 0x37, 0x3A, 0x2D, 0x20,
    0x6D, 0x60, 0x77, 0x7A, 0x59, 0x54, 0x43, 0x4E, 0x05, 0x08, 0x1F, 0
    x12, 0x31, 0x3C, 0x2B, 0x26,
    0xBD, 0xB0, 0xA7, 0xAA, 0x89, 0x84, 0x93, 0x9E, 0xD5, 0xD8, 0xCF, 0
    xC2, 0xE1, 0xEC, 0xFB, 0xF6,
    0xD6, 0xDB, 0xCC, 0xC1, 0xE2, 0xEF, 0xF8, 0xF5, 0xBE, 0xB3, 0xA4, 0
    xA9, 0x8A, 0x87, 0x90, 0x9D,
    0x06, 0x0B, 0x1C, 0x11, 0x32, 0x3F, 0x28, 0x25, 0x6E, 0x63, 0x74, 0
    x79, 0x5A, 0x57, 0x40, 0x4D,
    0xDA, 0xD7, 0xC0, 0xCD, 0xEE, 0xE3, 0xF4, 0xF9, 0xB2, 0xBF, 0xA8, 0
    xA5, 0x86, 0x8B, 0x9C, 0x91,
    0x0A, 0x07, 0x10, 0x1D, 0x3E, 0x33, 0x24, 0x29, 0x62, 0x6F, 0x78, 0
    x75, 0x56, 0x5B, 0x4C, 0x41,
    0x61, 0x6C, 0x7B, 0x76, 0x55, 0x58, 0x4F, 0x42, 0x09, 0x04, 0x13, 0
    x1E, 0x3D, 0x30, 0x27, 0x2A,
    0xB1, 0xBC, 0xAB, 0xA6, 0x85, 0x88, 0x9F, 0x92, 0xD9, 0xD4, 0xC3, 0
    xCE, 0xED, 0xE0, 0xF7, 0xFA,
    0xB7, 0xBA, 0xAD, 0xA0, 0x83, 0x8E, 0x99, 0x94, 0xDF, 0xD2, 0xC5, 0
    xC8, 0xEB, 0xE6, 0xF1, 0xFC,
    0x67, 0x6A, 0x7D, 0x70, 0x53, 0x5E, 0x49, 0x44, 0x0F, 0x02, 0x15, 0
    x18, 0x3B, 0x36, 0x21, 0x2C,
    0x0C, 0x01, 0x16, 0x1B, 0x38, 0x35, 0x22, 0x2F, 0x64, 0x69, 0x7E, 0
    x73, 0x50, 0x5D, 0x4A, 0x47,
    0xDC, 0xD1, 0xC6, 0xCB, 0xE8, 0xE5, 0xF2, 0xFF, 0xB4, 0xB9, 0xAE, 0
    xA3, 0x80, 0x8D, 0x9A, 0x97,
}

// Pre-computed multiplication by 14 in GF(256)
var gmul14 = [256]byte{
    0x00, 0x0E, 0x1C, 0x12, 0x38, 0x36, 0x24, 0x2A, 0x70, 0x7E, 0x6C, 0
    x62, 0x48, 0x46, 0x54, 0x5A,
    0xE0, 0xEE, 0xFC, 0xF2, 0xD8, 0xD6, 0xC4, 0xCA, 0x90, 0x9E, 0x8C, 0
    x82, 0xA8, 0xA6, 0xB4, 0xBA,
    0xDB, 0xD5, 0xC7, 0xC9, 0xE3, 0xED, 0xFF, 0xF1, 0xAB, 0xA5, 0xB7, 0
    xB9, 0x93, 0x9D, 0x8F, 0x81,
    0x3B, 0x35, 0x27, 0x29, 0x03, 0x0D, 0x1F, 0x11, 0x4B, 0x45, 0x57, 0
    x59, 0x73, 0x7D, 0x6F, 0x61,

```

```

    0xAD, 0xA3, 0xB1, 0xBF, 0x95, 0x9B, 0x89, 0x87, 0xDD, 0xD3, 0xC1, 0
    xCF, 0xE5, 0xEB, 0xF9, 0xF7,
    0x4D, 0x43, 0x51, 0x5F, 0x75, 0x7B, 0x69, 0x67, 0x3D, 0x33, 0x21, 0
    x2F, 0x05, 0x0B, 0x19, 0x17,
    0x76, 0x78, 0x6A, 0x64, 0x4E, 0x40, 0x52, 0x5C, 0x06, 0x08, 0x1A, 0
    x14, 0x3E, 0x30, 0x22, 0x2C,
    0x96, 0x98, 0x8A, 0x84, 0xAE, 0xA0, 0xB2, 0xBC, 0xE6, 0xE8, 0xFA, 0
    xF4, 0xDE, 0xD0, 0xC2, 0xCC,
    0x41, 0x4F, 0x5D, 0x53, 0x79, 0x77, 0x65, 0x6B, 0x31, 0x3F, 0x2D, 0
    x23, 0x09, 0x07, 0x15, 0x1B,
    0xA1, 0xAF, 0xBD, 0xB3, 0x99, 0x97, 0x85, 0x8B, 0xD1, 0xDF, 0xCD, 0
    xC3, 0xE9, 0xE7, 0xF5, 0xFB,
    0x9A, 0x94, 0x86, 0x88, 0xA2, 0xAC, 0xBE, 0xB0, 0xEA, 0xE4, 0xF6, 0
    xF8, 0xD2, 0xDC, 0xCE, 0xC0,
    0x7A, 0x74, 0x66, 0x68, 0x42, 0x4C, 0x5E, 0x50, 0x0A, 0x04, 0x16, 0
    x18, 0x32, 0x3C, 0x2E, 0x20,
    0xEC, 0xE2, 0xF0, 0xFE, 0xD4, 0xDA, 0xC8, 0xC6, 0x9C, 0x92, 0x80, 0
    x8E, 0xA4, 0xAA, 0xB8, 0xB6,
    0x0C, 0x02, 0x10, 0x1E, 0x34, 0x3A, 0x28, 0x26, 0x7C, 0x72, 0x60, 0
    x6E, 0x44, 0x4A, 0x58, 0x56,
    0x37, 0x39, 0x2B, 0x25, 0x0F, 0x01, 0x13, 0x1D, 0x47, 0x49, 0x5B, 0
    x55, 0x7F, 0x71, 0x63, 0x6D,
    0xD7, 0xD9, 0xCB, 0xC5, 0xEF, 0xE1, 0xF3, 0xFD, 0xA7, 0xA9, 0xBB, 0
    xB5, 0x9F, 0x91, 0x83, 0x8D,
}

// Initialized S-box
var sbbox = [256]byte{
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0
    x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0
    xAF, 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0
    xF1, 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0
    xE2, 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0
    xB3, 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0
    x39, 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0
    x7F, 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0
    x21, 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0
    x3D, 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0
    x14, 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0
    x62, 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0
    xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0
    x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0
    xB9, 0x86, 0xC1, 0x1D, 0x9E,

```

```

    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0
    xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0
    x0F, 0xB0, 0x54, 0xBB, 0x16,
}

// Initialized inverse S-box
var invSbox = [256]byte{
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0
    x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0
    x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0
    x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0
    x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0
    xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0
    x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0
    x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0
    x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0
    xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0
    xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0
    x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0
    xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0
    x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0
    x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0
    x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0
    x63, 0x55, 0x21, 0x0C, 0x7D,
}

// AddRoundKey XORs the round key xk with the state and puts the result
// in the state
func AddRoundKey(state, xk []uint32) {
    state[0] ^= xk[0]
    state[1] ^= xk[1]
    state[2] ^= xk[2]
    state[3] ^= xk[3]
}

// SubBytes replaces each byte of the state with its substitution in the
// S-box.
func SubBytes(state []uint32) {
    for i := 0; i < 4; i++ {
        state[i] = uint32(sbox[state[i]>>24])<<24 |

```

```

        uint32(sbox[state[i]>>16&0xff])<<16 |
        uint32(sbox[state[i]>>8&0xff])<<8 |
        uint32(sbox[state[i]&0xff])
    }
}

// InvSubBytes replaces each byte of the state with its substitution in
// the S-box.
func InvSubBytes(state []uint32) {
    for i := 0; i < 4; i++ {
        state[i] = uint32(invSbox[state[i]>>24])<<24 |
            uint32(invSbox[state[i]>>16&0xff])<<16 |
            uint32(invSbox[state[i]>>8&0xff])<<8 |
            uint32(invSbox[state[i]&0xff])
    }
}

// ShiftRows modifies the state to shift rows to the left
// First row is left unchanged
// Second row is shifted by one cell
// Third row is shifted by two cells
// Fourth row is shifted by three cells
func ShiftRows(state []uint32) {
    var s0, s1, s2, s3 uint32
    s0 = state[0]&(0xff<<24) | state[1]&(0xff<<16) | state[2]&(0xff<<8)
    | state[3]&0xff
    s1 = state[1]&(0xff<<24) | state[2]&(0xff<<16) | state[3]&(0xff<<8)
    | state[0]&0xff
    s2 = state[2]&(0xff<<24) | state[3]&(0xff<<16) | state[0]&(0xff<<8)
    | state[1]&0xff
    s3 = state[3]&(0xff<<24) | state[0]&(0xff<<16) | state[1]&(0xff<<8)
    | state[2]&0xff
    state[0], state[1], state[2], state[3] = s0, s1, s2, s3
}

// InvShiftRows modifies the state to shift rows to the left
// First row is left unchanged
// Second row is shifted by one cell
// Third row is shifted by two cells
// Fourth row is shifted by three cells
func InvShiftRows(state []uint32) {
    var s0, s1, s2, s3 uint32
    s0 = state[0]&(0xff<<24) | state[3]&(0xff<<16) | state[2]&(0xff<<8)
    | state[1]&0xff
    s1 = state[1]&(0xff<<24) | state[0]&(0xff<<16) | state[3]&(0xff<<8)
    | state[2]&0xff
    s2 = state[2]&(0xff<<24) | state[1]&(0xff<<16) | state[0]&(0xff<<8)
    | state[3]&0xff
    s3 = state[3]&(0xff<<24) | state[2]&(0xff<<16) | state[1]&(0xff<<8)
    | state[0]&0xff
    state[0], state[1], state[2], state[3] = s0, s1, s2, s3
}

// MixColumns updates the state with the MixColumns operation of the AES
// . It uses
// two pre-computed tables that implement multiplication by 2 and by 3

```

```

    in GF(256).
func MixColumns(state []uint32) {
    var b0, b1, b2, b3, d0, d1, d2, d3 byte

    for i := 0; i < 4; i++ {
        b0 = byte(state[i] >> 24)
        b1 = byte(state[i] >> 16)
        b2 = byte(state[i] >> 8)
        b3 = byte(state[i])

        d0 = gmul2[b0] ^ gmul3[b1] ^ b2 ^ b3
        d1 = gmul2[b1] ^ gmul3[b2] ^ b3 ^ b0
        d2 = gmul2[b2] ^ gmul3[b3] ^ b0 ^ b1
        d3 = gmul2[b3] ^ gmul3[b0] ^ b1 ^ b2

        state[i] = uint32(d0)<<24 | uint32(d1)<<16 | uint32(d2)<<8 |
uint32(d3)
    }
}

func aesd(v0 []uint32, v1 []uint32) {
    AddRoundKey(v0, v1)
    InvShiftRows(v0)
    InvSubBytes(v0)
}

func aese(v0 []uint32, v1 []uint32) {
    AddRoundKey(v0, v1)
    ShiftRows(v0)
    SubBytes(v0)
}

func xor(v0 []uint32, v1 []uint32) {
    AddRoundKey(v0, v1)
}

```

```

// decrypt_state.go
package main

import (
    "bufio"
    "fmt"
    "os"
    "regexp"
)

const OUT = "test_debugged.decoded"

// Same key, in reverse order
var key = []uint32{0x11111111, 0x22222222, 0x11111111, 0x22222222}
var invKey = []uint32{0x22222222, 0x11111111, 0x22222222, 0x11111111}

```

```

func decode(a1, a2 string) string {
    var b1, b2 uint64
    fmt.Sscanf(a1, "0x%x", &b1)
    fmt.Sscanf(a2, "0x%x", &b2)
    dec := []uint32{uint32(b1 >> 32), uint32(b1 & 0xffffffff), uint32(b2
>> 32), uint32(b2 & 0xffffffff)}
    if dec[0] != 0 && dec[1] != 0 && dec[2] != 0 && dec[3] != 0 {
        aese(dec, key)
        xor(dec, invKey)
    }
    return fmt.Sprintf("0x%02x%02x%02x%02x", dec[1]&0xff, dec[2]>>24, (
dec[1]>>16)&0xff, (dec[1]>>8)&0xff)
}

func regFromAddr(address string) string {
    switch address {
    case "0xe04a5d8":
        return "r0"
        break
    case "0xe04a5e8":
        return "r1 "
        break
    case "0xe04a5f8":
        return "r2 "
        break
    case "0xe04a608":
        return "r3 "
        break
    case "0xe04a618":
        return "r4 "
        break
    case "0xe04a628":
        return "r5 "
        break
    case "0xe04a638":
        return "r6 "
        break
    case "0xe04a648":
        return "r7 "
        break
    case "0xe04a658":
        return "r8 "
        break
    case "0xe04a668":
        return "r9 "
        break
    case "0xe04a678":
        return "r10"
        break
    case "0xe04a688":
        return "r11"
        break
    case "0xe04a698":
        return "r12"
        break
    case "0xe04a6a8":

```

```

        return "r13"
        break
    case "0xe04a6b8":
        return "r14"
        break
    case "0xe04a6c8":
        return "pc "
        break
    }
    return "unknown"
}

func main() {
    reg, _ := regexp.Compile(`(0x.*):\s*(0x.....)\s*(0x
    .....)` )
    fp, _ := os.Open("./test_debugged")
    os.Remove(OUT)
    output, _ := os.Create(OUT)
    scanner := bufio.NewScanner(fp)

    for scanner.Scan() {
        line := scanner.Text()
        matches := reg.FindStringSubmatch(line)
        if matches == nil {
            fmt.Fprintf(output, "%s\n", line)
        } else {
            decoded := decode(matches[2], matches[3])
            reg := regFromAddr(matches[1])
            fmt.Fprintf(output, "%s: %s\n", reg, decoded)
        }
    }
}

```

```

-- prog.zip
UESDBBQAAAAIA0k0r06kwlj2NgEAAABAAADABwAcHJvVVQJAAP219tcA5jbXHV4CwABBAAAAAAAE
AAAAAHWQsUtCURSHv/u6klTmKWl/SEOD9Roc2rptIqZuNVg4NjY2ZNY1cHBoaDCrIWqI8E+4
QnsS
DS1Bf4BztL2uLxAc0hw0/
A7f4cAnZITdkHUxWTG10ESaLKKFitCAfsA6dS2mIqaBSWJHi62IbWCT
ONTiKuIauH5os7MuA5/YHFpRV2k0YlqxafEepF2VbiC2yl0eju8NHl0xbV246zl/5
ZYYTjNXCXOU
8ufPE+Zjmr1MmLWU//I2Yf7/9T1h9o00qVLPsxXMcKg4w75GqsdCwA/
KFXDN8mDES5NVKMEtZdWG
YBxPUKaAaVJqlndG+PUXe/bG6/
KKvCtOiJnXJvKKvBbv3ZIKdkUFerWuBGyQi10i6HZhgxFHXPk
sY7NKKXrzTnUR9aDaisx4mkgR4ZbPBz0EzD1/9
QtQSwECHgMUAACADpNK9OpMJY9jYBAADAAQAA
AwAYAAAAAAAAAAAAAA/4
EAAAAAcHJvVVQFAAP219tcdXgLAEEEEAAAAAAQAAAAAUESFBgAAAAABAAEA
SQAAAHMBAAAAAA==

```

```

// disass_prog.go
package main

import (
    "fmt"
    "io/ioutil"
)

type instruction struct {
    opcode uint8
    flags  uint8
    arg1   uint8
    arg2   uint8
    arg3   uint16
}

func (ins *instruction) String() string {
    fmt.Printf("opcode: 0x%x, flags: 0b%02b, arg1: 0x%x, arg2: 0x%x,
arg3: 0x%03x\n", ins.opcode, ins.flags, ins.arg1, ins.arg2, ins.arg3)
    switch ins.opcode {
    case 0:
        if ins.flags == 3 {
            return fmt.Sprintf("mov r%d, 0x%02x", ins.arg1, ins.arg3)
        } else if ins.flags == 1 {
            return fmt.Sprintf("mov r%d, [r%d]", ins.arg1, ins.arg2)
        } else if ins.flags == 0 {
            return fmt.Sprintf("mov r%d, r%d", ins.arg1, ins.arg2)
        }
        return fmt.Sprintf("mov [r%d], r%d", ins.arg1, ins.arg2)
    case 1:
        if ins.flags == 0 {
            return fmt.Sprintf("dec r%d", ins.arg1)
        }
        break
    case 2:
        if ins.flags == 3 {
            arg3 := ins.arg3 | uint16(ins.arg2)<<10
            return fmt.Sprintf("add r%d, 0x%02x", ins.arg1, arg3)
        } else if ins.flags == 0 {
            return fmt.Sprintf("add r%d, r%d", ins.arg1, ins.arg2)
        }
        break
    case 3:
        if ins.flags == 3 {
            arg3 := ins.arg3 | uint16(ins.arg2)<<10
            return fmt.Sprintf("sub r%d, 0x%02x", ins.arg1, arg3)
        } else if ins.flags == 0 {
            return fmt.Sprintf("sub r%d, r%d", ins.arg1, ins.arg2)
        }
        break
    case 4:
        if ins.flags == 3 {
            return fmt.Sprintf("lsl r%d, 0x%02x", ins.arg1, ins.arg3)
        }
    }
}

```

```

        break
    case 5:
        if ins.flags == 3 {
            return fmt.Sprintf("lsr r%d, 0x%02x", ins.arg1, ins.arg3)
        }
        break
    case 6:
        if ins.flags == 0 {
            return fmt.Sprintf("xor r%d, r%d", ins.arg1, ins.arg2)
        }
        break
    case 7:
        if ins.flags == 3 {
            return fmt.Sprintf("and r%d, 0x%02x", ins.arg1, ins.arg3)
        }
        break
    case 8:
        if ins.flags == 3 {
            return fmt.Sprintf("jmp 0x%03x", ins.arg3)
        }
        break
    case 9:
        if ins.flags == 3 {
            return fmt.Sprintf("jmp 0x%03x if r%d != 0", ins.arg3, ins.
arg1)
        }
        break
    case 0xa:
        return "end"
    case 0xb:
        if ins.flags == 0 {
            return fmt.Sprintf("swap r%d[0], r%d[1]", ins.arg1, ins.arg1
)
        }
        break
    case 0xc:
        if ins.flags == 3 {
            return fmt.Sprintf("store key fragment")
        }
        break
    case 0xd:
        return "unknown"
    case 0xe:
        if ins.flags == 3 {
            return fmt.Sprintf("jmp 0x%03x", ins.arg3)
        }
        break
    }

    fmt.Printf("; 0x%06x\n", ins.encode())
    return fmt.Sprintf("opcode: 0x%x, flags: 0b%02b, arg1: 0x%x, arg2: 0
x%x, arg3: 0x%03x", ins.opcode, ins.flags, ins.arg1, ins.arg2, ins.
arg3)
}

func decode(encoded uint) *instruction {

```

```

    op := uint8(encoded >> 0x14 & 0xf)
    flags := uint8(encoded >> 0x12 & 3)
    arg1 := uint8(encoded >> 0xe & 0xf)
    arg2 := uint8(encoded >> 0xa & 0xf)
    arg3 := uint16(encoded & 0x3ff)
    return &instruction{op, flags, arg1, arg2, arg3}
}

func (instr *instruction) encode() uint {
    return uint(instr.arg3) | uint(instr.arg2)<<0xa | uint(instr.arg1)
    <<0xe | uint(instr.flags)<<0x12 | uint(instr.opcode)<<0x14
}

func main() {
    var instr uint
    var i int
    content, _ := ioutil.ReadFile("prog")
    for i < len(content)-2 {
        instr = uint(content[i+2])<<16 | uint(content[i+1])<<8 | uint(
        content[i])
        fmt.Printf("%03x: %s\n", i, decode(instr))
        i += 3
    }
}

```

```

// sm4.go

/*
Copyright Suzhou Tongji Fintech Research Institute 2017 All Rights
Reserved.
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

sm4 acceration
modified by Jack, 2017 Oct
*/

package main

import (
    "strconv"
)

const BlockSize = 16

```

```

type SM4Key []byte

type KeySizeError int

// Cipher is an instance of SM4 encryption.
type Sm4Cipher struct {
    subkeys []uint32
    block1  []uint32
    block2  []byte
}

// defined in the spec
var fk = [4]uint32{
    0xa3b1bac6, 0x56aa3350, 0x677d9197, 0xb27022dc,
}

// defined in the spec
var ck = [32]uint32{
    0x00070e15, 0x1c232a31, 0x383f464d, 0x545b6269,
    0x70777e85, 0x8c939aa1, 0xa8afb6bd, 0xc4cbd2d9,
    0xe0e7eef5, 0xfc030a11, 0x181f262d, 0x343b4249,
    0x50575e65, 0x6c737a81, 0x888f969d, 0xa4abb2b9,
    0xc0c7ced5, 0xdce3eaf1, 0xf8ff060d, 0x141b2229,
    0x30373e45, 0x4c535a61, 0x686f767d, 0x848b9299,
    0xa0a7aeb5, 0xbcc3cad1, 0xd8dfe6ed, 0xf4fb0209,
    0x10171e25, 0x2c333a41, 0x484f565d, 0x646b7279,
}

// sm4 sboxes
var sbox = [256]uint8{
    0xd6, 0x90, 0xe9, 0xfe, 0xcc, 0xe1, 0x3d, 0xb7, 0x16, 0xb6, 0x14, 0
xc2, 0x28, 0xfb, 0x2c, 0x05,
    0x2b, 0x67, 0x9a, 0x76, 0x2a, 0xbe, 0x04, 0xc3, 0xaa, 0x44, 0x13, 0
x26, 0x49, 0x86, 0x06, 0x99,
    0x9c, 0x42, 0x50, 0xf4, 0x91, 0xef, 0x98, 0x7a, 0x33, 0x54, 0x0b, 0
x43, 0xed, 0xcf, 0xac, 0x62,
    0xe4, 0xb3, 0x1c, 0xa9, 0xc9, 0x08, 0xe8, 0x95, 0x80, 0xdf, 0x94, 0
xfa, 0x75, 0x8f, 0x3f, 0xa6,
    0x47, 0x07, 0xa7, 0xfc, 0xf3, 0x73, 0x17, 0xba, 0x83, 0x59, 0x3c, 0
x19, 0xe6, 0x85, 0x4f, 0xa8,
    0x68, 0x6b, 0x81, 0xb2, 0x71, 0x64, 0xda, 0x8b, 0xf8, 0xeb, 0x0f, 0
x4b, 0x70, 0x56, 0x9d, 0x35,
    0x1e, 0x24, 0x0e, 0x5e, 0x63, 0x58, 0xd1, 0xa2, 0x25, 0x22, 0x7c, 0
x3b, 0x01, 0x21, 0x78, 0x87,
    0xd4, 0x00, 0x46, 0x57, 0x9f, 0xd3, 0x27, 0x52, 0x4c, 0x36, 0x02, 0
xe7, 0xa0, 0xc4, 0xc8, 0x9e,
    0xea, 0xbf, 0x8a, 0xd2, 0x40, 0xc7, 0x38, 0xb5, 0xa3, 0xf7, 0xf2, 0
xce, 0xf9, 0x61, 0x15, 0xa1,
    0xe0, 0xae, 0x5d, 0xa4, 0x9b, 0x34, 0x1a, 0x55, 0xad, 0x93, 0x32, 0
x30, 0xf5, 0x8c, 0xb1, 0xe3,
    0x1d, 0xf6, 0xe2, 0x2e, 0x82, 0x66, 0xca, 0x60, 0xc0, 0x29, 0x23, 0
xab, 0x0d, 0x53, 0x4e, 0x6f,
    0xd5, 0xdb, 0x37, 0x45, 0xde, 0xfd, 0x8e, 0x2f, 0x03, 0xff, 0x6a, 0
x72, 0x6d, 0x6c, 0x5b, 0x51,
    0x8d, 0x1b, 0xaf, 0x92, 0xbb, 0xdd, 0xbc, 0x7f, 0x11, 0xd9, 0x5c, 0

```

```

x41, 0x1f, 0x10, 0x5a, 0xd8,
  0x0a, 0xc1, 0x31, 0x88, 0xa5, 0xcd, 0x7b, 0xbd, 0x2d, 0x74, 0xd0, 0
x12, 0xb8, 0xe5, 0xb4, 0xb0,
  0x89, 0x69, 0x97, 0x4a, 0x0c, 0x96, 0x77, 0x7e, 0x65, 0xb9, 0xf1, 0
x09, 0xc5, 0x6e, 0xc6, 0x84,
  0x18, 0xf0, 0x7d, 0xec, 0x3a, 0xdc, 0x4d, 0x20, 0x79, 0xee, 0x5f, 0
x3e, 0xd7, 0xcb, 0x39, 0x48,
}

var sbbox0 = [256]uint32{
  0xd55b5b8e, 0x924242d0, 0xeaa7a74d, 0xfdfbfb06, 0xcf3333fc, 0
xe2878765, 0x3df4f4c9, 0xb5dede6b, 0x1658584e, 0xb4dada6e, 0x14505044
, 0xc10b0bca, 0x28a0a088, 0xf8efef17, 0x2cb0b09c, 0x05141411,
  0x2bacac87, 0x669d9dfb, 0x986a6af2, 0x77d9d9ae, 0x2aa8a882, 0
xbcfafa46, 0x04101014, 0xc00f0fcf, 0xa8aaaa02, 0x45111154, 0x134c4c5f
, 0x269898be, 0x4825256d, 0x841a1a9e, 0x0618181e, 0x9b6666fd,
  0x9e7272ec, 0x4309094a, 0x51414110, 0xf7d3d324, 0x934646d5, 0
xecfbfbf53, 0x9a6262f8, 0x7be9e992, 0x33ccccff, 0x55515104, 0x0b2c2c27
, 0x420d0d4f, 0xeeb7b759, 0xcc3f3ff3, 0xae2b2b21c, 0x638989ea,
  0xe7939374, 0xb1cece7f, 0x1c70706c, 0xaba6a60d, 0xca2727ed, 0
x08202028, 0xeba3a348, 0x975656c1, 0x82020280, 0xdc7f7fa3, 0x965252c4
, 0xf9ebeb12, 0x74d5d5a1, 0x8d3e3eb3, 0x3ffcfcc3, 0xa49a9a3e,
  0x461d1d5b, 0x071c1c1b, 0xa59e9e3b, 0xffff3f30c, 0xf0cfcf3f, 0
x72cdcdbf, 0x175c5c4b, 0xb8eaea52, 0x810e0e8f, 0x5865653d, 0x3cf0f0cc
, 0x1964647d, 0xe59b9b7e, 0x87161691, 0x4e3d3d73, 0xaa2a2a08,
  0x69a1a1c8, 0x6aadadc7, 0x83060685, 0xb0caca7a, 0x70c5c5b5, 0
x659191f4, 0xd96b6bb2, 0x892e2ea7, 0xfbe3e318, 0xe8afaf47, 0x0f3c3c33
, 0x4a2d2d67, 0x71c1c1b0, 0x5759590e, 0x9f7676e9, 0x35d4d4e1,
  0x1e787866, 0x249090b4, 0x0e383836, 0x5f797926, 0x628d8def, 0
x59616138, 0xd2474795, 0xa08a8a2a, 0x259494b1, 0x228888aa, 0x7df1f18c
, 0x3bececd7, 0x01040405, 0x218484a5, 0x79e1e198, 0x851e1e9b,
  0xd7535384, 0x00000000, 0x4719195e, 0x565d5d0b, 0x9d7e7ee3, 0
xd04f4f9f, 0x279c9cbb, 0x5349491a, 0x4d31317c, 0x36d8d8ee, 0x0208080a
, 0xe49f9f7b, 0xa2828220, 0xc71313d4, 0xcb2323e8, 0x9c7a7ae6,
  0xe9abab42, 0xbdfefe43, 0x882a2aa2, 0xd14b4b9a, 0x41010140, 0
xc41f1fdb, 0x38e0e0d8, 0xb7d6d661, 0xa18e8e2f, 0xf4dfdf2b, 0xf1cbcb3a
, 0xcd3b3bf6, 0xfae7e71d, 0x608585e5, 0x15545441, 0xa3868625,
  0xe3838360, 0xacbaba16, 0x5c757529, 0xa6929234, 0x996e6ef7, 0
x34d0d0e4, 0x1a686872, 0x54555501, 0xafb6b619, 0x914e4edf, 0x32c8c8fa
, 0x30c0c0f0, 0xf6d7d721, 0x8e3232bc, 0xb3c6c675, 0xe08f8f6f,
  0x1d747469, 0xf5bdbb2e, 0xe18b8b6a, 0x2eb8b896, 0x800a0a8a, 0
x679999fe, 0xc92b2be2, 0x618181e0, 0xc30303c0, 0x29a4a48d, 0x238c8caf
, 0xa9aeae07, 0x0d343439, 0x524d4d1f, 0x4f393976, 0x6ebdbdd3,
  0xd6575781, 0xd86f6fb7, 0x37dcdceb, 0x44151551, 0xdd7b7ba6, 0
xfef7f709, 0x8c3a3ab6, 0x2fbc9c93, 0x030c0c0f, 0xfcffff03, 0x6ba9a9c2
, 0x73c9c9ba, 0x6cb5b5d9, 0x6db1b1dc, 0x5a6d6d37, 0x50454515,
  0x8f3636b9, 0x1b6c6c77, 0xadbebe13, 0x904a4ada, 0xb9eeee57, 0
xde7777a9, 0xbef2f24c, 0x7efdfd83, 0x11444455, 0xda6767bd, 0x5d71712c
, 0x40050545, 0x1f7c7c63, 0x10404050, 0x5b696932, 0xdb6363b8,
  0x0a282822, 0xc20707c5, 0x31c4c4f5, 0x8a2222a8, 0xa7969631, 0
xce3737f9, 0x7aeded97, 0xbff6f649, 0x2db4b499, 0x75d1d1a4, 0xd3434390
, 0x1248485a, 0xbae2e258, 0xe6979771, 0xb6d2d264, 0xb2c2c270,
  0x8b2626ad, 0x68a5a5cd, 0x955e5ecb, 0x4b292962, 0x0c30303c, 0
x945a5ace, 0x76ddddab, 0x7ff9f986, 0x649595f1, 0xbbe6e65d, 0xf2c7c735
, 0x0924242d, 0xc61717d1, 0x6fb9b9d6, 0xc51b1bde, 0x86121294,
  0x18606078, 0xf3c3c330, 0x7cf5f589, 0xefb3b35c, 0x3ae8e8d2, 0

```

```

    xdf7373ac, 0x4c353579, 0x208080a0, 0x78e5e59d, 0xedbbbb56, 0x5e7d7d23
    , 0x3ef8f8c6, 0xd45f5f8b, 0xc82f2fe7, 0x39e4e4dd, 0x49212168,
}

var sbbox1 = [256]uint32{
    0x5b5b8ed5, 0x4242d092, 0xa7a74dea, 0xfbfb06fd, 0x3333fccf, 0
    x878765e2, 0xf4f4c93d, 0xdede6bb5, 0x58584e16, 0xdada6eb4, 0x50504414
    , 0x0b0bcac1, 0xa0a08828, 0xefef17f8, 0xb0b09c2c, 0x14141105,
    0xacac872b, 0x9d9dfb66, 0x6a6af298, 0xd9d9ae77, 0xa8a8822a, 0
    xfafa46bc, 0x10101404, 0x0f0fcfc0, 0xaaaa02a8, 0x11115445, 0x4c4c5f13
    , 0x9898be26, 0x25256d48, 0x1a1a9e84, 0x18181e06, 0x6666fd9b,
    0x7272ec9e, 0x09094a43, 0x414111051, 0xd3d324f7, 0x4646d593, 0
    xbfbf53ec, 0x6262f89a, 0xe9e9927b, 0xccccff33, 0x51510455, 0x2c2c270b
    , 0x0d0d4f42, 0xb7b759ee, 0x3f3ff3cc, 0xb2b21cae, 0x8989ea63,
    0x939374e7, 0xcece7fb1, 0x70706c1c, 0xa6a60dab, 0x2727edca, 0
    x20202808, 0xa3a348eb, 0x5656c197, 0x02028082, 0x7f7fa3dc, 0x5252c496
    , 0xebeb12f9, 0xd5d5a174, 0x3e3eb38d, 0xfcfc33f, 0x9a9a3ea4,
    0x1d1d5b46, 0x1c1c1b07, 0x9e9e3ba5, 0xf3f30cff, 0xcfcf3ff0, 0
    xcdbbf72, 0x5c5c4b17, 0xeaea52b8, 0x0e0e8f81, 0x65653d58, 0xf0f0cc3c
    , 0x64647d19, 0x9b9b7ee5, 0x16169187, 0x3d3d734e, 0xa2a208aa,
    0xa1a1c869, 0xadadc76a, 0x06068583, 0xcaca7ab0, 0xc5c5b570, 0
    x9191f465, 0x6b6bb2d9, 0x2e2ea789, 0xe3e318fb, 0xafaf47e8, 0x3c3c330f
    , 0x2d2d674a, 0xc1c1b071, 0x59590e57, 0x7676e99f, 0xd4d4e135,
    0x7878661e, 0x9090b424, 0x3838360e, 0x7979265f, 0x8d8def62, 0
    x61613859, 0x474795d2, 0x8a8a2aa0, 0x9494b125, 0x8888aa22, 0xf1f18c7d
    , 0xececd73b, 0x04040501, 0x8484a521, 0xe1e19879, 0x1e1e9b85,
    0x535384d7, 0x00000000, 0x19195e47, 0x5d5d0b56, 0x7e7ee39d, 0
    x4f4f9fd0, 0x9c9cbb27, 0x49491a53, 0x31317c4d, 0xd8d8ee36, 0x08080a02
    , 0x9f9f7be4, 0x828220a2, 0x1313d4c7, 0x2323e8cb, 0x7a7ae69c,
    0xabab42e9, 0xfefe43bd, 0x2a2aa288, 0x4b4b9ad1, 0x01014041, 0
    x1f1fdbcb4, 0xe0e0d838, 0xd6d661b7, 0x8e8e2fa1, 0xdfdf2bf4, 0xcbcb3af1
    , 0x3b3bf6cd, 0xe7e71dfa, 0x8585e560, 0x54544115, 0x868625a3,
    0x838360e3, 0xbaba16ac, 0x7575295c, 0x929234a6, 0x6e6ef799, 0
    xd0d0e434, 0x6868721a, 0x55550154, 0xb6b619af, 0x4e4edf91, 0xc8c8fa32
    , 0xc0c0f030, 0xd7d721f6, 0x3232bc8e, 0xc6c675b3, 0x8f8f6fe0,
    0x7474691d, 0xdbdb2ef5, 0x8b8b6ae1, 0xb8b8962e, 0x0a0a8a80, 0
    x9999fe67, 0x2b2be2c9, 0x8181e061, 0x0303c0c3, 0xa4a48d29, 0x8c8caf23
    , 0xaeae07a9, 0x3434390d, 0x4d4d1f52, 0x3939764f, 0xbdbdd36e,
    0x575781d6, 0x6f6fb7d8, 0xdcdbceb37, 0x15155144, 0x7b7ba6dd, 0
    xf7f709fe, 0x3a3ab68c, 0xbcbcb932f, 0xc0c0f03, 0xffff03fc, 0xa9a9c26b
    , 0xc9c9ba73, 0xb5b5d96c, 0xb1b1dc6d, 0x6d6d375a, 0x45451550,
    0x3636b98f, 0x6c6c771b, 0xbebe13ad, 0x4a4ada90, 0xeeee57b9, 0
    x7777a9de, 0xf2f24cbe, 0xdfdf837e, 0x44445511, 0x6767bdda, 0x71712c5d
    , 0x05054540, 0x7c7c631f, 0x40405010, 0x6969325b, 0x6363b8db,
    0x2828220a, 0x0707c5c2, 0xc4c4f531, 0x2222a88a, 0x969631a7, 0
    x3737f9ce, 0xeded977a, 0xf6f649bf, 0xb4b4992d, 0xd1d1a475, 0x434390d3
    , 0x48485a12, 0xe2e258ba, 0x979771e6, 0xd2d264b6, 0xc2c270b2,
    0x2626ad8b, 0xa5a5cd68, 0x5e5ecb95, 0x2929624b, 0x30303c0c, 0
    x5a5ace94, 0xddddab76, 0xf9f9867f, 0x9595f164, 0xe6e65dbb, 0xc7c735f2
    , 0x24242d09, 0x1717d1c6, 0xb9b9d66f, 0x1b1bdec5, 0x12129486,
    0x60607818, 0xc3c330f3, 0xf5f5897c, 0xb3b35cef, 0xe8e8d23a, 0
    x7373acdf, 0x3535794c, 0x8080a020, 0xe5e59d78, 0xbbbb56ed, 0x7d7d235e
    , 0xf8f8c63e, 0x5f5f8bd4, 0x2f2fe7c8, 0xe4e4dd39, 0x21216849,
}

var sbbox2 = [256]uint32{

```

```

    0x5b8ed55b, 0x42d09242, 0xa74deaa7, 0xfb06fdfb, 0x33fccf33, 0
    x8765e287, 0xf4c93df4, 0xde6bb5de, 0x584e1658, 0xda6eb4da, 0x50441450
    , 0x0bcac10b, 0xa08828a0, 0xef17f8ef, 0xb09c2cb0, 0x14110514,
    0xac872bac, 0x9dfb669d, 0x6af2986a, 0xd9ae77d9, 0xa8822aa8, 0
    xfa46bcfa, 0x10140410, 0x0fcfc00f, 0xaa02a8aa, 0x11544511, 0x4c5f134c
    , 0x98be2698, 0x256d4825, 0x1a9e841a, 0x181e0618, 0x66fd9b66,
    0x72ec9e72, 0x094a4309, 0x41105141, 0xd324f7d3, 0x46d59346, 0
    xbf53ecbf, 0x62f89a62, 0xe9927be9, 0xccff33cc, 0x51045551, 0x2c270b2c
    , 0x0d4f420d, 0xb759eeb7, 0x3ff3cc3f, 0xb21caeb2, 0x89ea6389,
    0x9374e793, 0xce7fb1ce, 0x706c1c70, 0xa60daba6, 0x27edca27, 0
    x20280820, 0xa348eba3, 0x56c19756, 0x02808202, 0x7fa3dc7f, 0x52c49652
    , 0xeb12f9eb, 0xd5a174d5, 0x3eb38d3e, 0xfcc33ffc, 0x9a3ea49a,
    0x1d5b461d, 0x1c1b071c, 0x9e3ba59e, 0xf30cfff3, 0xcf3ff0cf, 0
    xcdbf72cd, 0x5c4b175c, 0xea52b8ea, 0x0e8f810e, 0x653d5865, 0xf0cc3cf0
    , 0x647d1964, 0x9b7ee59b, 0x16918716, 0x3d734e3d, 0xa208aaa2,
    0xa1c869a1, 0xadc76aad, 0x06858306, 0xca7ab0ca, 0xc5b570c5, 0
    x91f46591, 0x6bb2d96b, 0x2ea7892e, 0xe318fbe3, 0xaf47e8af, 0x3c330f3c
    , 0x2d674a2d, 0xc1b071c1, 0x590e5759, 0x76e99f76, 0xd4e135d4,
    0x78661e78, 0x90b42490, 0x38360e38, 0x79265f79, 0x8def628d, 0
    x61385961, 0x4795d247, 0x8a2aa08a, 0x94b12594, 0x88aa2288, 0xf18c7df1
    , 0xecd73bec, 0x04050104, 0x84a52184, 0xe19879e1, 0x1e9b851e,
    0x5384d753, 0x00000000, 0x195e4719, 0x5d0b565d, 0x7ee39d7e, 0
    x4f9fd04f, 0x9cbb279c, 0x491a5349, 0x317c4d31, 0xd8ee36d8, 0x080a0208
    , 0x9f7be49f, 0x8220a282, 0x13d4c713, 0x23e8cb23, 0x7ae69c7a,
    0xab42e9ab, 0xfe43bdfe, 0x2aa2882a, 0x4b9ad14b, 0x01404101, 0
    x1fdbbc41f, 0xe0d838e0, 0xd661b7d6, 0x8e2fa18e, 0xdf2bf4df, 0xcb3af1cb
    , 0x3bf6cd3b, 0xe71dfae7, 0x85e56085, 0x54411554, 0x8625a386,
    0x8360e383, 0xba16acba, 0x75295c75, 0x9234a692, 0x6ef7996e, 0
    xd0e434d0, 0x68721a68, 0x55015455, 0xb619afb6, 0x4edf914e, 0xc8fa32c8
    , 0xc0f030c0, 0xd721f6d7, 0x32bc8e32, 0xc675b3c6, 0x8f6fe08f,
    0x74691d74, 0xdb2ef5db, 0x8b6ae18b, 0xb8962eb8, 0x0a8a800a, 0
    x99fe6799, 0x2be2c92b, 0x81e06181, 0x03c0c303, 0xa48d29a4, 0x8caf238c
    , 0xae07a9ae, 0x34390d34, 0x4d1f524d, 0x39764f39, 0xbdd36ebd,
    0x5781d657, 0x6fb7d86f, 0xdceb37dc, 0x15514415, 0x7ba6dd7b, 0
    xf709fef7, 0x3ab68c3a, 0xbc932fbc, 0x0c0f030c, 0xff03fcff, 0xa9c26ba9
    , 0xc9ba73c9, 0xb5d96cb5, 0xb1dc6db1, 0x6d375a6d, 0x45155045,
    0x36b98f36, 0x6c771b6c, 0xbe13adbe, 0x4ada904a, 0xee57b9ee, 0
    x77a9de77, 0xf24cbef2, 0xfd837efd, 0x44551144, 0x67bdda67, 0x712c5d71
    , 0x05454005, 0x7c631f7c, 0x40501040, 0x69325b69, 0x63b8db63,
    0x28220a28, 0x07c5c207, 0xc4f531c4, 0x22a88a22, 0x9631a796, 0
    x37f9ce37, 0xed977aed, 0xf649bff6, 0xb4992db4, 0xd1a475d1, 0x4390d343
    , 0x485a1248, 0xe258bae2, 0x9771e697, 0xd264b6d2, 0xc270b2c2,
    0x26ad8b26, 0xa5cd68a5, 0x5ecb955e, 0x29624b29, 0x303c0c30, 0
    x5ace945a, 0xddab76dd, 0xf9867ff9, 0x95f16495, 0xe65dbbe6, 0xc735f2c7
    , 0x242d0924, 0x17d1c617, 0xb9d66fb9, 0x1bdec51b, 0x12948612,
    0x60781860, 0xc330f3c3, 0xf5897cf5, 0xb35cefb3, 0xe8d23ae8, 0
    x73acdf73, 0x35794c35, 0x80a02080, 0xe59d78e5, 0xbb56edbb, 0x7d235e7d
    , 0xf8c63ef8, 0x5f8bd45f, 0x2fe7c82f, 0xe4dd39e4, 0x21684921,
}

var sbbox3 = [256]uint32{
    0x8ed55b5b, 0xd0924242, 0x4deaa7a7, 0x06fdfbfb, 0xfccf3333, 0
    x65e28787, 0xc93df4f4, 0x6bb5dede, 0x4e165858, 0x6eb4dada, 0x44145050
    , 0xcac10b0b, 0x8828a0a0, 0x17f8efef, 0x9c2cb0b0, 0x11051414,
    0x872bacac, 0xfb669d9d, 0xf2986a6a, 0xae77d9d9, 0x822aa8a8, 0
    x46bcfafafa, 0x14041010, 0xcfc00f0f, 0x02a8aaaa, 0x54451111, 0x5f134c4c

```

```

, 0xbe269898, 0x6d482525, 0x9e841a1a, 0x1e061818, 0xfd9b6666,
0xec9e7272, 0x4a430909, 0x10514141, 0x24f7d3d3, 0xd5934646, 0
x53ecbfbf, 0xf89a6262, 0x927be9e9, 0xff33cccc, 0x04555151, 0x270b2c2c
, 0x4f420d0d, 0x59eeb7b7, 0xf3cc3f3f, 0x1caeb2b2, 0xea638989,
0x74e79393, 0x7fb1cece, 0x6c1c7070, 0x0daba6a6, 0xedca2727, 0
x28082020, 0x48eba3a3, 0xc1975656, 0x80820202, 0xa3dc7f7f, 0xc4965252
, 0x12f9ebef, 0xa174d5d5, 0xb38d3e3e, 0xc33ffcf, 0x3ea49a9a,
0x5b461d1d, 0x1b071c1c, 0x3ba59e9e, 0x0cfff3f3, 0x3ff0cfcf, 0
xbf72cdcd, 0x4b175c5c, 0x52b8eaea, 0x8f810e0e, 0x3d586565, 0xcc3cf0f0
, 0x7d196464, 0x7ee59b9b, 0x91871616, 0x734e3d3d, 0x08aaa2a2,
0xc869a1a1, 0xc76aadad, 0x85830606, 0x7ab0caca, 0xb570c5c5, 0
xf4659191, 0xb2d96b6b, 0xa7892e2e, 0x18fbe3e3, 0x47e8afaf, 0x330f3c3c
, 0x674a2d2d, 0xb071c1c1, 0x0e575959, 0xe99f7676, 0xe135d4d4,
0x661e7878, 0xb4249090, 0x360e3838, 0x265f7979, 0xef628d8d, 0
x38596161, 0x95d24747, 0x2aa08a8a, 0xb1259494, 0xaa228888, 0x8c7df1f1
, 0xd73bec, 0x05010404, 0xa5218484, 0x9879e1e1, 0x9b851e1e,
0x84d75353, 0x00000000, 0x5e471919, 0x0b565d5d, 0xe39d7e7e, 0
x9fd04f4f, 0xbb279c9c, 0x1a534949, 0x7c4d3131, 0xee36d8d8, 0xa020808
, 0x7be49f9f, 0x20a28282, 0xd4c71313, 0xe8cb2323, 0xe69c7a7a,
0x42e9abab, 0x43bdfefe, 0xa2882a2a, 0x9ad14b4b, 0x40410101, 0
xdbc41f1f, 0xd838e0e0, 0x61b7d6d6, 0x2fa18e8e, 0x2bf4dfdf, 0x3af1cbcb
, 0xf6cd3b3b, 0x1dfae7e7, 0xe5608585, 0x41155454, 0x25a38686,
0x60e38383, 0x16acbaba, 0x295c7575, 0x34a69292, 0xf7996e6e, 0
xe434d0d0, 0x721a6868, 0x01545555, 0x19afb6b6, 0xdf914e4e, 0xfa32c8c8
, 0xf030c0c0, 0x21f6d7d7, 0xbc8e3232, 0x75b3c6c6, 0x6fe08f8f,
0x691d7474, 0x2ef5dbdb, 0xae18b8b, 0x962eb8b8, 0x8a800a0a, 0
xfe679999, 0xe2c92b2b, 0xe0618181, 0xc0c30303, 0x8d29a4a4, 0xaf238c8c
, 0x07a9aeae, 0x390d3434, 0x1f524d4d, 0x764f3939, 0xd36ebdbd,
0x81d65757, 0xb7d86f6f, 0xeb37dc, 0x51441515, 0xa6dd7b7b, 0
x09fef7f7, 0xb68c3a3a, 0x932fbc, 0xf030c0c, 0x03fcffff, 0xc26ba9a9
, 0xba73c9c9, 0xd96cb5b5, 0xdc6db1b1, 0x375a6d6d, 0x15504545,
0xb98f3636, 0x771b6c6c, 0x13adbebe, 0xda904a4a, 0x57b9eeee, 0
xa9de7777, 0x4cbef2f2, 0x837efdfd, 0x55114444, 0xbdda6767, 0x2c5d7171
, 0x45400505, 0x631f7c7c, 0x50104040, 0x325b6969, 0xb8db6363,
0x220a2828, 0xc5c20707, 0xf531c4c4, 0xa88a2222, 0x31a79696, 0
xf9ce3737, 0x977aeded, 0x49bff6f6, 0x992db4b4, 0xa475d1d1, 0x90d34343
, 0x5a124848, 0x58bae2e2, 0x71e69797, 0x64b6d2d2, 0x70b2c2c2,
0xad8b2626, 0xcd68a5a5, 0xcb955e5e, 0x624b2929, 0x3c0c3030, 0
xce945a5a, 0xab76dddd, 0x867ff9f9, 0xf1649595, 0x5dbbe6e6, 0x35f2c7c7
, 0x2d092424, 0xd1c61717, 0xd66fb9b9, 0xdec51b1b, 0x94861212,
0x78186060, 0x30f3c3c3, 0x897cf5f5, 0x5cefb3b3, 0xd23ae8e8, 0
xacdf7373, 0x794c3535, 0xa0208080, 0x9d78e5e5, 0x56edbbbb, 0x235e7d7d
, 0xc63ef8f8, 0x8bd45f5f, 0xe7c82f2f, 0xdd39e4e4, 0x68492121,
}

func rl(x uint32, i uint8) uint32 { return (x << i) | (x >> (32 - i)) }

func l0(b uint32) uint32 { return b ^ rl(b, 13) ^ rl(b, 23) }

func feistel0(x0, x1, x2, x3, rk uint32) uint32 { return x0 ^ l0(p(x1^x2
^x3^rk)) }

func p(a uint32) uint32 {
    return (uint32(sbox[a>>24]) << 24) ^ (uint32(sbox[(a>>16)&0xff]) <<
16) ^ (uint32(sbox[(a>>8)&0xff]) << 8) ^ uint32(sbox[(a)&0xff])
}

```

```

func permuteInitialBlock(b []uint32, block []byte) {
    for i := 0; i < 4; i++ {
        b[i] = (uint32(block[i*4]) << 24) | (uint32(block[i*4+1]) << 16)
        |
        (uint32(block[i*4+2]) << 8) | (uint32(block[i*4+3]))
    }
}

func permuteFinalBlock(b []byte, block []uint32) {
    for i := 0; i < 4; i++ {
        b[i*4] = uint8(block[i] >> 24)
        b[i*4+1] = uint8(block[i] >> 16)
        b[i*4+2] = uint8(block[i] >> 8)
        b[i*4+3] = uint8(block[i])
    }
}

func load(k []byte) uint32 {
    return uint32(k[0])<<24 | uint32(k[1])<<16 | uint32(k[2])<<8 |
    uint32(k[3])
}

//encrypt block
func cryptBlock(subkeys []uint32, b []uint32, r []byte, dst, src []byte,
    decrypt bool) {
    var x uint32
    permuteInitialBlock(b, src)
    if decrypt {
        for i := 0; i < 8; i++ {
            x = b[1] ^ b[2] ^ b[3] ^ subkeys[31-4*i]
            b[0] = b[0] ^ sbbox0[x&0xff] ^ sbbox1[(x>>8)&0xff] ^ sbbox2[(x
>>16)&0xff] ^ sbbox3[(x>>24)&0xff]
            x = b[0] ^ b[2] ^ b[3] ^ subkeys[31-4*i-1]
            b[1] = b[1] ^ sbbox0[x&0xff] ^ sbbox1[(x>>8)&0xff] ^ sbbox2[(x
>>16)&0xff] ^ sbbox3[(x>>24)&0xff]
            x = b[0] ^ b[1] ^ b[3] ^ subkeys[31-4*i-2]
            b[2] = b[2] ^ sbbox0[x&0xff] ^ sbbox1[(x>>8)&0xff] ^ sbbox2[(x
>>16)&0xff] ^ sbbox3[(x>>24)&0xff]
            x = b[1] ^ b[2] ^ b[0] ^ subkeys[31-4*i-3]
            b[3] = b[3] ^ sbbox0[x&0xff] ^ sbbox1[(x>>8)&0xff] ^ sbbox2[(x
>>16)&0xff] ^ sbbox3[(x>>24)&0xff]
        }
    } else {
        for i := 0; i < 8; i++ {
            x = b[1] ^ b[2] ^ b[3] ^ subkeys[4*i]
            b[0] = b[0] ^ sbbox0[x&0xff] ^ sbbox1[(x>>8)&0xff] ^ sbbox2[(x
>>16)&0xff] ^ sbbox3[(x>>24)&0xff]
            x = b[0] ^ b[2] ^ b[3] ^ subkeys[4*i+1]
            b[1] = b[1] ^ sbbox0[x&0xff] ^ sbbox1[(x>>8)&0xff] ^ sbbox2[(x
>>16)&0xff] ^ sbbox3[(x>>24)&0xff]
            x = b[0] ^ b[1] ^ b[3] ^ subkeys[4*i+2]
            b[2] = b[2] ^ sbbox0[x&0xff] ^ sbbox1[(x>>8)&0xff] ^ sbbox2[(x
>>16)&0xff] ^ sbbox3[(x>>24)&0xff]
            x = b[1] ^ b[2] ^ b[0] ^ subkeys[4*i+3]
            b[3] = b[3] ^ sbbox0[x&0xff] ^ sbbox1[(x>>8)&0xff] ^ sbbox2[(x

```

```

    >>16)&0xff] ^ sbbox3[(x>>24)&0xff]
    }
    }
    b[0], b[1], b[2], b[3] = b[3], b[2], b[1], b[0]
    permuteFinalBlock(r, b)
    copy(dst, r)
}

func generateSubKeys(key []byte, addr uint32) []uint32 {
    current := make([]uint32, 4)
    current[0] = load(key)
    current[1] = load(key[4:])
    current[2] = load(key[8:])
    current[3] = load(key[12:])
    for i := 0; i < 4; i++ {
        cki := ck[i] ^ addr
        intval := current[3] ^ current[2] ^ current[1] ^ cki
        intval = p(intval)
        intval = intval ^ rl(intval, 13) ^ rl(intval, 23)
        intval = intval ^ current[0]

        current[0] = current[1]
        current[1] = current[2]
        current[2] = current[3]
        current[3] = intval
    }
    return current
}

func sm4e(content []uint32, rks []uint32) []uint32 {
    current := make([]uint32, 4)
    current[0] = content[0]
    current[1] = content[1]
    current[2] = content[2]
    current[3] = content[3]
    for i := 0; i < 4; i++ {
        cki := rks[3-i]

        intval := current[3] ^ current[2] ^ current[1] ^ cki
        intval = p(intval)
        intval = intval ^ rl(intval, 2) ^ rl(intval, 10) ^ rl(intval,
18) ^ rl(intval, 24)
        intval = intval ^ current[0]

        current[0] = current[1]
        current[1] = current[2]
        current[2] = current[3]
        current[3] = intval
    }
    return current
}

func (k KeySizeError) Error() string {
    return "SM4: invalid key size " + strconv.Itoa(int(k))
}

```

```
func rev(i uint32) uint32 {
    return (i >> 24) | (i>>8)&0xff00 | (i<<8)&0xff0000 | (i << 24)
}
```

```
// solve_step4.go
package main

import (
    "fmt"
    "io/ioutil"
)

var memory []byte
var forwardLUT [16][65536]uint32
var reverseLUT [16][65536][]uint32

// Clé de chiffrement SM4
var key = []byte{0x20, 0x78, 0xcc, 0x93, 0xb4, 0xc1, 0x50, 0x38, 0xd5, 0
    xdc, 0x43, 0x9c, 0x6, 0x25, 0xf8, 0x24}

func pp_mem(addr uint32) uint32 {
    offset := addr & 0xf
    addr = addr & 0xfffff0
    c0 := uint32(memory[addr])<<24 | uint32(memory[addr+1])<<16 | uint32
(memory[addr+2])<<8 | uint32(memory[addr+3])
    c1 := uint32(memory[addr+4])<<24 | uint32(memory[addr+5])<<16 |
uint32(memory[addr+6])<<8 | uint32(memory[addr+7])
    c2 := uint32(memory[addr+8])<<24 | uint32(memory[addr+9])<<16 |
uint32(memory[addr+10])<<8 | uint32(memory[addr+11])
    c3 := uint32(memory[addr+12])<<24 | uint32(memory[addr+13])<<16 |
uint32(memory[addr+14])<<8 | uint32(memory[addr+15])
    keys := generateSubKeys(key, addr)
    blk := []uint32{c0, c1, c2, c3}
    dec := sm4e(blk, keys)
    res := make([]byte, 16)
    res[0] = byte(dec[0])
    res[1] = byte(dec[0] >> 8)
    res[2] = byte(dec[0] >> 16)
    res[3] = byte(dec[0] >> 24)
    res[4] = byte(dec[1])
    res[5] = byte(dec[1] >> 8)
    res[6] = byte(dec[1] >> 16)
    res[7] = byte(dec[1] >> 24)
    res[8] = byte(dec[2])
    res[9] = byte(dec[2] >> 8)
    res[10] = byte(dec[2] >> 16)
    res[11] = byte(dec[2] >> 24)
    res[12] = byte(dec[3])
    res[13] = byte(dec[3] >> 8)
    res[14] = byte(dec[3] >> 16)
    res[15] = byte(dec[3] >> 24)
    return uint32(res[15-offset])
}
```

```

func gen_decrypted() {
    dec := make([]byte, len(memory))
    for i := uint32(0); int(i) < 0x100000; i++ {
        dec[i] = byte(pp_mem(i))
    }
    ioutil.WriteFile("blob.decrypted", dec, 0644)
}

func mem(elt uint32) uint32 {
    return uint32(memory[elt])
}

func processBlk(blk []byte, path []bool) (uint32, uint32) {
    var r0, r1, r2, r3, r7, r9, r14 uint32
    var r8 uint32

    r0 = uint32(blk[1]) | uint32(blk[0])<<8
    r1 = uint32(blk[3]) | uint32(blk[2])<<8
    r2 = uint32(blk[5]) | uint32(blk[4])<<8
    r3 = uint32(blk[7]) | uint32(blk[6])<<8

    r7 = 7
    for r14 = 0x1f; r14 < 0x20; r14-- {
        r9 = forwardLUT[r7][r1]
        r7 = decR7(r7)
        r7 = decR7(r7)
        r7 = decR7(r7)
        r7 = decR7(r7)
        if r14&0x08 == 0 {
            r8 = r3
            r3 = r14
            r3 = r3 + 1
            r3 = r3 ^ r0
            if path[r14] {
                r3 = r3 ^ r1
                r0 = r9
                r1 = r2
                r2 = r8
            }
        } else {
            r8 = r0
            r0 = r9
            r1 = r14
            r1 = r1 + 1
            r1 = r1 ^ r0
            r1 = r1 ^ r2
            if path[r14] {
                r2 = r3
                r3 = r8
            }
        }
        fmt.Println(r14)
        fmt.Println(r7)
        fmt.Printf("r0: %04x\n", r0)
        fmt.Printf("r1: %04x\n", r1)
    }
}

```

```

        fmt.Printf("r2: %04x\n", r2)
        fmt.Printf("r3: %04x\n", r3)
        fmt.Printf("r8: %04x\n", r8)

    }
    swap(r0)
    swap(r1)
    swap(r2)
    swap(r3)
    return (r1<<0x10 + r0), (r3<<0x10 + r2)
}

type state struct {
    r0  uint32
    r1  uint32
    r2  uint32
    r3  uint32
    r8  uint32
    r14 uint32
}

func (st *state) computeNext(states []*state) {
    r7s := []uint32{3, 7, 1, 5, 9, 3, 7, 1, 5, 9, 3, 7, 1, 5, 9, 3, 7,
1, 5, 9, 3, 7, 1, 5, 9, 3, 7, 1, 5, 9, 3, 7, 1, 5}
    r7 := r7s[st.r14]
    for _, guessR1 := range reverseLUT[r7][st.r0] {
        ns := &state{
            r0:  st.r3 ^ guessR1 ^ (st.r14 + 1),
            r1:  guessR1,
            r2:  st.r1,
            r3:  st.r2,
            r8:  st.r1,
            r14: st.r14 + 1,
        }
        *states = append(*states, ns)
    }
}

func (st *state) computeNextOther(states []*state) {
    r7s := []uint32{3, 7, 1, 5, 9, 3, 7, 1, 5, 9, 3, 7, 1, 5, 9, 3, 7,
1, 5, 9, 3, 7, 1, 5, 9, 3, 7, 1, 5, 9, 3, 7}
    r7 := r7s[st.r14]
    for _, guessR1 := range reverseLUT[r7][st.r0] {
        *states = append(*states, &state{
            r0:  st.r3,
            r1:  guessR1,
            r2:  st.r0 ^ (st.r14 + 1) ^ st.r1,
            r3:  st.r2,
            r8:  st.r2,
            r14: st.r14 + 1,
        })
    }
}

func (st *state) String() string {
    return fmt.Sprintf("{\nr14: %02x\nr0: %04x\nr1: %04x\nr2: %04x\nr3:

```

```

    %04x\nr8: %04x\n}", st.r14, st.r0, st.r1, st.r2, st.r3, st.r8)
}

func (st *state) Equal(other *state) bool {
    return st.r0 == other.r0 && st.r1 == other.r1 && st.r2 == other.r2
    && st.r3 == other.r3
}

func swap(i uint32) uint32 {
    return ((i & 0xff) << 8) | ((i >> 8) & 0xff)
}

func decR7(r7 uint32) uint32 {
    if r7 == 0 {
        return 0x9
    }
    return r7 - 1
}

func step(r7, r1 uint32) uint32 {
    r4 := r1
    r5 := r4 & 0xff
    r4 = r4 >> 8
    r10 := r7<<0x10 + (r5 << 8) + r4 + 0x1000
    r6 := mem(r10) & 0xff
    r7 = decR7(r7)
    // 0x08a
    r10 = r7<<0x10 + (r4 << 8) + r6 + 0x1000
    r5 = mem(r10)
    r7 = decR7(r7)
    //0x0ae
    r10 = r7<<0x10 + (r6 << 8) + r5 + 0x1000
    r4 = mem(r10)
    r7 = decR7(r7)
    // 0x0d2
    r10 = r7<<0x10 + (r5 << 8) + r4 + 0x1000
    r6 = mem(r10)
    r7 = decR7(r7)
    // 0x0f6
    return r6<<8 + r4
}

func genLuts() {
    for r7 := uint32(0); r7 < 16; r7++ {
        for r1 := uint32(0); r1 < 65536; r1++ {
            r9 := step(r7, r1)
            forwardLUT[r7][r1] = r9
            reverseLUT[r7][r9] = append(reverseLUT[r7][r9], r1)
        }
    }
}

func main() {
    // Déchiffrement du blob
    memory, _ = ioutil.ReadFile("blob.bin")
    gen_decrypted()
}

```

```

// Génération de Look-up tables pour améliorer les performances
memory, _ = ioutil.ReadFile("blob.decrypted")
genLuts()
fmt.Println("generated LUTs")

res := make([]uint16, 0)

// Valeur de référence
bfList := make([]*state, 0)
bfList = append(bfList, &state{
    r0: 0x7072,
    r1: 0x2e61,
    r2: 0x2e72,
    r3: 0x6667,
})
bfList = append(bfList, &state{
    r0: 0x2e63,
    r1: 0x6e66,
    r2: 0x2e66,
    r3: 0x762e,
})
bfList = append(bfList, &state{
    r0: 0x736e,
    r1: 0x7076,
    r2: 0x7972,
    r3: 0x4066,
})
bfList = append(bfList, &state{
    r0: 0x6667,
    r1: 0x7670,
    r2: 0x2e62,
    r3: 0x6574,
})

for _, ini := range bfList {
    initial := &state{
        r0:  uint32(ini.r0),
        r1:  uint32(ini.r1),
        r2:  uint32(ini.r2),
        r3:  uint32(ini.r3),
        r8:  uint32(ini.r2),
        r14: 0,
    }

    states := make([]*state, 0)
    states = append(states, initial)
    for i := 0; i < 8 && len(states) > 0; i++ {
        tmp := states
        states = make([]*state, 0)
        for _, st := range tmp {
            st.computeNext(&states)
        }
    }
    fmt.Println("End first chunk")
    for i := 0; i < 8 && len(states) > 0; i++ {
        tmp := states
    }
}

```

```

        states = make([]*state, 0)
        for _, st := range tmp {
            st.computeNextOther(&states)
        }
    }
    fmt.Println("End second chunk")
    for i := 0; i < 8 && len(states) > 0; i++ {
        tmp := states
        states = make([]*state, 0)
        for _, st := range tmp {
            st.computeNext(&states)
        }
    }
    fmt.Println("End third chunk")
    for i := 0; i < 8 && len(states) > 0; i++ {
        tmp := states
        states = make([]*state, 0)
        for _, st := range tmp {
            st.computeNextOther(&states)
        }
    }
    fmt.Println("End fourth chunk")
    fmt.Printf("%d possible solutions\n", len(states))

    for _, st := range states {
        fmt.Println(st)
        res = append(res, uint16(st.r0))
        res = append(res, uint16(st.r1))
        res = append(res, uint16(st.r2))
        res = append(res, uint16(st.r3))
    }
}

fmt.Printf("Solution is %04x%04x%04x%04x%04x%04x%04x%04x%04x%04x%04x%04x%04x%04x%04x\n", res[0], res[1], res[2], res[3], res[4], res[5], res[6], res[7], res[8], res[9], res[10], res[11], res[12], res[13], res[14], res[15])
}

```