Solution pour le Challenge SSTIC 2012

Éric Alata[1,2] et Fernand Lone Sang[1,2]
{eric.alata,fernand.lone-sang}(at)laas(dot)fr
[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2] Université de Toulouse, INSA, LAAS, F-31400 Toulouse, France

29 mai 2012

1 Introduction

Le défi de cette année consiste à analyser une image de disque d'environ 1 gigaoctets et de retrouver, au sein de celle-ci, une adresse e-mail qui permet de valider notre réponse au concours. Cet article présente la démarche que nous avons appliquée pour retrouver cet adresse e-mail. Nous présentons les différentes approches que nous avons explorées : celles qui se sont soldées par des succès, mais également celles qui ont échoué et que nous trouvons néanmoins intéressantes. Dans un souci de clarté, cet article ne détaille pas l'ensemble des codes sources obtenus au fur et à mesure de notre avancement. Nous préférons décrire les grandes lignes des algorithmes implémentés et introduire les informations nécessaires au lecteur pour appréhender ces codes sources, fournis en annexe. Leur compréhension est laissée en exercice aux lecteurs curieux. Pour terminer, il a nous semblé intéressant d'indiquer, sous la forme d'annotations, les compétences nécessaires pour résoudre les nombreux problèmes auxquels nous avons été confrontés. Ces annotations guident ainsi la lecture sur les facettes multiples de ce défi.

2 Analyse du fichier *challenge*

L'épreuve de cette année débute par la récupération d'un fichier nommé *challenge* sur le site Internet de la communauté SSTIC [18]. Dans cette section, nous détaillons les informations que nous avons obtenu sur le contexte du défi de cette année suite à l'analyse de ce fichier.

2.1 Extraction de l'image de disque et association à un point de montage

Lorsqu'on est confronté à un fichier sur lequel on n'a aucune information, il est d'usage, en premier lieu, de déterminer son type afin d'identifier les outils qui permettront de le manipuler par la suite. Comme nous travaillons dans un environnement UNIX, nous nous aidons de la commande *file* pour identifier le type du fichier *challenge*. D'après la description fournie, il s'agit d'une archive compressée avec l'algorithme GZIP [3] d'un fichier initialement nommé *dump.img*. Nous pouvons alors renommer le fichier *challenge* en *dump.img.gz*, puis le décompresser avec la commande *gzip* pour retrouver un fichier rigoureusement identique à l'original (cf. listing 1).

```
sh-3.2$ file challenge
challenge: gzip compressed data, was "dump.img", from Unix, [...]
sh-3.2$ mv challenge dump.img.gz
sh-3.2$ gzip -d dump.img.gz
sh-3.2$ file dump.img
dump.img: x86 boot sector; partition 1: ID=0x83, active, starthead 1, startsector 63, [...]
```

Listing 1 – Analyse préliminaire du fichier *challenge*

La commande *file* sur le fichier *dump.img* nous indique que ce dernier correspond à l'image d'un disque dur. Il s'agit vraisemblablement de l'image de disque qu'il nous est demandé d'analyser dans la brève description de l'épreuve. Il convient de remarquer que la chaîne « *x86 boot sector* » peut induire en erreur un analyste en laissant penser qu'il s'agit d'une image de disque issue d'une machine de type PC. Nous verrons dans la suite que ce n'est pas le cas.

Nous lions l'image de disque à un périphérique virtuel, puis nous l'associons à un point de montage afin d'explorer son contenu (cf. listing 2). La première partition correspond à une partition Linux étendue (ou *ext*), et contient le système de fichiers racine (en anglais, *rootfs*) d'une machine de type UNIX. Comme le périphérique virtuel */dev/loop0p1* n'existe pas, il a été nécessaire de le créer. Nous lions, à cet effet, la première partition à un nouveau périphérique virtuel en précisant le décalage de son premier secteur par rapport au début du disque. Ce décalage s'exprime en octets et est calculé en fonction de la géométrie du disque : 63 (secteurs) ∗ 512 (octets / secteur), soit 32256 octets.

Admin. de systèmes UNIX

```
sh-3.2$ sudo modprobe loop
sh-3.2$ sudo losetup /dev/loop0 dump.img
sh-3.2$ sudo fdisk -l /dev/loop0
Disque /dev/loop0: 1073 Mo, 1073741824 octets
255 tetes, 63 secteurs/piste, 130 cylindres, total 2097152 secteurs
Unites = secteur de 1 * 512 = 512 octets
Taille de secteur (logique / physique) : 512 octets / 512 octets
taille d'E/S (minimale / optimale) : 512 octets / 512 octets
Identifiant de disque : 0xcf660900
Peripherique Amorce  Debut        Fin      Blocs     Id  Systeme
/dev/loop0p1   *          63     2088449     1044193+  83  Linux
sh-3.2$ sudo losetup -d /dev/loop0
sh-3.2$ sudo losetup -o `python -c 'print 63*512'` /dev/loop0 dump.img
sh-3.2$ sudo mount -o ro /dev/loop0 /mnt
sh-3.2$ ls /mnt
bin  boot  dev  etc  home  lib  lost+found  media  mnt  opt  proc  [...]
```

Listing 2 – Association du fichier *dump.img* au point de montage */mnt*

2.2 Exploration du système de fichiers racine

Notre analyse se poursuit par l'exploration du système de fichiers nouvellement chargé. L'image de disque est issue d'une machine qui se nomme *sstic-host* et qui repose sur un système d'exploitation Debian [14] « Lenny ». Le fichier */etc/passwd* révèle un utilisateur intéressant : « *sstic* ». Nous apprenons davantage sur le contexte de l'épreuve proposée cette année en consultant son répertoire personnel.

```
sh-3.2$ cat /mnt/etc/hostname
sstic-host
sh-3.2$ head -n4 /mnt/etc/apt/source.list
# deb http://ftp.fr.debian.org/debian/ lenny main

deb http://ftp.fr.debian.org/debian/ lenny main
deb-src http://ftp.fr.debian.org/debian/ lenny main
sh-3.2$ tail -n1 /mnt/etc/passwd
sstic:x:1000:1000:sstic,,,:/home/sstic:/bin/bash
sh-3.2$ ls /mnt/home/sstic/
total 1344
-rw-r--r-- 1 root     root           871 23 mars  09:51 irc.log
-rwxr-xr-x 1 root     root       1048592 23 mars  09:29 secret
-rwxr-xr-x 1 root     root           128 23 mars  09:30 ssticrypt
```

Listing 3 – Exploration du système de fichiers racine

Le répertoire personnel de l'utilisateur *sstic* contient trois fichiers. Le fichier *irc.log* est un extrait d'une conversation entre *lobster_dog* et *blue_footed_booby* sur le canal IRC *#sstic-challenge*. Leur conversation nous révèle que l'image de disque que nous analysons appartient à *blue_footed_booby*, qui s'est proposé de garder un fichier, probablement *secret*, à l'abri de *lobster_cat*. Le fichier *ssticrypt* correspond probablement au système de chiffrement également mentionné dans leur conversation (cf. listing 4).

```
sh-3.2$ cat /mnt/home/sstic/irc.log
#sstic-challenge: <lobster_dog> I've a problem
#sstic-challenge: <lobster_dog> lobster cat is trying to steal one of my files
#sstic-challenge: <blue_footed_booby> lobster cat ?
#sstic-challenge: <lobster_dog> http://amazingdata.com/mediadata56/Image/ [...]
#sstic-challenge: <blue_footed_booby> k -_-
#sstic-challenge: <blue_footed_booby> gimme your file, I'll hide it
#sstic-challenge: <lobster_dog> k
#sstic-challenge: <blue_footed_booby> your data is secure ;) I just finished the encryption
    system
#sstic-challenge: <lobster_dog> ok, I secure erase it on my side!
```

```
#sstic-challenge: <blue_footed_booby> unfortunately my hard drive is making strange noises
    right now ... :(
-!- blue_footed_booby [~booby@galapagos] has quit [Read error: Connection reset by peer]
#sstic-challenge: <lobster_dog> ...
sh-3.2$ file /mnt/home/sstic/*
home/sstic/irc.log:   ASCII text
home/sstic/secret:    COM executable for DOS
home/sstic/ssticrypt: ELF 32-bit MSB executable, MIPS, MIPS-I version 1 [...]
sh-3.2$ file /mnt/bin/bash
/mnt/bin/bash: ELF 32-bit MSB executable, MIPS, MIPS-I version 1 (SYSV), [...]
```

Listing 4 – Analyse du contenu du répertoire */home/sstic/*

En approfondissant notre analyse, il s'avère que le fichier *ssticrypt* est un programme exécutable au format ELF [19] destiné à être exécuté sur un processeur MIPS [12]. Nous pouvons supposer que le système d'exploitation de la machine *sstic-host* s'exécute sur un processeur MIPS. L'analyse des fichiers exécutables stockés dans le répertoire */mnt/bin/* conforte cette hypothèse.

2.3 Exécution de l'image du disque *dump.img* dans une machine virtuelle

Nous avons recueilli suffisamment d'informations sur l'image de disque *dump.img*. Nous avons également ciblé deux fichiers intéressants à analyser dynamiquement dans leur environnement d'exécution : les fichiers *secret* et *ssticrypt*. Nous allons exécuter, à présent, le système d'exploitation installé sur l'image de disque dans une machine virtuelle et analyser plus finement ces deux fichiers.

Admin. de systèmes UNIX

Pour émuler une plate-forme MIPS, nous avons utilisé le gestionnaire de machines virtuelles *qemu*. Il nécessite cependant qu'on lui précise l'emplacement d'un noyau de système d'exploitation à utiliser pour démarrer le système. Nous n'en trouvons malheureusement pas sur l'image de disque. Nous constatons en effet que le répertoire */boot* est vide. Quelques recherches sur Internet nous mènent rapidement au site d'un développeur Debian, Aurélien Jarno [1]. Son site contient plusieurs versions du noyau Linux déjà compilés, des images de disque Debian/MIPS et détaille les paramètres qu'il faut préciser au gestionnaire de machines virtuelles *qemu* pour les utiliser.

```
sh-3.2$ wget http://people.debian.org/~aurel32/qemu/mips/vmlinux-2.6.32-5-4kc-malta
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 6765k  100 6765k    0     0   513k      0  0:00:13  0:00:13 --:--:--  600k
sh-3.2$ qemu-system-mips -M malta -kernel vmlinux-2.6.32-5-4kc-malta -hda dump.img -snapshot
    -append "root=/dev/sda1 console=ttyS0 init=/bin/sh" -no-reboot -nographic
QEMU 1.0,1 monitor - type 'help' for more information
(qemu) QEMU 1.0,1 monitor - type 'help' for more information
(qemu)
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[...]
```

Listing 5 – Chargement du disque *dump.img* dans *qemu*

Il convient de noter que nous avons utilisé *qemu* avec l'option `-snapshot` afin qu'il ne répercute pas les changements sur l'image de disque, suite à l'exécution du système. Ces changements pourraient en effet fausser nos futures analyses, voire faire disparaitre des informations importantes dont nous n'avons pas encore pris connaissance à ce stade du concours.

```
sh-3.2# cd /home/sstic/
sh-3.2# ./ssticrypt
sh: ./ssticrypt: Input/output error
sh-3.2# readelf -h ssticrypt
readelf: Error: Unable to read in 0x28 bytes of section headers
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x400c40
```

```
  Start of program headers:          52 (bytes into file)
  Start of section headers:          305184 (bytes into file)
  Flags:                             0x1007, noreorder, pic, cpic, o32, mips1
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         39
  Section header string table index: 36
readelf: Error: Unable to read in 0x618 bytes of section headers
readelf: Error: Unable to read in 0x100 bytes of program headers
```

Listing 6 – Première exécution du programme *ssticrypt* et analyse de l'erreur d'entrées-sorties

Le listing 6 montre malheureusement que l'exécutable *ssticrypt* est corrompu. En effet, son exécution conduit directement à une erreur d'entrées-sorties. Les bruits étranges de disque évoqués par *blue_footed_booby* dans sa conversation avec *lobster_dog* prennent alors du sens : son disque était vraisemblablement défectueux. La prochaine étape consiste logiquement à retrouver, sur le disque, une ancienne copie de *ssticrypt* que l'on puisse exécuter et analyser. Nous utilisons, à cet effet, des outils généralement utilisés dans le cadre d'investigations numériques légales. Cela fait l'objet de la prochaine section.

3 Recherche et reconstruction du fichier *ssticrypt*

La présente section décrit notre démarche pour retrouver une copie du fichier *ssticrypt* sur le disque, puis comment à partir de celle-ci, nous sommes parvenus à reconstruire le programme exécutable initial.

3.1 Recherche d'une copie de *ssticrypt*

Les outils de récupération de données, tels que *PhotoRec* [7] et *Hachoir* [8], permettent de retrouver efficacement la majorité des fichiers perdus sur un disque. Dans le cadre de nos investigations, nous préférons utiliser l'outil *PhotoRec*. Le fait que *dump.img* utilise un système de fichiers *ext2* facilite énormément la récupération de fichiers. Il aurait été plus ardu de récupérer des fichiers, par exemple, sur un système de fichiers *ext3* ou *ext4* à cause de leur principe de fonctionnement.

Inforensique

```
sh-3.2$ photorec dump.img
PhotoRec 6.13, Data Recovery Utility, November 2011
Christophe GRENIER <grenier@cgsecurity.org>
http://www.cgsecurity.org

Disk dump.img - 1073 MB / 1024 MiB (RO)
     Partition                  Start        End    Size in sectors
 1 * Linux                    0   1  1   129 254 63    2088387


4156 files saved in /tmp/recup_dir directory.
Recovery completed.
[...]
```

Listing 7 – Récupération de documents avec *PhotoRec*

Afin de ne pas manquer des fichiers et d'obtenir des résultats les plus complets possibles, il est important de modifier la manière dont la géométrie du disque est déterminée par *PhotoRec* [1] et de configurer ce dernier de façon à ce qu'il garde les fichiers récupérés même s'ils sont endommagés [2]. Comme la récupération se fait sur un système de fichiers de type *ext*, il est également important d'inclure ce format de fichier dans notre recherche [3]. L'outil *PhotoRec* retrouve ainsi 4156 fichiers en une dizaine de secondes, parmi lesquels se trouve une version plus complète du fichier *ssticrypt*. Les fichiers récupérés sont stockés dans différents répertoires préfixés par la chaîne de caractères *recup_dir*.

La recherche d'une copie valide de *ssticrypt* se fait par raffinements successifs : nous identifions une suite d'octets caractéristique de *ssticrypt* à partir de la version endommagée, et nous les utilisons comme motif de recherche pour tester les 4156 fichiers candidats. Si le nombre de candidats potentiels est trop important, nous utilisons un motif plus grand et nous réitérons la recherche jusqu'à arriver à un nombre de candidats réduit.

1. Par l'activation de l'option « *Allow partial last cylinder* ».
2. Par l'activation de l'option « *Keep corrupted files* ».
3. Par la selection des formats de fichier « *ext2/ext3/ext4 Superblock* » et « *ext2/ext3/ext4 Filesystem* ».

```
sh-3.2$ xxd -a -g1 ssticrypt | head -n 5
0000000: 7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00  .ELF............
0000010: 00 02 00 08 00 00 00 01 00 40 0c 40 00 00 00 34  .........@.@...4
0000020: 00 04 b4 20 00 00 10 07 00 34 00 20 00 08 00 28  ... .....4. ...(
0000030: 00 27 00 24 00 00 00 06 00 00 00 34 00 40 00 34  .'.$.......4.@.4
0000040: 00 40 00 34 00 00 01 00 00 00 01 00 00 00 00 05  .@.4............
sh-3.2$ grep -o -rasbP '\x00\x04\xA8\x20\x00\x00\x10\x07' recup_dir.*
./recup_dir.1/f0163840.elf:32:........
sh-3.2$ mipsel-linux-gnu-readelf -h ./recup_dir.1/f0163840.elf
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x400c40
  Start of program headers:          52 (bytes into file)
  Start of section headers:          305184 (bytes into file)
  Flags:                             0x1007, noreorder, pic, cpic, o32, mips1
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         39
  Section header string table index: 36
readelf: Error: Unable to read in 0xe130b03 bytes of string table
readelf: Error: no .dynamic section in the dynamic segment
```

Listing 8 – Recherche de *ssticrypt* sur l'image de disque *dump.img*

Les octets à partir de l'adresse `0x20` constituent un motif de recherche intéressant. En effet, ils nous permettent d'isoler un unique fichier candidat pour *ssticrypt* : le fichier *f0163840.elf*. L'analyse de ce fichier nous également donne une erreur. Cependant, celle-ci est différente de celle précédemment obtenue : elle indique visiblement des erreurs dans les sections du fichier. Nous rémédions à cette erreur dans la sous-section suivante.

3.2 Reconstruction des sections du fichier *ssticrypt*

Le format ELF (*Executable and Linkable Format*) [19] est un format de fichier généralement associé à du code compilé pour les systèmes UNIX. Par exemple, les fichiers objets, les programmes exécutables ou les bibliothèques de fonctions sont stockés sur les disques dans ce format. Il est constitué d'un entête fixe, puis de plusieurs segments destinés à être chargés en mémoire, eux-mêmes découpés en plusieurs sections (cf. figure 1). Au sein de cet entête fixe, un premier entête décrit le type de fichier et définit globalement son organisation. La *Program Header Table* constitue une seconde partie de l'entête et liste les différents segments de code à charger en mémoire. Finalement, la *Section Header Table* décrit les différentes sections présentes dans le fichier exécutable.

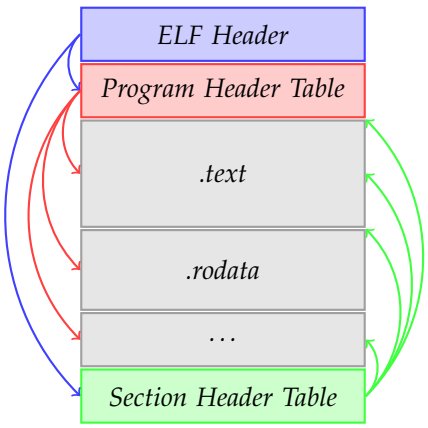

FIGURE 1 – Le format de fichier ELF

Puisque l'erreur constatée se situe au niveau des sections, la suite de cette sous-section se focalise sur la *Section Header Table*. Nous tentons de localiser cette table dans le fichier *f0163840.elf*, puis nous analysons ses entrées. Le format des entrées de cette table ainsi qu'une brève description de ses champs les plus importants sont précisés dans le listing 9.

```
typedef struct {
    Elf32_Word sh_name;      // Offset du nom de la section dans .symtab
    Elf32_Word sh_type;      // Type de la section
    Elf32_Word sh_flags;     // Drapeaux pour la section
    Elf32_Addr sh_addr;      // Adresse memoire de la section
    Elf32_Off  sh_offset;    // Adresse de la section dans le fichier
    Elf32_Word sh_size;      // Taille de la section
    Elf32_Word sh_link;
    Elf32_Word sh_info;
    Elf32_Word sh_addralign;
    Elf32_Word sh_entsize;
} Elf32_Shdr;
```

Listing 9 – Structure `Elf32_Shdr` décrivant une entrée de la *Section Header Table*

Intuition

Afin de localiser la *Section Header Table* au sein du fichier *f0163840.elf*, nous avons procédé par analogie. Nous avons crée un nouvel exécutable, puis nous avons identifié quelques signatures caractéristiques sa *Section Header Table*. Nous avons ensuite utilisé ces signatures comme motifs de recherche dans *f0163840.elf*. Cette stratégie s'est avérée payante car nous avons pu localisé la *Section Header Table* à l'adresse `0x4b420` dans le fichier *f0163840.elf* (cf. listing 10). En effet, nous retrouvons la première entrée constituée entièrement d'octets nuls, puis les entrées relatives aux différentes sections (`.symtab`, etc.).

```
sh-3.2$ mipsel-linux-gnu-readelf -h /mnt/bin/bash
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x415c40
  Start of program headers:          52 (bytes into file)
  Start of section headers:          1090896 (bytes into file)
  Flags:                             0x1007, noreorder, pic, cpic, o32, mips1
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         10
  Size of section headers:           40 (bytes)
  Number of section headers:         32
  Section header string table index: 31
sh-3.2$ xxd -a -g1 -s 1090896 /mnt/bin/bash
010a550: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
010a560: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
010a570: 00 00 00 00 00 00 00 00 00 00 00 0b 00 00 00 01  ................
010a580: 00 00 00 02 00 40 01 74 00 00 01 74 00 00 00 0d  .....@.t...t....
010a590: 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00  ................
sh-3.2$ xxd -a -g1 ./recup_dir.1/f0163840.elf | grep "00 00 00 1b 00 00 00 01"
004b440: 00 00 00 00 00 00 00 00 00 00 00 1b 00 00 00 01  ................
sh-3.2$ python -c 'print hex(0x004b440 - 4*8)'
0x4b420
sh-3.2$ xxd -a -g1 -s 0x4b420 ./recup_dir.1/f0163840.elf
004b420: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004b430: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004b440: 00 00 00 00 00 00 00 00 00 00 00 1b 00 00 00 01  ................
004b450: 00 00 00 02 00 40 01 34 00 00 01 34 00 00 00 0d  .....@.4...4....
004b460: 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00  ................
```

Listing 10 – Recherche de la *Section Header Table* dans *f0163840.elf*

L'analyse minutieuse les adresses des sections décrites dans cette table révèle qu'elles sont toutes décalées de 3072 octets (*0xc00* en hexadécimal). Deux scénarios sont alors envisageables : a) soit les organisateurs ont délibérément altérés toutes les entrées de cette table, soit b) le décalage provient du fait que le programme exécutable a été stocké sur des secteurs du disque non-contigüs. Nous commençons par explorer cette seconde piste à l'aide d'un éditeur hexadécimal.

```
sh-3.2$ xxd -a -g1 -s 0xbff0 ./recup_dir.1/f0163840.elf | head -n5
000bff0: 4c 37 39 4c 0a 4c 39 32 4c 0a 4c 36 36 4c 0a 4c  L79L.L92L.L66L.L
000c000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
000cc00: 38 37 4c 0a 4c 37 34 4c 0a 4c 38 30 4c 0a 4c 36  87L.L74L.L80L.L6
000cc10: 38 4c 0a 4c 39 33 4c 0a 4c 37 33 4c 0a 4c 38 36  8L.L93L.L73L.L86
```

Listing 11 – Octets nuls à supprimer dans le fichier *f0163840.elf*

Intuition

En consultant le contenu du fichier *f0163840.elf*, nous constatons une longue série d'octets nuls à partir de l'adresse `0xc000` (cf. listing 11). Ces octets nuls nous paraissent rapidement suspects, d'une part, car ils sont compris entre deux blocs d'octets corrélés et d'autre part, parce qu'il y a autant d'octets nuls que de décalage pour chaque section. Ces octets proviennent probablement du fait que le programme exécutable *ssticrypt* n'était pas stocké sur des secteurs de disque contigüs. Explorer la seconde piste a donc été un bon choix. Nous décidons de supprimer ces octets superflus en espérant ainsi tomber sur une version de *ssticrypt* non-corrompue (cf. listing 12).

```
sh-3.2$ cp ./recup_dir.1/f0163840.elf ./recup_dir.1/f0163840.elf.cropped
[ suppression des octets 0xc000-0xcc00 dans ./recup_dir.1/f0163840.elf.cropped ]
sh-3.2$ xxd -a -g1 -s 0xbff0 ./recup_dir.1/f0163840.elf.cropped
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x400c40
  Start of program headers:          52 (bytes into file)
  Start of section headers:          305184 (bytes into file)
  Flags:                             0x1007, noreorder, pic, cpic, o32, mips1
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         39
  Section header string table index: 36
sh-3.2$ ./recup_dir.1/f0163840.elf.cropped
--> SSTICRYPT <--
usage: ./f0163840.elf.cropped [-d|-e] <key> <secure container>
    -d: uncrypt
    -e: crypt
sh-3.2$ mv ./recup_dir.1/f0163840.elf.cropped ssticrypt
```

Listing 12 – Reconstitution du programme exécutable *ssticrypt*

Notre hypothèse au sujet de *ssticrypt* (cf. section 2) est confirmée : le programme exécutable correspond effectivement au système de chiffrement mis au point par *blue_footed_booby*. Ce système de chiffrement repose sur une clé de 128 bits, autrement dit 32 chiffres hexadécimaux. Nous remarquons qu'un programme exécutable *python* est invoqué par *ssticrypt* au moment du déchiffrement (cf. listing 13). Dans la prochaine section, nous déterminons d'où le programme *python* provient et à quelle fin il est utilisé.

```
sh-3.2$ ./ssticrypt -d AAAAAA secret
Error: key should be a 128-bits hexadecimal string
sh-3.2$ ./ssticrypt -e `python -c 'print "A"*32'` /bin/bash
Using keys AAAAAAAAAAAAAAAA / AAAAAAAAAAAAAAAA ...
sh-3.2$ ./ssticrypt -d `python -c 'print "A"*32'` secret
Using keys AAAAAAAAAAAAAAAA / AAAAAAAAAAAAAAAA ...
Traceback (most recent call last):
  File "check.py", line 50107, in <module>
    AssertionError
```

Listing 13 – Chiffrement et déchiffrement en utilisant *ssticrypt* et une clé quelconque

4 Rétro-conception du système de chiffrement *ssticrypt*

Rétro-ingénierie

A présent que le programme exécutable *ssticrypt* a été reconstitué, nous nous concentrons sur sa rétro-conception. Cette approche nous a permis, d'une part, identifier l'algorithme de chiffrement implémenté dans le système de chiffrement *ssticrypt* et, d'autre part, de comprendre la fonction du programme exécutable *python* invoqué lors de son exécution. Pour des raisons de clarté, nous ne présentons pas, dans la suite de cette section, le programme *ssticrypt* décompilé. Le lecteur est invité à se reporter à l'annexe A pour consulter le code source en C obtenu après décompilation (manuelle) de son code assembleur, aidé du de-assembleur Hex-Rays IDA [9].

4.1 Chiffrement d'un fichier

L'algorithme de chiffrement implémenté dans *ssticrypt* est réprésentée sur la figure 2. Nous constatons que le chiffrement s'effectue selon deux étapes. Dans un premier temps, les octets du fichier spécifié sont chiffrés avec l'algorithme de chiffrement symétrique RC4 [17]. Une transformation, reposant sur la fonction *xor*, est ensuite appliquée à ces octets afin d'obtenir le message chiffré. Le résultat est écrit dans un fichier dont l'entête est constituée de l'empreinte des octets après chiffrement, calculée avec l'algorithme MD5 [15]. Il convient de noter que le chiffrement des octets, ainsi que le calcul de leur empreinte invoquent directement la bibliothèque de fonctions OpenSSL [13].

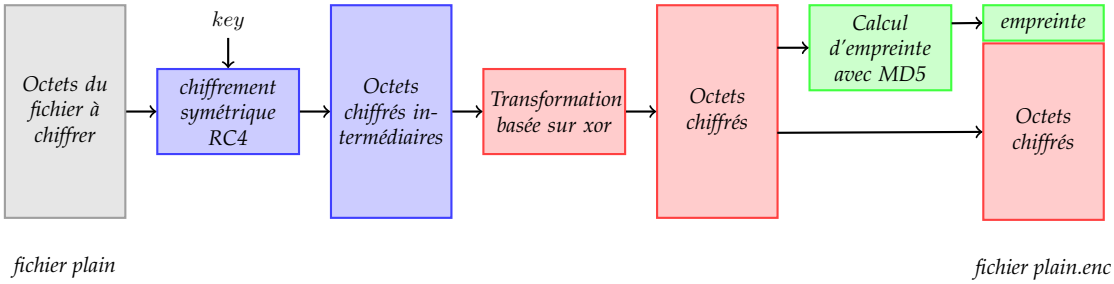

FIGURE 2 – Algorithme de chiffrement implémenté dans *ssticrypt*

4.2 Déchiffrement d'un fichier

La figure 3 présente l'algorithme de déchiffrement implémenté dans *ssticrypt*. Nous constatons qu'il correspond quasiment à la transformation inverse de l'opération de chiffrement : une transformation reposant sur la fonction *xor* est tout d'abord appliquée avant de déchiffrer les octets obtenus avec l'algorithme de chiffrement symétrique RC4. Une différence est cependant à noter : une fonction *check_key()* est insérée avant le déchiffrement du fichier. Cette fonction sert probablement à vérifier que la clé saisie est bien celle utilisée pour déchiffrer le fichier *secret*.

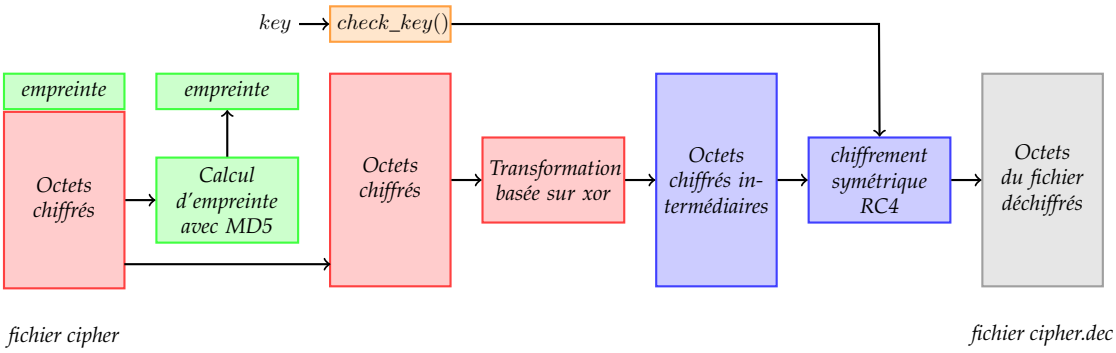

FIGURE 3 – Algorithme de déchiffrement implémenté dans *ssticrypt*

Nous nous intéressons à présent au processus de vérification de la clé implémenté dans *check_key()*. La figure 4 décrit différentes étapes qui le constitue. Nous remarquons que la vérification de la clé repose d'une part sur un programme exécutable *python*, lequel vérifie les 64 bits de poids fort de la clé,

et d'autre part sur un périphérique USB, à priori une webcam USRobotics, pour les 64 bits de poids faibles restants. Ces 64 bits restants sont vérifiés par morceaux de 32 bits (soit 4 octets). Pour cette seconde partie, à chaque fois qu'un morceau de 4 octets est vérifié, *ssticrypt* déchiffre des données qui sont utilisés par la suite pour vérifier les 4 octets suivants. Les données déchiffrés sont ensuite copiés par la fonction *load_layer()* dans la mémoire partagée avec le périphérique USB, laquelle est pointée par la variable *ram*. Une fois la clé de déchiffrement trouvée, nous retrouvons la chaîne de caractères `"V29vdCAhISBTbWVsbHMgZ29vZCA6KQ=="` dans la mémoire partagée avec le périphérique USB. Il s'agit d'une chaîne encodée en base64 [11] correspondant à « *Woot ! ! Smells good :)* » une fois décodée.

5 Vérification des 64 bits de poids fort de la clé

La rétro-ingénierie du programme exécutable *ssticrypt* a mis en évidence un processus de vérification de la clé de déchiffrement en deux temps. Dans un premier temps, les 64 bits de poids fort de la clé de déchiffrement sont vérifiés par un programme exécutable *check.pyc*. Cette section est dédiée à l'analyse de cet exécutable. Puis, les 64 bits restants sont vérifiés à l'aide d'un périphérique USB. L'algorithme de vérification des ces 64 autres bits fait l'objet de la prochaine section.

5.1 Extraction du programme exécutable *check.pyc*

La vérification des 64 bits de poids fort de la clé de déchiffrement fournie en paramètre à *ssticrypt* repose sur un programme exécutable écrit dans le langage *python*. À chaque déchiffrement, *ssticrypt* crée un fichier exécutable *check.pyc*, l'exécute, puis l'efface aussitôt l'exécution achevée. Nous allons l'extraire de *ssticrypt* afin de faciliter son analyse. Pour cela, il est possible d'adopter plusieurs stratégies.

La première stratégie consiste à utiliser un dé-assembleur tel que *Hex-Rays IDA* [9], puis de copier les octets relatifs au programme exécutable *check.pyc* dans un nouveau fichier. Cela suppose que ces octets ne sont pas modifiés lors de l'exécution de *ssticrypt*. À la lecture du code associé à *ssticrypt*, nous constatons que ce n'est effectivement pas le cas. Cette stratégie est donc applicable.

La seconde stratégie repose sur l'utilisation d'un débogueur tel que *gdb* [6]. Elle consiste à forcer l'arret de l'exécution du programme avant que le fichier extrait ne soit effacé du disque. Il suffit, pour cela, de placer un point d'arret à l'adresse de l'instruction correspondante. Nous avons alors le temps nécessaire pour effectuer une copie du fichier *check.pyc* extrait.

La dernière façon d'extraire le fichier *check.pyc* s'appuie sur des fonctionnalités fournies par l'éditeur de lien. Nous pensons en particulier à la variable d'environnement `LD_PRELOAD` et la fonctionnalité de chargement de bibliothèques partagées. C'est la stratégie que nous avons adopté dans cette épreuve parce qu'elle a le mérite d'être simple à mettre en œuvre.

`LD_PRELOAD` est une variable d'environnement qui précise à l'éditeur de lien les bibliothèques partagées à charger en plus de celles utilisées par le programme exécutable. Lorsque cette variable est définie, l'éditeur de lien charge automatiquement la bibliothèque spécifiée avant toutes les autres. Cela permet, entre autres, de surcharger quelques fonctions de la librairie C que le programme exécutable invoque. Davantage de cas d'utilisation de la variable d'environnement `LD_PRELOAD` sont discutées dans [5].

Prog. système

Pour empêcher le programme exécutable *ssticrypt* d'effacer le fichier extrait *check.pyc*, nous surchargeons la fonction *unlink()* par une fonction de notre choix. Nous décidons de la surcharger avec une fonction vide, c'est-à-dire une fonction qui n'effectue aucune action (cf. listing 14).

```
sh-3.2$ echo "int unlink(const char* path) { return 0; }" > hook.c
sh-3.2$ gcc -Wall -fpic -shared hook.c -o hook.so
sh-3.2$ LD_PRELOAD=./hook.so ./ssticrypt -d `python -c "print 'A'*32"` secret
--> SSTICRYPT <--
Using keys AAAAAAAAAAAAAAAA / AAAAAAAAAAAAAAAA ...
Traceback (most recent call last):
  File "check.py", line 50107, in <module>
  AssertionError
sh-3.2$ ls -al check.pyc
-rwxr-xr-x 1 root root 279933 May 25 21:10 check.pyc
```

Listing 14 – Extraction du programme exécutable *check.pyc* en utilisant `LD_PRELOAD`

5.2 Décompilation de *check.pyc*

Après avoir récupéré le fichier *pyc*, une première manipulation consiste à l'exécuter. Une première exécution est présentée sur le listing 15. Le programme nécessite, en paramètre, une chaîne de caractères

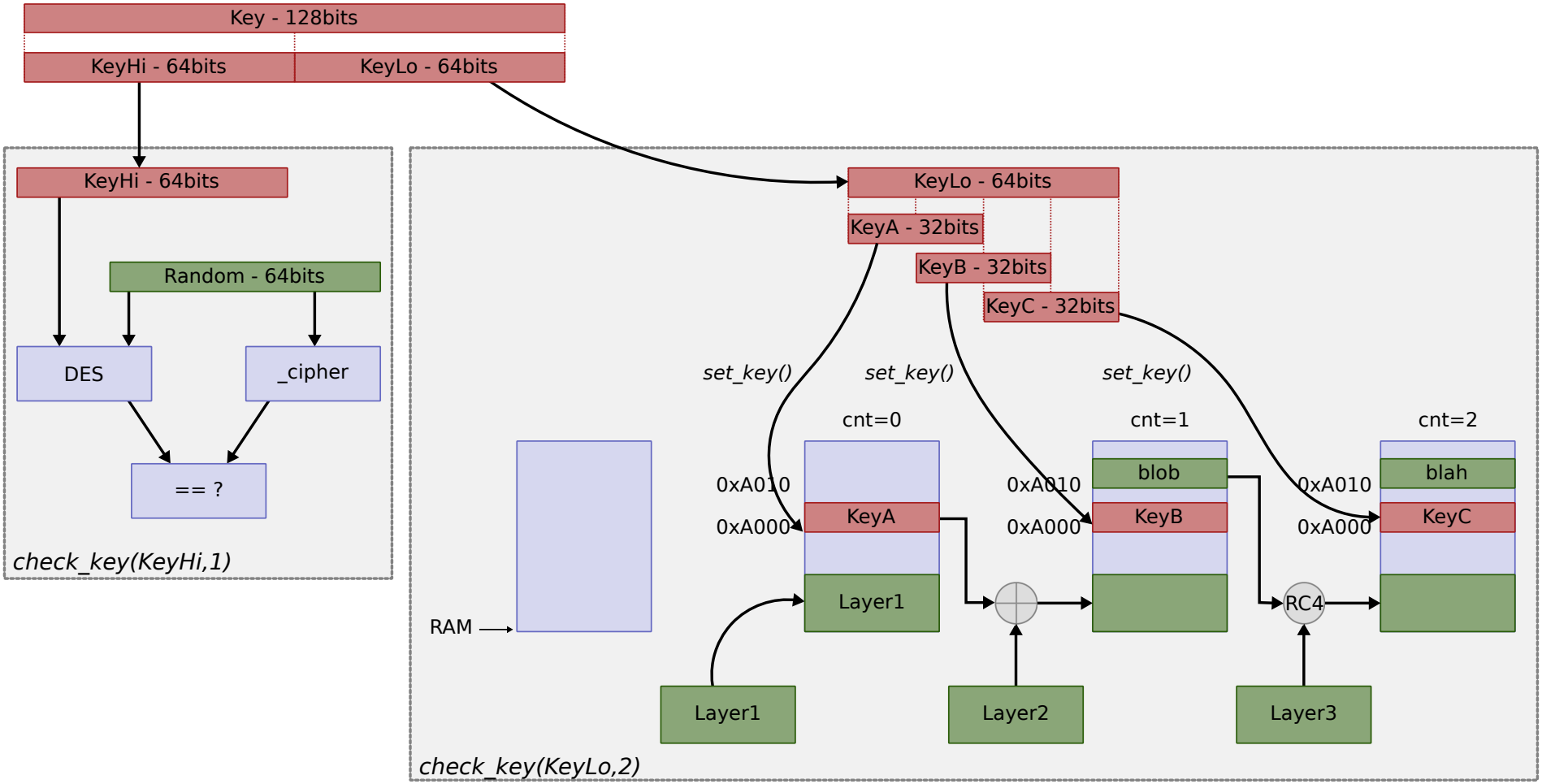

FIGURE 4 – Algorithme de vérification de la clé de déchiffrement dans *ssticrypt*

représentant la clé à tester. Cette clé est codée sur 16 caractères en notation hexadécimale. Elle peut donc être codée sur 8 octets. Une exécution du programme avec une clé choisie aléatoirement dure environ un dixième de seconde. Or, il y a $2^{8\times8}$ valeurs possibles. Pour balayer l'espace des clés, il faudrait donc $2^{(8*8)}/10/60/60/24/365 = 58494241736$ années sur un portable. Une analyse du code du programme est donc inévitable pour identifier un éventuel biais permettant de trouver une partie de la clé. Nous espérons que ce programme, pour mener ces tests, ne lance pas un challenge en attendant un résultat précalculé. Sinon, il y a de fortes chances qu'aucune information sur la clé ne soit présente et nous serions donc sur une mauvaise piste.

```
Usage: python check.pyc <key>
   - key: a 64 bits hexlify-ed string
Example: python check.pyc 0123456789abcdef
```

Listing 15 – Exécution de *check.pyc*

Un fichier avec l'extension *pyc* correspond à du *bytecode python* qui résulte de la compilation d'un fichier *py*. Un fichier *pyc* contient trois sections : a) quatre octets pour un nombre magique, b) quatre octets pour la date de modification et c) le code. La récupération de ces informations a été réalisée en utilisant le script *python* du listing 16 et le résultat de l'exécution est présenté dans le listing 17. Le nombre magique 62131 correspond à la version *2.5c2* de *python*. L'utilisation d'un interpréteur correspondant à cette version simplifie l'analyse de *check.pyc*.

```
import struct
import time
f = open("check.pyc", "rb")
magic = f.read(4)
magic = sum([int(magic[i]) << (i * 8) for i in range(4)]) & 0xFFFF
moddate = f.read(4)
moddate = time.asctime(time.localtime(struct.unpack('I', moddate)[0]))
f.close()
print("date:  ", moddate)
print("magic: ", magic)
```

Listing 16 – Programme de récupération des informations sur *check.pyc*

```
sh-3.2$ python info.py
date:   Thu Mar  8 09:38:21 2012
magic:  62131
```

Listing 17 – Récupération des informations sur *check.pyc*

Langage

N'étant pas encore outillé pour décompiler du *bytecode python*, nous avons développé notre propre décompilateur. Le *bytecode python* correspond à un langage à pile, à l'instar du *bytecode java* et du *postscript*. Nous avons tout d'abord restructuré les différentes fonctions en structures de contrôle. Pour ce faire, nous avons localisé les emplacements dans le code des sauts et les destinations de ces sauts afin de construire des blocs. Etant donné le nombre limité de structures de contrôle différentes dans le langage *python*, il suffisait ensuite d'identifier le type de saut réalisé (en avant ou en arrière, par exemple), pour retrouver la structure de contrôle employée. Les structures de contrôle permettent de découper le code des fonctions en blocs. Les blocs sont ensuite décompilés en tenant compte de l'enchaînement imposé par les structures de contrôle. Pour poursuivre la décompilation, il suffit de réaliser une exécution symbolique du programme, en stockant dans une pile, non pas les résultats des opérations réalisées mais directement les expressions à exécuter. Plusieurs cas de figures sont possibles :

1. Lorsqu'un élément de la pile est retiré pour être stocké dans une variable, un affichage est réalisé en utilisant l'expression associée à l'élément dépilé et le nom de la variable affectée.
2. Lorsque plusieurs éléments de la pile sont retirés pour être passés en paramètre d'une fonction, un affichage est réalisé en utilisant les expressions associées aux éléments dépilés et le nom de la fonction.
3. Lors de l'empilement d'une variable nous empilons le nom de la variable au lieu de la valeur associée.
4. Lors de l'empilement du résultat d'une opération, nous empilons une chaîne de caractères qui représente l'opération réalisée.

Le code du décompilateur est disponible en annexe D. Ce décompilateur souffre encore de plusieurs défauts (par exemple, il ne traite pas les exceptions) mais il permet tout de même de se faire une bonne idée de la logique du programme. Le résultat de la décompilation du programme est présenté en annexe E, en tronquant les lignes trop longues. Notons que le décompilateur est capable de récupérer les chaînes de caractères utilisées par le paquet *pickle* afin d'en analyser le contenu.

5.3 Analyse du fonctionnement de *check.pyc*

Rétro-ingénierie

Nous reprenons dans le listing 18, le code principal du programme en annexe D. Ce code vérifie tout d'abord la clé fournie en entrée. Si le format de cette clé n'est pas valide, le message d'erreur identifié précédemment est affiché. De plus, certains bits de la clé doivent être positionnés à des valeurs fixées. Plus précisément, une clé k doit satisfaire la condition : $k \,\&\, 0x8888 = 0x175$. Par conséquent, il nous reste *seulement* $64-4=60$ bits à trouver sur les 8 octets. Notons que la clé est chargée dans un objet de la classe *Bits* pour faciliter l'accès aux différents bits d'un nombre. Ensuite, un nombre aléatoire M est généré. Il est utilisé comme *challenge* : il est traité par deux méthodes différentes et la clé fournie par l'utilisateur est considérée correcte si les résultats des deux traitements sont identiques. Étant donné que M est généré aléatoirement, il y a de fortes chances que les tests ne reposent pas sur des *challenges* pré-calculés. Les informations sur les clés doivent se cacher dans *check.pyc*.

```
...
if __name__ == '__main__':
  __name__ == '__main__' # Pop from stack
  WT = pickle.loads(ccopy_reg\n_reconstructor\np0\n(ccheck\nWhiteDES\np1\nc__buil
  if len(sys.argv) == 1:
    len(sys.argv) == 1 # Pop from stack
    print('Usage: python check.pyc <key>')
    print('\n')
    print('   - key: a 64 bits hexlify-ed string')
    print('\n')
    print('Example: python check.pyc 0123456789abcdef')
    print('\n')
    return None
  len(sys.argv) == 1 # Pop from stack
  K = Bits(a2b_hex(sys.argv[1]), 64)
  if not K[range(7, 64, 8)] == 175:
    K[range(7, 64, 8)] == 175 # Pop from stack
    raise 1
  K[range(7, 64, 8)] == 175 # Pop from stack
  M = Bits(random.getrandbits(64), 64)
  if hex(WT._cipher(M, 1)) == hex(enc(K, M)):
    hex(WT._cipher(M, 1)) == hex(enc(K, M)) # Pop from stack
    exit(0) # Pop from stack
    return None
  hex(WT._cipher(M, 1)) == hex(enc(K, M)) # Pop from stack
  exit(1) # Pop from stack
__name__ == '__main__' # Pop from stack
```

Listing 18 – Code principal décompilé de *check.pyc*

Le premier traitement (`hex(enc(K,M))`) manipule à la fois le nombre aléatoire et la clé fournie par l'utilisateur. Le code associé est présenté sur le listing 19. Le code de la fonction F, invoquée par la fonction enc, a également été ajouté à ce listing. Après avoir vérifié la taille de la clé, la fonction enc poursuit par une permutation des bits de la clé. Le message est ensuite découpé en deux parties L et R. Ces deux parties vont subir 16 transformations à la manière d'un réseau de Feistel. Ces traitements nous indiquent qu'il s'agit de l'algorithme de chiffrement *DES*[2]. En particulier, la fonction F se charge à la fois d'étendre le bloc sur 48 bits (fonction $subkey$), de faire un $\oplus$ entre ce résultat et une partie de la clé sur 48 bits, et de transformer ces valeurs via les $Sbox$, en procédant par morceaux de 6 bits. Cette partie de *check.pyc* ne nous semble pas souffrir d'un biais.

```
...
# F <type 'function'>
def F(R,k,r):
  RE = E(R)
  Z = Bits(0, 32)
  fk = subkey(k, r)
  s = RE ^ fk
  ri = ['(0, 0)'][1]
```

```
  ro = ['(0, 0)'][0]
  for n in range(8):
    nri = (ri+6)
    nro = (ro+4)
    x = s[ri:nri]
    i = x[(5, 0)].ival
    j = x[(4, 3, 2, 1)].ival
    Z[ro:nro] = Bits(S(n, ((i<<4)+j)), 4)[range(None,None,-1)]
    ri = nri
    ro = nro
    continue
  return P(Z)
...
# enc <type 'function'>
def enc(K,M):
  if not M.size == 64:
    M.size == 64 # Pop from stack
    raise 1
  M.size == 64 # Pop from stack
  k = PC1(K)
  blk = IP(M)
  L = blk[0:32]
  R = blk[32:64]
  for r in range(16):
    fout = F(R, k, r)
    L = L ^ fout
    tmp = L
    L = R
    R = tmp
    continue
  tmp = L
  L = R
  R = tmp
  C = Bits(0, 64)
  C[0:32] = L
  C[32:64] = R
  return IPinv(C)
...
```

Listing 19 – Code de la fonction *enc* de *check.pyc*

Le second traitement (`hex(WT._cipher(M, 1))`) manipule uniquement le nombre aléatoire. Etant donné que le résultat est comparé au résultat du traitement précédant, ce traitement doit donc également procéder à un chiffrement par la méthode *DES*. Or, étant donné qu'aucune valeur correspondant à une clé n'est fournie en paramètre [4], la clé peut éventuellement être cachée dans l'implémentation de ce traitement. Le listing 20 présente l'implémentation de ce traitement. Une première lecture de cette fonction nous indique que la clé n'est pas récupérée par ailleurs. Par contre, nous retrouvons les 16 itérations du chiffrement par la méthode *DES*. Un premier point de divergence par rapport à une implémentation standard du *DES* concerne la permutation initiale. La table utilisée $tM1$ est différente de celle indiquée dans le standard. De plus, elle étend le message à chiffrer sur 96 bits. Certains bits peuvent donc potentiellement être dupliqués. Par conséquent, les tables de permutation utilisées dans la suite travaille sur 96 bits découpés en 12 morceaux de 8 bits. Les blocs entre les itérations ont donc une taille de 96 bits. Dans ces blocs, ont doit retrouver les deux blocs L_i et R_i de l'implémentation standard du *DES*, potentiellement mélangés. Notons qu'il y a 32 bits dont nous ne connaissons pas encore l'utilité. Pour finir, la clé doit se trouver dans la table KT. Cette table doit correspondre à la fonction F du *DES* avec la clé entrelacée. La figure 5 présente le fonctionnement de la fonction $\_cipher$. Si nous arrivons à exécuter pas à pas ce traitement en comparant avec les résultats attendus par une implémentation standard, nous pourrons séparer la clé. Ce travail fait l'objet de la section suivante.

```
...
# _cipher <type 'instancemethod'>
def _cipher(self,M,d):
  if not M.size == 64:
    M.size == 64 # Pop from stack
    raise 1
  M.size == 64 # Pop from stack
  if d == 1:
    d == 1 # Pop from stack
```

4. Notons que le paramètre 1 permet uniquement d'indiquer le type de traitement à réaliser : chiffrement ou déchiffrement

```
    blk = M[self.tM1]
    for r in range(16):
      t = 0
      for n in range(12):
        nt = (t+8)
        blk[t:nt] = self.KT[r][n][blk[t:nt].ival]
        t = nt
        continue
      blk = self.FX(blk)
      continue
    return blk[self.tM3]
    if d == -1:
      d == -1 # Pop from stack
      raise 1
      return None
    d == -1 # Pop from stack
  d == 1 # Pop from stack
...
```

Listing 20 – Code de la fonction *_cypher* de *check.pyc*

5.4 Cryptanalyse de *check.pyc*

Crypto.

Afin de retrouver la clé qui est cachée dans les différentes tables, nous avons analysé les différentes tables utilisées ($tM1$, $tM2$, $tM3$, etc.). Ces tables sont brouillées par rapport au standard. C'est à dire que, soit les ordres des bits après les permutations sont différents, soit des informations sont dupliquées, soit une partie de la clé est entrelacée dans les tables. Connaître les relations qui les lient ces tables au standard permet jusqu'à une itération de l'algorithme de comparer le résultat de l'itération suivante au résultat escompté, afin d'en extirper la clé. Progressivement, il est possible de reconstituer l'intégralité de la clé.

Le premier traitement consiste à déterminer, dans blk_i, la position des différents bits de L_i. Notons que ces positions sont indépendantes de i, sinon $tM2$ ne serait pas le même en fonction de l'itération i. Il nous suffit donc d'identifier la position des bits de L_0. Pour mener cette identification, nous avons adopté une approche systématique. Nous avons considéré un nombre L_0R_0 du standard DES avec seulement un des bits de L_0 positionné. Nous avons appliqué la transformation $IPInv$ sur ce nombre, suivie de la transformation $tM1$. Le seul bit positionné dans le résultat correspond au bit positionné de L_0. Le résultat de ce traitement nous permet d'obtenir la fonction $blk2li$. Notons que pour chaque bit testé, nous avons réalisé une vérification sur le résultat afin de nous assurer qu'il n'y avait qu'un seul bit positionné. Cette vérification est nécessaire car, à ce niveau, nous ne savons pas *pourquoi* blk_i est sur 96 bits et non 64. Le code correspondant à ce traitement est présenté sur le listing 21.

```
# Identification des positions des bits de Li dans blki.
blk2li = [0] * 32
zero64 = [0] * 64
for i in range(32):
  zero64[i] = 1
  input = [zero64[j] for j in IPinv]
  blk = [input[j] for j in tM1]
  bits_set_in_blk = [j for j in range(len(blk)) if blk[j] == 1]
  blk2li[i] = bits_set_in_blk[0]
  zero64[i] = 0
```

Listing 21 – Identification des bits de L_i

Le traitement précédent a été appliqué sur les bits de R_0. Un bit positionné dans R_0 conduit à *l'apparition* de deux bits dans blk_0. Les 32 bits supplémentaires dans blk_i par rapport au standard s'expliquent donc par une redondance de R_0 dans blk_0. Ce résultat est valable pour le début de la première itération pour l'instant. Au final, ce traitement n'est pas adapté pour l'identification des bits de R_0. Nous avons donc adopté une autre stratégie qui nous amène au second traitement.

Le second traitement doit donc permettre d'identifier les bits de R_i dans blk_i. Nous nous concentrons sur R_0. Pour ce faire, nous nous appuyons sur le fait que dans le standard DES, $L_{i+1} = R_i$, donc $L_1 = R_0$. Bien que nous connaissons déjà les positions des bits de L_1 dans blk_1, l'identification n'est pas aussi facile que pour le traitement précédent car le passage de blk_0 à blk_1 ne correspond pas à un simple changement d'ordre des bits ou une duplication. Effectivement, ce passage concerne également la table $tM2$, via FX, qui fait indirectement intervenir la clé pour *brouiller* blk_0. Ce qui est assuré est la recopie intacte de R_0

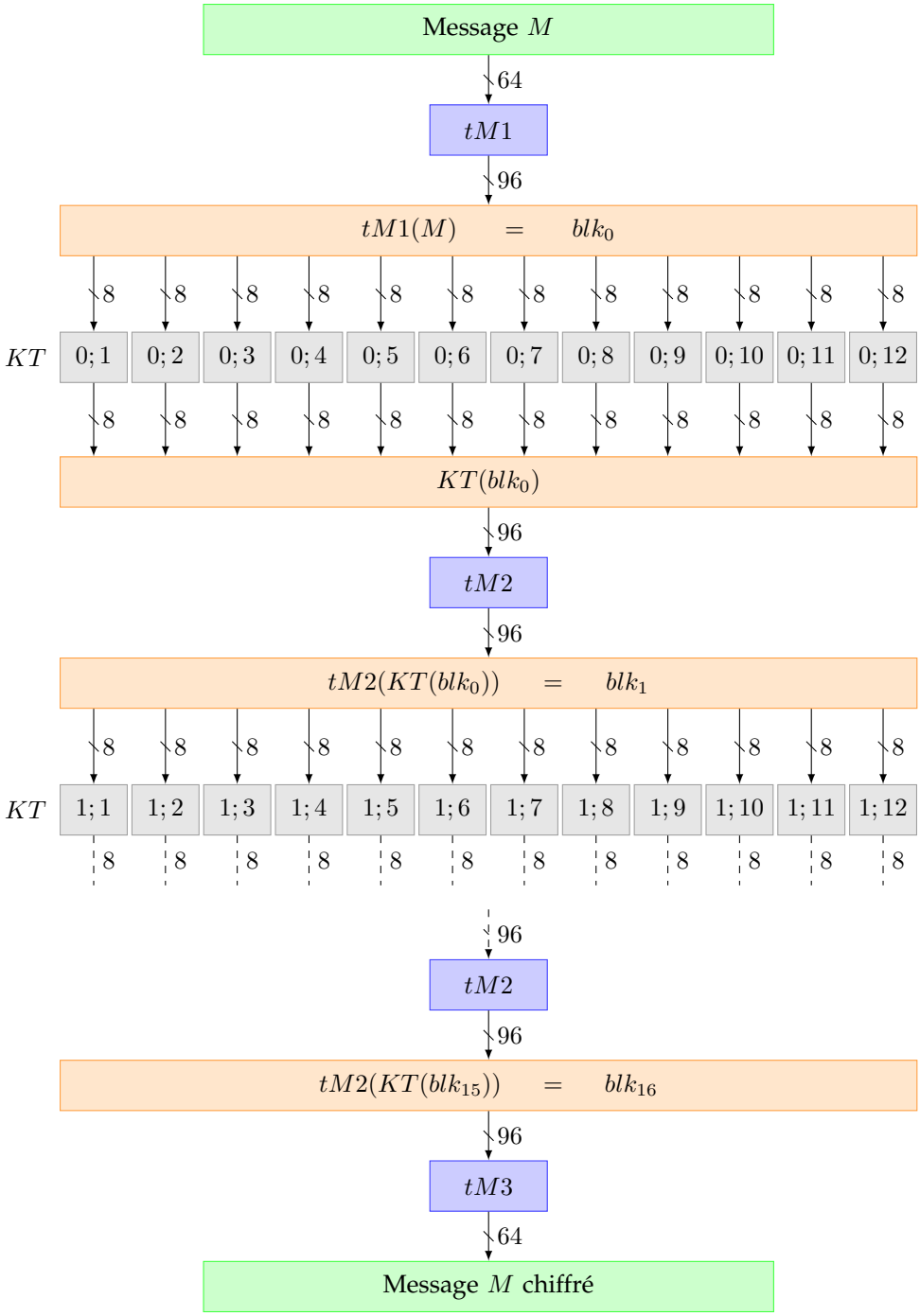

FIGURE 5 – Fonction *_cipher*

dans blk_1, même si l'intégrité de la donnée blk_0 n'est pas assurée (c'est-à-dire que les 32 bits ajoutés par rapport au standard ne sont pas intègres). Pour l'instant, chacun des 64 bits de blk_0 (autres que ceux correspondant à L_0) sont candidats pour être associés à R_0, donc L_1. Donc, pour obtenir la position du bit j de R_0, il suffit de générer un nombre aléatoire noté $ablk_0$, de le transformer avec KT et FX en $ablk_1$ et, pour le bit j de la partie de $ablk_1$ correspondant L_1, identifier ceux de $ablk_0$ qui ont la même valeur. Les autres ne sont plus des candidats. Ce test doit être répéter jusqu'à ce qu'il ne reste plus qu'un candidat pour le bit j et il doit être répété pour tous les bits de R_0. Ce traitement nous permet d'obtenir $blk2ri$. Dans l'implémentation de ce traitement, présenté à la figure 22, chaque nombre aléatoire généré est utilisé pour tester tous les bits de R_0 et nous répétons le traitement 400 fois, ce qui est suffisant pour obtenir un candidat par bit de R_0. Ce point pourrait largement être simplifié.

```
# Identification des positions des bits de Ri dans blki.
candidates = dict([(i, list(range(96))) for i in range(96)])
for i in range(400):
  ablk0 = laleatoire(96)
  ablk1 = ablk0[:]
  t = 0L
  for n in range(12):
      nt = t + 8
      l = KT[0][n]
      ablk1[t:nt] = intToTab(l[tabToInt(ablk1[t:nt])], nt - t)
      t = nt
  ablk1 = FX(ablk1)
  # Dans ces boucles, si le j-eme bit de ablk1 n'est pas egale
  # au k-eme bit de ablk0, alors k n'est pas un candidat pour
  # la generation du j-eme bit de ablk1.
  for j in range(96):
    for k in candidates[j][:]:
      if ablk0[k] != ablk1[j] and k in candidates[j]:
        candidates[j].remove(k)
blk2ri = [0] * 32
for i in range(len(blk2li)):
  blk2ri[i] = candidates[blk2li[i]][0]
```

Listing 22 – Identification des bits de R_i

A présent, pour poursuivre, il est nécessaire de comprendre mieux le fonctionnement de $tM2$ et l'intérêt de manipuler des nombres sur 96 bits. Certaines valeurs de $tM2$ contiennent un seul bit positionné à 1 et les autres en contiennent 2. Ces valeurs correspondent à des masques qui sont appliqués à blk_i et les nombres de bits positionnés à 1 sont comptés dans les résultats, modulo 2. Ces nombres constituent blk_{i+1}. Dans le standard, la dernière opération réalisée pour obtenir R_{i+1} est le $\oplus$ entre L_i et $f(K_i, R_i)$. Pour obtenir L_{i+1}, il s'agit d'une recopie de R_i. Donc, un masque de $tM2$ contenant un seul bit positionné à 1 correspond à la recopie de R_i tandis qu'un masque contenant deux bits positionnés à 1 correspond au $\oplus$ de L_i et $f(K_i, R_i)$. Notons que les auteurs ont programmé un outil permettant de récupérer la position de ces bits. Or cet outil n'est pas utilisé dans la suite.

A ce niveau, nous connaissons les positions des bits de R_i et L_i dans blk_i. Or, R_{i+1} est obtenu en effectuant un $\oplus$ entre $f(K_i, R_i)$ et L_i. Donc, en effectuant un nouveau $\oplus$ entre R_{i+1} et L_i, nous retrouvons $f(K_i, R_i)$. Dans le standard, à l'itération i, R_i n'est pas fonction de K_i, contrairement à $f(K_i, R_i)$. La clé doit donc être entrelacée dans les $Sbox$, c'est-à-dire, pour l'implémentation du challenge, dans KT. L'obtention de la clé, au final, se fait donc en analysant les possibilités pour, depuis R_i, obtenir $f(K_i, R_i)$, Ceci est possible en procédant $Sbox$ par $Sbox$ et itération par itération. Lorsque toutes les sous-clés utilisées par les $Sbox$ d'une itération ont été trouvées, nous pouvons passer à l'itération suivante. Aussi, par simplicité, il suffit de tester toutes les combinaisons pour chaque $Sbox$, soit 2^6. Les fonctions obtenues dans les traitements précédents nous permettent de traduire tous les résultats intermédiaires du *challenge* en résultats intermédiaires du standard *DES*. Elles sont utiles pour ce traitement visant à retrouver la clé. Pour chacun de ces $Sbox$, il y a 64 candidats. Pour obtenir la clé correspondant à une $Sbox$, il suffit d'éliminer les mauvais candidats. Pour ce faire, nous générons un nombre aléatoire que nous chiffrons jusqu'au $Sbox$ étudié à la fois avec l'implémentation du challenge et l'implémentation standard et un candidat, en utilisant les clés trouvées pour les premières itérations. Si les bits que le $Sbox$ doit engendrer diffèrent entre le résultat avec l'implémentation standard et le candidat et le résultat avec l'algorithme du challenge, alors le candidats est mauvais et il est retiré. Nous réitérons jusqu'à ce qu'il n'y ait plus qu'un candidat par $Sbox$, soit 8 candidats pour l'itération. A ce moment, nous pouvons poursuivre à l'itération suivante. Ce traitement est présenté sur le listing 23. La figure 6 synthétise la démarche pour le test d'un candidat d'un $Sbox$.

```
# Recuperation des indices permettant d'obtenir f(Ri,Ki+1) depuis tM2.
blk2pfi = [0] * 32
for j in range(len(blk2ri)):
  i = blk2ri[j]
  # A ce niveau, bitLenCount(tM2[i]) == 2, sinon c'est pas un bit de Ri.
  cont = True
  pos = bitPos(tM2[i])
  while cont:
    input = laleatoire(64)
    blk = [input[n] for n in tM1]
    l0 = [blk[n] for n in blk2li]
    # Iteration de l'etage 1.
    r = 0
    t = 0L
    for n in range(12):
      nt = t + 8
      l = KT[0][n]
      blk[t:nt] = intToTab(l[tabToInt(blk[t:nt])], nt - t)
      t = nt
    cont = blk[pos[0]] == blk[pos[1]]
  if blk[pos[0]] == l0[j]:
    blk2pfi[j] = pos[1]
  else:
    blk2pfi[j] = pos[0]
# Recuperation des clefs intermediaires.
Kq = [None] * 16
for q in range(16):
  # 48 premiers bits de Kq
  candidatesKq = dict([(i, map(list, list(product(range(2), repeat=6)))) for i in range(8)])
  #candidatesKqb1a6 = map(list, list(product(range(2), repeat=6)))
  last = sum([len(candidatesKq[i]) for i in range(8)])
  while last > 8:
    # Pour un nombre, recuperation de L0, R0, L1 et R1.
    # Il faut que les Li et Ri soient coherents, on part donc de input.
    input = laleatoire(64)
    blk0 = [input[i] for i in tM1]
    # Iterations des etages 1 a q.
    for x in range(q + 1):
      blk1 = blk0[:]
      t = 0L
      for n in range(12):
        nt = t + 8
        l = KT[x][n]
        blk1[t:nt] = intToTab(l[tabToInt(blk1[t:nt])], nt-t)
        t = nt
      blk1_before_xor = blk1[:]
      blk1 = FX(blk1)
      if x < q:
        blk0 = blk1
    l0 = [blk0[i] for i in blk2li]
    r0 = [blk0[i] for i in blk2ri]
    l1 = [blk1[i] for i in blk2li]
    r1 = [blk1[i] for i in blk2ri]
    fr0k1 = [int(r1[i] ^ l0[i]) for i in range(32)]
    # Calcul de f(R,K) pour chacun des candidats et comparaison avec fr0k1.
    r0e = [r0[i] for i in E]
    for ci in candidatesKq:
      ii = ci * 6
      io = ci * 4
      for k in candidatesKq[ci][:]:
        r0exk = [int(r0e[ii + i] ^ k[i]) for i in range(6)]
        m = (r0exk[0] << 1) + r0exk[5]
        n = (r0exk[1] << 3) + (r0exk[2] << 2) + (r0exk[3] << 1) + r0exk[4]
        r0exks = S[ci][(m << 4) + n]
        r0exks = intToTab2(r0exks, 4)
        r0exkss = ([2] * (io)) + r0exks + ([2] * (32 - 4 - io))
        r0exkssp = [r0exkss[i] for i in P]
        for i in range(32):
          # Pour le bit i, si dans r0exkssp la valeur est 2, c'est que ce bloc
          # n'engendre pas ce bit. On ignore ce test pour l'instant.
          if r0exkssp[i] != 2 and r0exkssp[i] != fr0k1[i]:
            candidatesKq[ci].remove(k)
            last = last - 1
            break
```

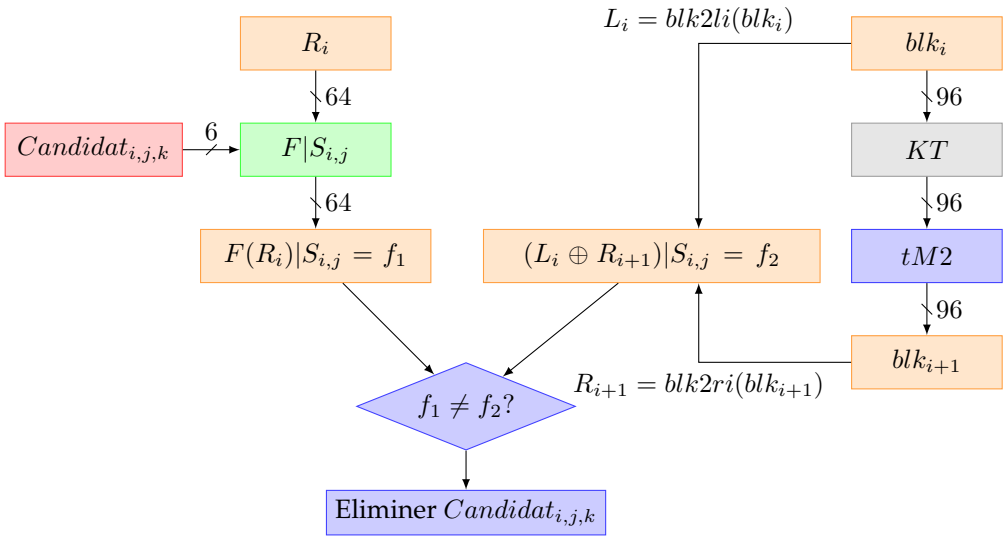

FIGURE 6 – Test des candidats pour un $Sbox$

```
  Kqq = []
  for i in range(8):
    Kqq = Kqq + candidatesKq[i][0]
  Kq[q] = Kqq

f = open("out.Kqq", "wb")
import pickle
pickle.dump(Kq, f)
f.close()
```

Listing 23 – Identification des K_i

Les sous-clés obtenues K_i ne correspondent pas exactement à la clé utilisée. Il faut inverser le traitement réalisé dans le standard, par la fonction $subkey$. Le seul point particulier concerne les bits de parité : ces derniers sont imposés dans le code principal de *check.pyc* ($k \ \& \ 0x8888 = 0x175$). Ce traitement est présenté en annexe G.

Le résultat de l'exécution est fournie dans le listing 24. Le temps d'exécution est correct et la clé identifiée est `fd4185ff66a94afd`. Notons qu'une exécution de *check.pyc* nous confirme que cette clé est valide.

```
sh-3.2$ time python2.5 reverse.py
real    0m2.084s
user    0m2.056s
sys 0m0.018s
sh-3.2$ time python2.5 subkey2key.py
fd4185ff66a94afd
real    0m0.032s
user    0m0.014s
sys 0m0.011s
sh-3.2$ python2.5 check.pyc
Usage: python check.pyc <key>
   - key: a 64 bits hexlify-ed string
Example: python check.pyc 0123456789abcdef
sh-3.2$ python2.5 check.pyc fd4185ff66a94afd
sh-3.2$ echo $?
0
sh-3.2$ python2.5 check.pyc 1111111111111111
Traceback (most recent call last):
  File "check.py", line 50107, in <module>
AssertionError
```

Listing 24 – Recherche et vérification de la clé pour le *DES*

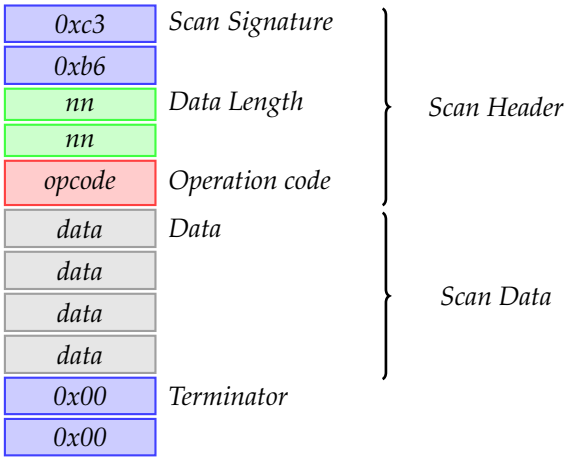

FIGURE 7 – Représentation simplifiée du format SCAN

6 Vérification des 64 bits de poids faible de la clé

Cette section est dédiée à l'analyse du processus de vérification des 64 bits de poids faible de la clé fournie en paramètre à *ssticrypt*. Ce dernier repose sur un blob binaire chargé dans un périphérique USB, une webcam USRobotics pour être plus précis. Nous détaillons dans la suite de cette section la manière avec laquelle nous avons appréhendé ce blob binaire.

6.1 Identification du processeur utilisé par le périphérique USB

Google kung-foo

Bien que le périphérique USB utilisé pour cette épreuve corresponde à une webcam, il parait difficile d'imaginer qu'il ne s'agisse que de simples données. Nous pensons alors à un *firmware* probablement exécuté par un processeur embarqué dans la webcam. L'analyse des sources du noyau Linux associé à ce périphérique USB nous confirme notre intuition à propos du *firmware*. Il ne nous reste plus donc qu'à identifier le processeur utilisé puis analyser le code chargé.

Identifier le processeur embarqué dans la webcam n'a pas été simple. Les nombreuses recherches sur Internet en utilisant les mots-clés « USRobotics », « *processor* » et « *firmware* » n'ont pas donné de résultats intéressants. Face à la difficulté, nous avons essayé de glaner des informations que d'autres participants auraient laissés sur Internet. Cette approche s'est avérée très fructueuse, et nous avons découvert au travers d'une discussion sur le réseau social *Twitter* que le périphérique USB utilise un processeur Cypress *cy16* [16]. La consultation du dépôt du projet *metasm* [4] vient consolider cette information : la définition du processeur Cypress *cy16* a été ajouté peu après que la première réponse au concours a été soumise.

6.2 Format des blobs binaires et extraction du code

À présent que nous avons identifié le processeur utilisé par le périphérique USB, nous avons essayé de comprendre la procédure avec laquelle le *firmware* est chargé dans le périphérique USB. Le document [10] décrit en détail cette procédure et précise le format des blobs binaires chargés.

Les blobs binaires sont stockés au format SCAN. Il s'agit d'un format qui permet de charger du code exécutable dans les périphériques USB disposant d'un processeur Cypress *cy16* et de modifier leur comportement par défaut. Concrètement, le programme embarqué par défaut dans le périphérique USB recherche continuellement la signature `0xc3b6`. Cette signature est suivie de la longeur des données utiles. L'octet qui suit précise ensuite les operations à appliquer aux octets contenus dans le blob binaire. Une représentation simplifiée du format SCAN est rappellé en figure 7.

Pour extraire le code exécuté par le processeur Cypress *cy16* dans le périphérique USB, nous avons implémenté un interpréteur de fichier au format SCAN. L'interpréteur développé est donné en annexe H. Il convient de noter qu'il n'est pas complet. Les fonctionnalités implémentées suffisent amplement pour extraire le code exécuté par le processeur. Le listing 25 présente un extrait du *firmware* contenu dans les blobs binaires *init_rom* et *stage2_rom*. Le code a été desassemblé en utilisant *metasm*.

```
sh-3.2$ python load_rom.py init_rom.fw init_rom.blob
sh-3.2$ python load_rom.py stage2_rom.fw stage2_rom.blob
sh-3.2$ ruby cy16dis.py init_rom.fw
```

```
init_rom:
    mov word ptr [0c008h], 0
    mov word ptr [0c03eh], 0
    and word ptr [0c00eh], 0fffdh
    mov word ptr [8eh], word ptr [0deh]
    mov r0, word ptr [340h]
    add r0, 344h
    int 4bh
    mov r0, 344h
    int 4bh
    mov r0, 0
    int 4bh
    ret
sh-3.2$ ruby cy16dis.py stage2_rom.fw
[...]
```

Listing 25 – Extraction du *firmware* contenu dans *init_rom* et *stage2_rom*

6.3 Machine virtuelle

Nous avons écrit un premier émulateur pour processeur *cy16*, en *python*. Cet émulateur ne prend pas en compte toutes les subtilités du processeur mais nous permet déjà de caractériser un peu plus le fonctionnement du programme. En particulier, il permet de tracer l'exécution, en collectant les valeurs successives que le pointeur d'instruction peut prendre. L'intérêt de cette approche est de rapidement identifier les routines les plus sollicitées dans le programme. En l'exécutant, nous avons identifié des octets ne correspondant pas à de véritables instructions pour ce processeur. Dans un premier temps, nous avons remplacé ces octets par des *nop*. Le résultat de l'exécution de l'émulateur à partir des *roms* et de *layer1* est représenté dans la figure 8. Le temps est représenté en abscisse et les valeurs du pointeur d'instruction sont représentées en ordonnée.

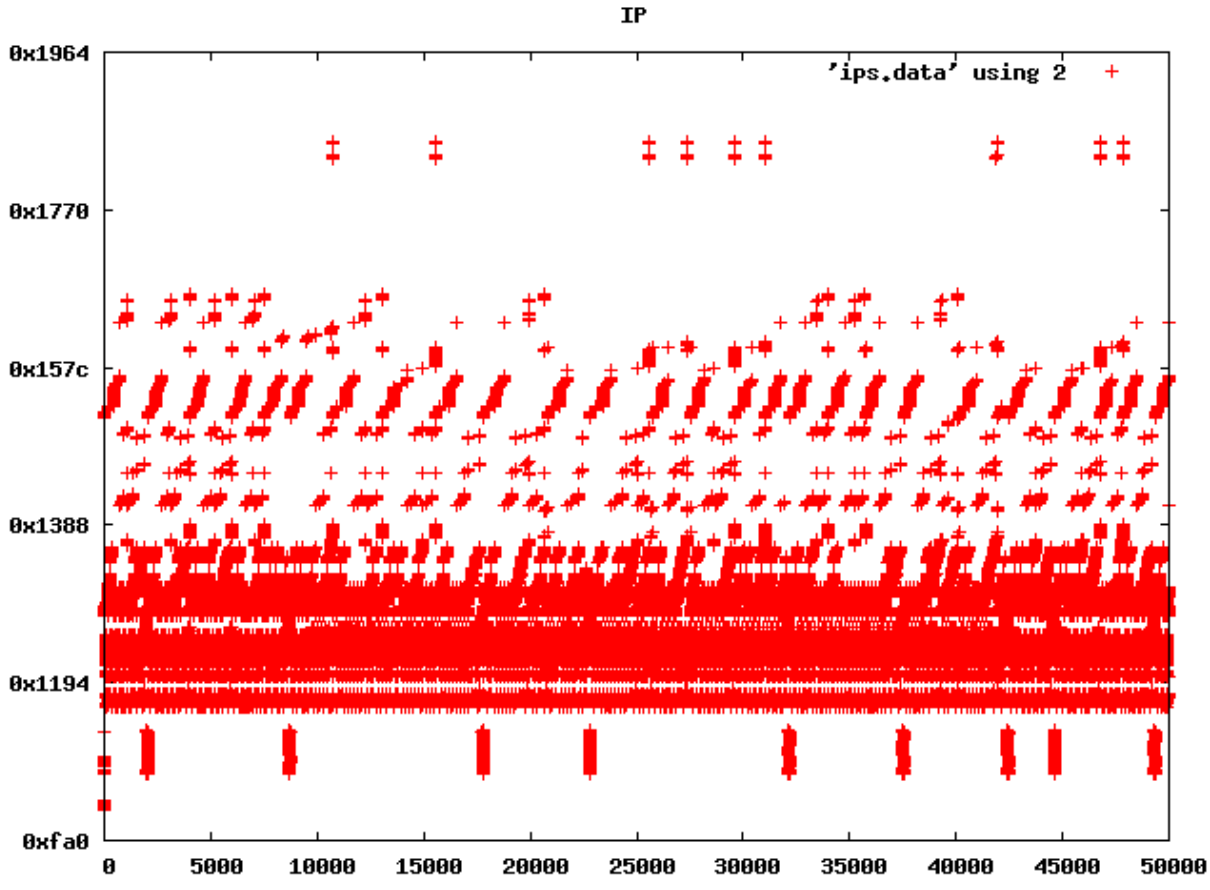

FIGURE 8 – Évolution du pointeur d'instruction lors d'une exécution

En analysant cette figure, nous pouvons effectivement identifier des routines exécutées plus souvent que les autres, par exemple, autour de l'adresse $0x1194$. Une lecture du code source assembleur permet de localiser plus précisément ces routines (pour l'adresse $0x1194$, il s'agit de $0x1162$, $0x1166$, $0x116a$ et $0x11da$). Une lecture plus approfondie du code assembleur est alors nécessaire pour comprendre le fonctionnement de plusieurs routines. Elles sont présentées dans la suite :

$0x1076$ Cette routine correspond au *handler* permettant de traiter les interruptions liées à la réception de données envoyées par le client par le port *USB*.

$0x11da$ Cette routine commence par incrémenter les 5 valeurs de la table associée à l'adresse $0x105e$. Ensuite, elle recherche dans la table associée à l'adresse $0x1054$ la valeur $r0\&0xfff0$. Si cette valeur est trouvée, son indice dans la table associée à l'adresse $0x1054$ est ajouté à $r0\&0xf$ et retourné à la routine appelante. Si aucune valeur n'est retournée, un saut brutal est réalisé à la fin du traitement du *handler* précédent, via la restauration du pointeur de pile sp depuis une sauvegarde à l'adresse $0x1070$ qui a été réalisée au début de la routine $0x14da$. Vraisemblablement, il s'agit d'un code correspondant au fonctionnement d'un gestionnaire de mémoire. L'entrée i de la table $0x1054$ de 5 entrées contient l'adresse virtuelle d'une page et elle est associée à l'adresse physique $0x10i0$. Notons que le bit de poids faible est utilisé pour indiquer si un accès en écriture a eu lieu sur chacune de ces pages. Les adresses physiques sont comprises entre $0x1000$ et $0x1050$. La table $0x105e$ contient le nombre d'accès aux pages depuis le dernier accès à chacune de ces pages. Cette information permet d'identifier une page à remplacer en cas de défaut de page et de table remplie. Plus globalement, le registre $r0$ contient une adresse "virtuelle" que cette routine transforme en adresse physique.

$0x1162$ Cette routine est utilisée pour permettre les accès en lecture ou en écriture à des octets ou des mots de 16 bits en s'appuyant sur la routine $0x11da$.

$0x1268$ Cette routine manipule deux valeurs importantes. La première est stockée à l'adresse $0x111c$. Elle pointe sur un octet en mémoire. Cette valeur est utilisée pour récupérer un octet en mémoire en utilisant la routine $0x1162$. La seconde valeur importante est stockée à l'adresse $0x111e$. Elle correspond au nombre de bits déjà consommés dans l'octet pointé par l'adresse stockée en $0x111c$. Au final, cette routine permet de lire les valeurs en mémoire par blocs de bits. Le nombre de bits à lire est founi par la fonction appelante via le registre $r10$. Il s'agit donc d'une routine permettant de faire un *fetch*. **A ce niveau, nous étions sur d'avoir à faire à une machine virtuelle.**

$0x13c4$ Cette routine récupère plusieurs séquences de bits et, en fonction des valeurs de ces bits, elle réalise différents types d'accès à la mémoire en mettant à jour la structure $0x1126$. Plus précisément, elle permet à la fois de calculer des adresses et de récupérer des valeurs stockées en mémoire à ces adresses. Il s'agit d'une routine permettant le décodage des opérandes d'une instruction.

$0x1346$ Cette routine permet de transférer le contenu de la structure à l'adresse $0x1126$ à l'adresse $0x1134$. Elle est utilisée dans le cas d'une instruction nécessitant plusieurs opérandes. Elle sauvegarde les valeurs obtenues dans la structure $0x1126$ lors du décodage du premier opérande (via la routine $0x13c4$) pour permettre ultérieurement le décodage du second opérande.

$0x14da$ Cette routine permet d'exécuter plusieurs instructions en s'appuyant sur les routines précédentes. L'analyse des structures *if* imbriquées permet d'obtenir le jeu d'instruction de cette machine virtuelle.

En comprenant le rôle de ces différentes routines, nous avons pu corriger correctement les erreurs identifiées en début de cette section : les octets ne correspondant pas à de véritables instructions pour le processeur. La structure de la machine virtuelle est représentée dans la figure 9. Dans cette figure, la flèche numérotée ❶ correspond à l'invocation du *fetch* depuis la routine principale $0x14da$. Ce *fetch* doit récupérer des bits associés à l'instruction en cours d'exécution (❷). Pour ce faire, il s'appuie sur les informations associées aux tables $0x1154$ et $0x115e$ (❸) pour récupérer les bits en mémoire (❹). Si besoin, la page contenant les bits associés à l'instruction est récupérée auprès du processeur *MIPS* (❺). Une partie du jeu d'instructions pour cette machine est représentée dans la figure 10.

6.4 Interpréteur pour la machine virtuelle

L'analyse à partir de l'émulateur du processeur *cy16* manque de souplesse. Effectivement, la moindre instruction de la machine virtuelle est traitée par plusieurs instructions du processeur *cy16*, elles-mêmes émulées avec plusieurs instructions *python*. Au final, l'émulateur nous aura permis de comprendre le fonctionnement de *stage2_rom* et d'en déduire la structure et le jeu d'instructions de la machine virtuelle. A partir de ce point, nous avons décidé de développer notre propre interpréteur pour la machine virtuelle. Cet interpréteur reconnait exactement les mêmes instructions que la machine virtuelle de *stage2_rom*. Le code est fourni en annexe I. L'exécution de l'interpréteur aboutit à un gain énorme. A titre

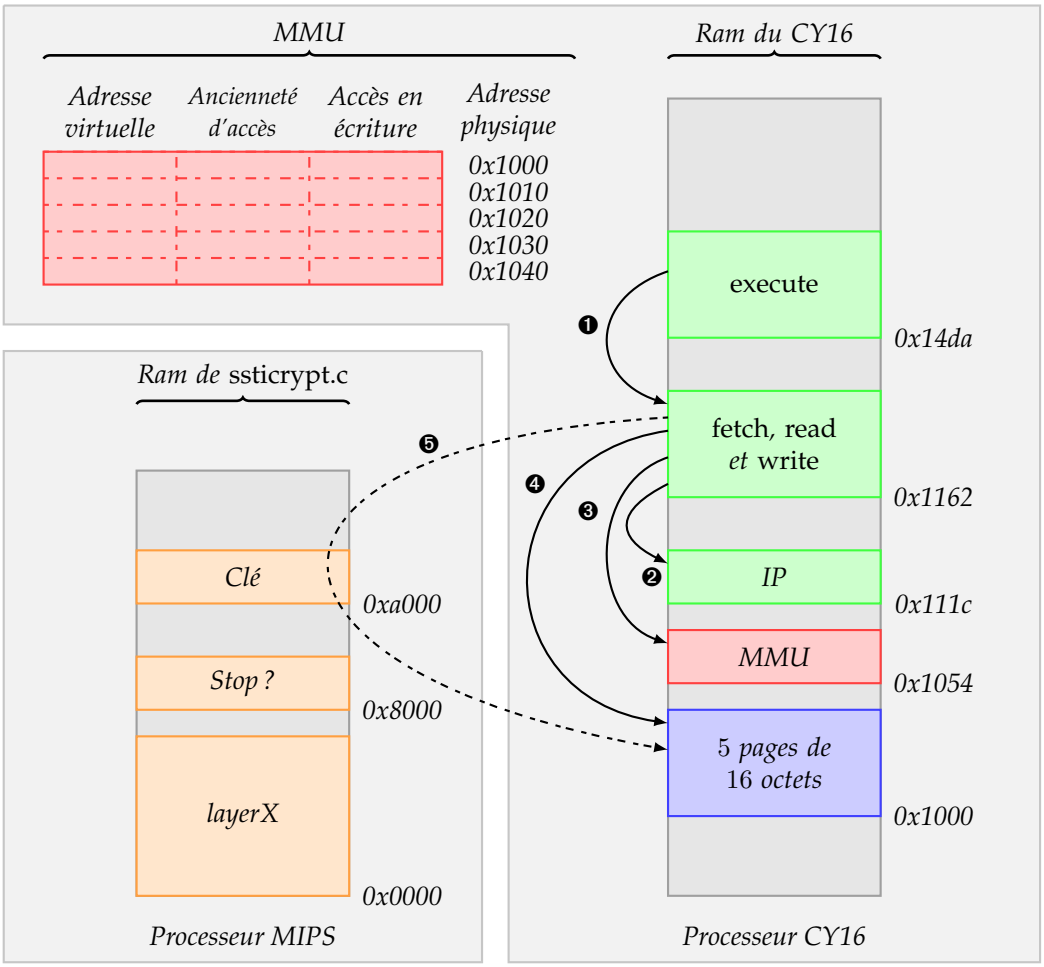

FIGURE 9 – Structure de la machine virtuelle

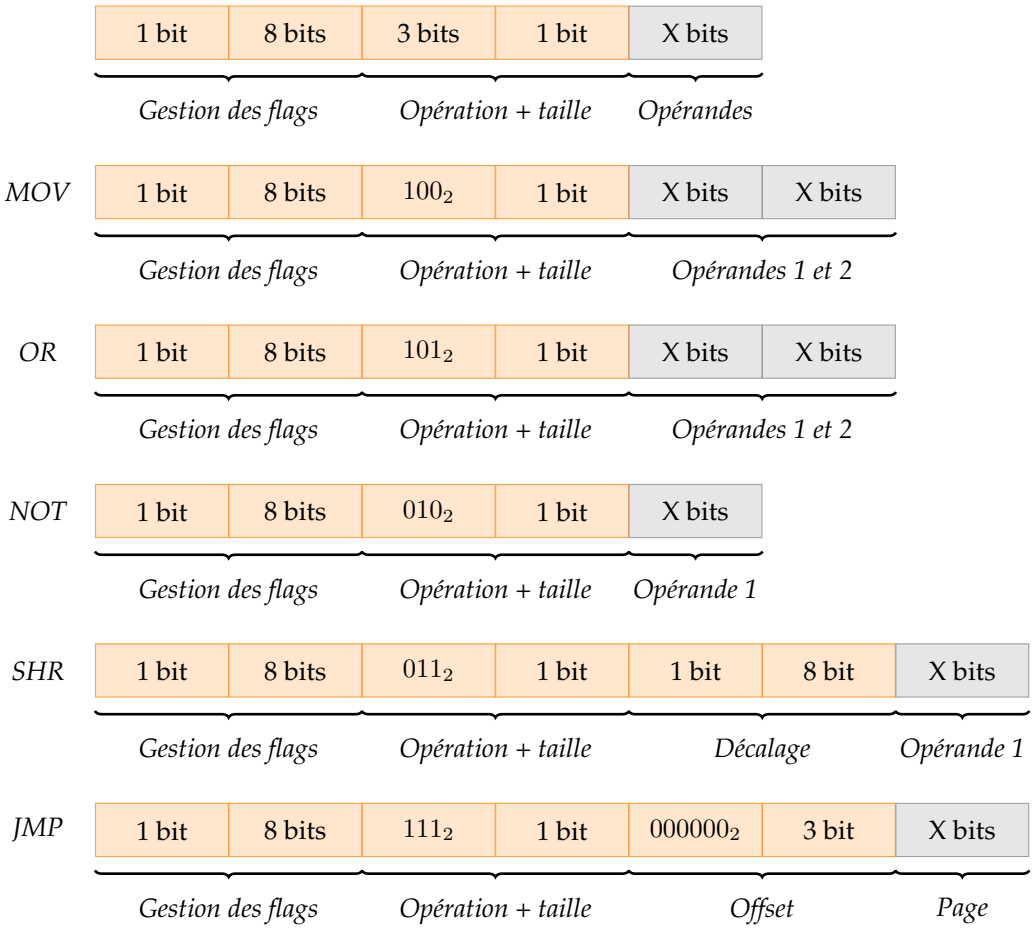

FIGURE 10 – Extrait du jeu d'instruction de la machine virtuelle

de comparaison, une exécution avec une clé à partir de l'émulateur peut durée environ 2 secondes contre seulement $0,01$ secondes sur l'interpréteur. Cette différence est normale, étant donné le niveau auquel on se situe pour réaliser notre traitement (au niveau *micro-instruction*, contre le niveau instruction de la machine virtuelle).

Cet interpréteur est dédié à l'exécution de *layer1*, *layer2* et *layer3*. Connaissant la structure de la machine virtuelle, nous aurions pu nous lancer directement dans le désassemblage et la décompilation des *layerX*. Plutôt que d'adopter cette approche, nous avons décidé de tracer l'exécution de ces programmes. Autrement dit, au lieu de désassembler tous les *layerX* en intégralité, nous avons choisi de désassembler uniquement les instructions réellement exécutées lors d'une exécution. Pour ce faire, nous avons enrichi notre propre interpréteur avec des affichages représentant les instructions en cours d'exécution. Sur la base de ces outils, nous avons commencé à chercher la clé.

6.5 Reconstruction de la clé

La première manipulation que nous avons réalisé a été de tester si, par force brute, nous étions capable d'obtenir la clé. Un premier test nous a indiqué qu'il aurait fallut 16 heures pour balayer l'espace des clés, en répartissant les calculs sur 16 cœurs, sur une machine équipée de quatre processeurs *AMD Opteron* double cœurs. Nous avons lancé en tâche de fond ce traitement tout en suivant une nouvelle piste. Il s'avère que la méthode que nous vous présentons dans la suite nous a permis d'obtenir rapidement la clé. Bien que nous ayons trouvé la clé, le traitement en tâche de fond a tout de même été maintenu pour vérifier la cohérence des résultats.

No skills

Le résultat du désassemblage à l'exécution avec la clé $0x12345678$ sur *layer1* est fourni dans le listing 26. Cette exécution commence par récupérer la clé à partir de l'adresse $0xa000$. Cette clé est traitée par plusieurs opérations. Un saut est ensuite réalisé à la ligne 25. Ce saut semble tester la valeur `data[d89e]`, or cette valeur ne dépend pas de la clé. Il s'agit donc d'un saut inconditionnel. Ensuite, nous devinons l'exécution de plusieurs itérations d'une boucle. Effectivement, les lignes 28 à 31 sont identiques aux lignes 34 à 37, aux lignes 40 à 47 et aux lignes 40 à 49. Chacune de ces séquences d'instructions se termine par un saut (aux lignes 32, 38, 44 et 50). A ce niveau, nous sommes persuadé que ces lignes concernent un saut conditionnel correspondant à un `do {...} while (...);`. A la fin de l'exécution de la boucle, de nouvelles opérations sont réalisées, sûrement sur la base du résultat du calcul effectué par la boucle. Un nouveau saut est alors réalisé et l'exécution s'arrête avec une erreur. Nous avons donc la structure algorithmique simplifiée présentée dans le listing 27.

```
1  rc <- data[a000]
2  r3 <- data[a002]
3  r0 <- rc >> 8
4  r6 <- ~((r0 & rc) | (~(r0|rc)))
5  r6 <- (~(r6 | (~9577))) | (r6 & (~9577))
6
7  r1 <- r3 >> 8
8  r8 <-   (r1 & ~r3) | ~(r1 | ~r3)
9  r1 <- rc & ff
10 r8 <-   (r8 &  r1) | ~(r8 |  r1)
11 r1 <- r1 << 8
12 r8 <-   (r8 &  r1) | ~(r8 |  r1)
13 r1 <- r0 << 8
14 r8 <- ~((r8 &  r1) | ~(r8 |  r1))
15 r8 <- (~(r8 & r0)) & (r8 | r0)
16
17 rd <- 1175
18 r5 <- ~((~1 | (~ae4d)) & (1 | ae4d))
19
20 data[c75e] <- r5
21 data[cf9a] <- (~(~ae4d))
22 data[d89e] <- (1 & (~ae4d)) << 1
23 data[f9f8] <- r8
24
25 jump to 191:1? (data[d89e] & data[d89e]) -> no
26 rd <- (~rd) << 1
27 r5 <- 16b8
28
29 rd <- ~(~data[f9f8] | data[c75e]) | (~data[f9f8] & data[c75e])
30 data[cb26] <- (data[f9f8] & data[c75e]) << 1
31 data[d4ac] <- rd
32 data[deb4] <- ~(~data[f9f8] | data[c75e]) | (~data[f9f8] & data[c75e])
33 jump to 20e:0? (data[cb26] & data[cb26]) -> yes
```

```
34
35 rd <- ~(~data[d4ac] | data[cb26]) | (~data[d4ac] & data[cb26])
36 data[cb26] <- (data[d4ac] & data[cb26]) << 1
37 data[d4ac] <- rd
38 data[deb4] <- (~(~data[d4ac] | data[cb26])) | (~data[d4ac] & data[cb26])
39 jump to 20e:0? (data[cb26] & data[cb26]) -> yes
40
41 rd <- ~(~data[d4ac] | data[cb26]) | (~data[d4ac] & data[cb26])
42 data[cb26] <- (data[d4ac] & data[cb26]) << 1
43 data[d4ac] <- rd
44 data[deb4] <- (~(~data[d4ac] | data[cb26])) | (~data[d4ac] & data[cb26])
45 jump to 20e:0? (data[cb26] & data[cb26]) -> yes
46
47 rd <- ~(~data[d4ac] | data[cb26]) | (~data[d4ac] & data[cb26])
48 data[cb26] <- (data[d4ac] & data[cb26]) << 1
49 data[d4ac] <- rd
50 data[deb4] <- (~(~data[d4ac] | data[cb26])) | (~data[d4ac] & data[cb26])
51 jump to 20e:0? (data[cb26] & data[cb26]) -> no
52
53 r5 <- (~r5)
54 r5 <- (r5<<1)
55 data[f9f8] <- (data[d4ac]&data[d4ac])
56 jump to 560:0? (data[f9f8]&data[f9f8]) -> yes
57 ERROR Layer1
```

Listing 26 – Désassemblage des instructions exécutées par *layer1* pour la clé $0x12345678$

```
1  rc = key & 0xffff;
2  r3 = (key >> 16) & 0xffff;
3  data[0xc75e] = f1(rc, r3);
4  data[0xcf9a] = f2(rc, r3);
5  data[0xd89e] = f3(rc, r3);
6  data[0xf9f8] = f4(rc, r3);
7  // Saut inconditionnel ignore.
8  rd = (~0x1175) << 1;
9  do {
10   rd = (~(~data[0xd4ac] | data[0xcb26]) | (~data[0xd4ac] & data[0xcb26])) & 0xffff;
11   data[cb26] = (data[d4ac] & data[cb26]) << 1;
12   data[d4ac] = rd;
13   data[deb4] <- (~(~data[d4ac] | data[cb26]));
14 } while (data[0xcb26] > 0);
15 r5 = (~0x16b8) << 1;
16 if (data[0xf9f8] != 0) {
17   bad_key();
18 }
19 // Suite ignore.
20 // ...
```

Listing 27 – Début de décompilation de *layer1*

A priori, toutes les exécutions de *layer1* qui exhibe le même comportement que celui présenté sur le listing 27 correspondent à des clés invalides. Nous avons donc exécuté le *layer1* avec différentes clés en nous arrêtant au niveau du *if* de la ligne 16 du listing 27. Nous nous sommes aperçu que la différence entre les exécutions se situe au niveau du nombre d'itérations dans la boucle. L'exécution pour une clé étant assez rapide, nous avons opté pour, à présent, balayer toutes les clés en nous limitant à l'exécution du début de *layer1* et en comptant le nombre d'itérations pour la boucle. Le code utilisé est présenté sur le listing 28 et les résultats sont présentés sur le listing 29.

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int data[0x10000];

int test(unsigned int k) {
  int rc, r3, r0, r6, r1, r8, rd, r5, c;

  data[0xa002] = (k >> 16) & 0xffff;
  data[0xa000] = (k >>  0) & 0xffff;

  rc = data[0xa000];
```

```
  r3 = data[0xa002];
  r0 = rc >> 8;
  r6 = (~((r0 & rc) | (~(r0|rc)))) & 0xffff;
  r6 = ((~(r6 | (~0x9577))) | (r6 & (~0x9577))) & 0xffff;

  r1 = r3 >> 8;
  r8 =   ((r1 & ~r3) | ~(r1 | ~r3)) & 0xffff;
  r1 = rc & 0xff;
  r8 =   ((r8 &  r1) | ~(r8 |  r1)) & 0xffff;
  r1 = r1 << 8;
  r8 =   ((r8 &  r1) | ~(r8 |  r1)) & 0xffff;
  r1 = r0 << 8;
  r8 = (~((r8 &  r1) | ~(r8 |  r1))) & 0xffff;
  r8 = ((~(r8 & r0)) & (r8 | r0)) & 0xffff;

  rd = 0x1175;
  r5 = (~((~1 | (~0xae4d)) & (1 | 0xae4d))) & 0xffff;

  data[0xc75e] = r5;
  data[0xcf9a] = 0xae4d;
  data[0xd89e] = ((1 & (~0xae4d)) << 1) & 0xffff;
  data[0xf9f8] = r8;

  //#jump to 191:1? (data[0xd89e] & data[0xd89e]) -> no;
  //#print("jump 191:1? ", data[0xd89e]);
  if (data[0xd89e] != 0) {
    return 0;
  }
  rd = ((~rd) << 1) & 0xffff;
  r5 = 0x16b8;

  rd = (~(~data[0xf9f8] | data[0xc75e]) | (~data[0xf9f8] & data[0xc75e])) & 0xffff;
  data[0xcb26] = ((data[0xf9f8] & data[0xc75e]) << 1) & 0xffff;
  data[0xd4ac] = rd;
  data[0xdeb4] = (~(~data[0xf9f8] | data[0xc75e]) | (~data[0xf9f8] & data[0xc75e])) & 0xffff;
  //#jump to 20e:0? (data[0xcb26] & data[0xcb26]) -> yes;
  //#print("jump 20e:0? ", data[0xcb26]);
  if (data[0xcb26] == 0) {
    return 1;
  }
  c = 1;
  while (data[0xcb26] != 0) {
    rd = (~(~data[0xd4ac] | data[0xcb26]) | (~data[0xd4ac] & data[0xcb26])) & 0xffff;
    data[0xcb26] = ((data[0xd4ac] & data[0xcb26]) << 1) & 0xffff;
    data[0xd4ac] = rd;
    data[0xdeb4] = ((~(~data[0xd4ac] | data[0xcb26])) | (~data[0xd4ac] & data[0xcb26])) & 0
        xffff;
    //#jump to 20e:0? (data[0xcb26] & data[0xcb26]) -> yes;
    //#print("jump 20e:0? ", data[0xcb26]);
    c = c + 1;
  }
  return c;
}

int main(int argc, char** argv) {
  char* p;
  int res[18];
  unsigned int key_from;
  unsigned int key_to;
  key_to = strtoul(argv[2], &p, 16);
  key_from = strtoul(argv[1], &p, 16);
  memset(res, 0, 18 * sizeof(int));
  int i;
  for (i = key_from; i < key_to; i++) {
    int r = test(i);
    if (r > 16) {
      res[17]++;
    } else {
      res[r]++;
    }
  }
  for (i = 0; i < 18; i++) {
    printf("%d: %d\n", i, res[i]);
  }
```

```
  return 0;
}
```

Listing 28 – Comptage du nombre d'itérations en fonction de la clé

```
sh-3.2$ time ./comptage 0x0 0xffffffff
0: 0
1: 16777216
2: 385875968
3: 1138753536
4: 1276116992
5: 693108735
6: 385875968
7: 197918720
8: 116654080
9: 50331648
10: 16777216
11: 10485760
12: 4194304
13: 1048576
14: 524288
15: 393216
16: 131072
17: 0

real    6m13.578s
user    6m8.268s
sys 0m0.408s
```

Listing 29 – Résultat du comptage du nombre d'itérations en fonction de la clé

La plupart des clés entraînent entre 2 et 9 itérations dans la boucle. Le nombre maximal d'itérations est 16. Nous nous sommes dit que le comportement associé à la bonne clé devait se détacher de la plupart des comportements (en terme de nombre d'itérations). Notre intuitions nous a donc poussé à nous concentrer sur les clés qui entraînent exactement 16 itérations. Dans en premier temps, nous avons recherché l'ensemble de ces clés. Pour ce faire, il suffit d'afficher les valeurs des clés pour lesquelles nous atteignons 16 itérations, dans le programme précédent. Cette liste, nommée `list` dans les programmes *C* est ensuite utilisée pour alimenter la fonction `execute_layer1` dans le listing 30 qui se charge de tester chacune de ces clés à la recherche de la bonne. L'exécution de cette fonction nous fournit, 32 secondes plus tard, la bonne clé (`0x94e3e5df`), comme indiqué dans le listing 31. Notre intuition était donc bonne et elle nous a permi de nous passer de la décompilation de *layer1*.

```
void execute_layer1(uint from, uint to) {
  uint key = from;
  uint r4_save = r4;
  unsigned char* ram_save = malloc(RAMSIZE);
  memcpy(ram_save, ram, RAMSIZE);
  int idx_list;
  for (idx_list = 0; idx_list < sizeof(list) / sizeof(int); idx_list++) {
    key = list[idx_list];
    memcpy(ram, ram_save, RAMSIZE);
    r4 = r4_save;
    ram[0xa000] = key & 0xFF;
    ram[0xa001] = (key >>  8) & 0xFF;
    ram[0xa002] = (key >> 16) & 0xFF;
    ram[0xa003] = (key >> 24) & 0xFF;
    fatal = 0;
    stop = 0;
    uint cont = 1;
    while (cont && (fatal == 0) && (stop == 0)) {
      loc_14dah();
      cont = ! (GET_SHORT_FROM_RAM(0x111c) == 0x577);
    }
    if (fatal != 0) {
      printf("fatal with: %16x\n", (uint)key);
    }
    uint result1 = ram[0xa000] + (ram[0xa001] << 8) + (ram[0xa002] << 16) + (ram[0xa003] << 24)
        ;
    uint result2 = ram[0x8000] + (ram[0x8001] << 8) + (ram[0x8002] << 16) + (ram[0x8003] << 24)
        ;
```

```
    if (key != result1) {
      printf("valid: %16x - %16x - %16x\n", (uint)key, result1, result2);
      key_layer1 = key;
      key_valid = 1;
    }
    key++;
  }
}
```

Listing 30 – Fonction de test des clés pour *layer1*

```
sh-3.2$ time ./execute_layer1
valid:         94e3e5df -         d5fb8cfa -         beefdead

real    0m32.386s
user    0m32.030s
sys 0m0.033s
sh-3.2$ time ./execute_layer2
valid:             f63d -         f63d94e3 -         beefdead

real    30m18.969s
user    30m2.297s
sys 0m1.876s
sh-3.2$ time ./execute_layer3
valid:             8937 -         8937f63d -         beefdead

real    0m45.232s
user    0m44.620s
sys 0m0.067s
```

Listing 31 – Calcul des clés pour chaque *layer*

La suite de la recherche de la clé a été réglé très rapidement. La partie de la clé utilisée par *layer2* (resp. *layer3*) est codée sur 32 bits mais 16 de ces bits correspondent à des bits de la partie de la clé utilisée pour *layer1* (resp. *layer2*). Donc, une recherche parmi des nombres codés sur 16 bits est largement envisageable par force brute, avec notre interpréteur et sachant qu'une exécution pour une clé est très rapide. Au final, nous avons obtenu les autres parties de la clé en 30 minutes pour le *layer2* et 45 secondes pour le *layer3* (cf. listing 31).

7 Reconstitution et déchiffrement du fichier *secret*

Lorsque nous tentons de déchiffrer le fichier *secret* avec l'application *ssticrypt* en utilisant une clé quelconque, nous lisons un avertissement qui nous laisse comprendre que l'intégrité du fichier n'est pas valide (cf. listing 32). En scrutant son contenu, la suite d'octets à zéro à partir de l'adresse `0x3000` attire notre attention. Il est possible que le contenu du fichier *secret* complet ait été écrasé volontairement par cette suite de zéros. Il est envisageable également que seuls les 12 premiers kilooctets du fichier *secret* complet aient été copiés. Finalement, il se peut que le fichier *secret* complet soit inclus dans ce fichier et que des octets ont été ajoutés volontairement à la fin du fichier pour changer l'empreinte calculée. Quelque soit la méthode adoptée par les organisateurs, la démarche pour retrouver et reconstituer le fichier *secret* reste la même. Nous la détaillons dans la sous-section qui suit.

```
sh-3.2$ ./ssticrypt -d `python -c "print 'A'*32"` secret
--> SSTICRYPT <--
Warning: MD5 mismatch for container
Using keys AAAAAAAAAAAAAAAA / AAAAAAAAAAAAAAAA ...
Traceback (most recent call last):
  File "check.py", line 50107, in <module>
AssertionError
sh-3.2$ xxd -a -g1 secret | tail -n 5
0002fe0: af 26 b1 cb d3 84 19 00 df 75 6f 5f 61 89 c5 fc  .&.......uo_a...
0002ff0: c3 28 81 e9 39 3d 83 a4 1c 3d 37 27 e7 4d 9d 9d  .(..9=...=7'.M..
0003000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0100000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Listing 32 – Déchiffrement du fichier *secret* avec une clé quelconque

7.1 Reconstitution du fichier *secret*

Pour commencer, nous tentons de retrouver l'ensemble des fichiers, parmi ceux que l'outil *photorec* a pu extraire de l'image de disque *dump.img*, qui peuvent contenir le fichier *secret*. Pour cela, nous allons procéder par raffinements successifs : nous utilisons, pour commencer, les 8 premiers octets du fichier *secret* comme motif de recherche. En fonction du nombre de fichiers trouvés, nous déterminerons s'il est nécessaire d'utiliser un motif plus long. L'utilitaire *grep* [5] suffit pour rechercher les fichiers pouvant contenir *secret* (cf. listing 33).

Seul le fichier *b0000000.ext* semble contenir le fichier *secret*. Il n'est donc pas nécessaire d'utiliser un motif plus grand. La chaîne recherchée se trouve au 98463744e octet à partir du début du fichier. En faisant l'hypothèse que le début du fichier *secret* n'ait pas été altérée et connaissant à la fois son empreinte [6] et l'algorithme utilisé pour la calculer, nous allons essayer de le reconstituer. Nous extrayons alors les octets utiles du fichier *b0000000.ext* pour faciliter les calculs.

```
sh-3.2$ xxd -a -g1 secret | head -n 1
0000000: b8 4d b9 ec 23 52 4e 4e 55 77 03 fb 55 df c0 83  .M..#RNNUw..U...
sh-3.2$ grep -o -rasbP '\xb8\x4d\xb9\xec\x23\x52\x4e\x4e' photorec/
photorec/recup_dir.9/b0000000.ext:98463744:.M..#RNN
sh-3.2$ xxd -a -g1 -s 98463744 recup_dir.9/b0000000.ext | head -n1
5de7000: b8 4d b9 ec 23 52 4e 4e 55 77 03 fb 55 df c0 83  .M..#RNNUw..U...
sh-3.2$ dd if=recup_dir.9/b0000000.ext of=secret.inc bs=1024 skip=96156
80254+0 enregistrements lus
580254+0 enregistrements écrits
594180096 octets (594 MB) écopis, 10,0418 s, 59,2 MB/s
```

Listing 33 – Recherche des fichiers pouvant contenir le fichier *secret*

Pour reconstituer le fichier *secret*, nous avons adopté une approche par force-brute (cf. algorithme 1). Une implémentation de cet algorithme en langage Python est donnée dans l'annexe C. Concrètement, nous supposons à l'état initial que le fichier *secret* n'est constitué que d'un octet utile et plus de son empreinte. Nous calculons l'empreinte de cet octet utile, puis nous la comparons à celle que nous recherchons, qui est stockée dans les 16 premiers octets du fichier *secret*. Si les empreintes diffèrent, le fichier *secret* n'est pas encore complet, et nous allons considérer un nouvel octet utile supplémentaire. Nous recalculons l'empreinte de ces deux octets et nous la comparons à nouveau avec celle recherchée. Nous itérons ces calculs jusqu'à trouver la bonne empreinte. Pour accélérer les calculs, il est nécessaire de se rappeler que l'algorithme MD5 fonctionne itérativement sur des blocs de 512 bits. Il n'est donc pas nécessaire de recalculer l'empreinte de tous les octets à chaque fois qu'on considère des octets supplémentaires. Nous pouvons directement ré-utiliser les calculs effectués aux itérations précédentes.

No skills

Algorithm 1 Reconstitution du fichier *secret*

Require: fichier $secret.inc$
Ensure: fichier $secret.complet$ si succès
  $data \Leftarrow read('secret.inc')$
  if $data < 16$ **then**
    return 1
  end if
  $md5Ref \Leftarrow data[0:15]$
  for $offset \Leftarrow 16 \rightarrow len(data)$ **do**
    if $md5Ref = MD5(data[16:offset])$ **then**
      $write('secret.complet', data[0:offset])$
      return 0
    end if
  end for
  return 1

Nous reconstituons ainsi le fichier *secret* complet en moins de dix minutes avec une implémentation de notre algorithme en langage Python (cf. listing 34). Par comparaison du fichier *secret* original avec le fichier *secret* reconstitué, il semblerait que les organisateurs ait simplement écrasé une partie du fichier *secret* par une suite de zéros.

5. Il convient de noter que l'option `-o` configure *grep* de façon à n'afficher que la chaîne que l'on recherche. On évite ainsi de parasiter notre terminal avec une série de caractères non-imprimables.
6. Nous rappelons que l'empreinte du fichier *secret* est stockée dans les 16 premiers octets du fichier, et qu'elle a été calculée avec l'algorithme MD5.

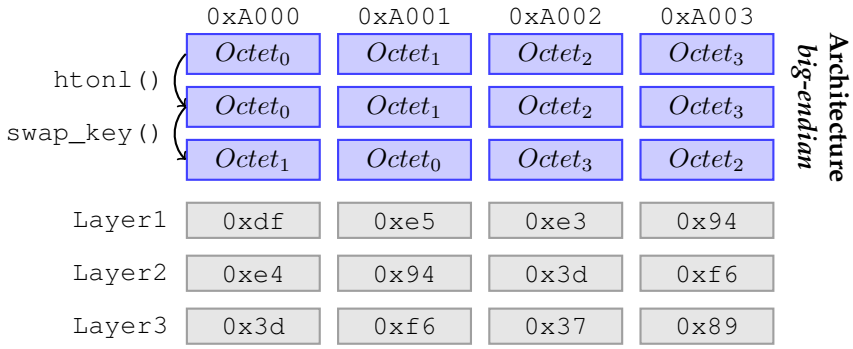

FIGURE 11 – Reconstitution de la partie de la clé validée par le périphérique USB

```
sh-3.2$ time python retrieve-secret.py secret.inc
real    7m10.254s
user    0m0.000s
sys     0m0.000s
```

Listing 34 – Temps de calcul nécessaire pour reconstituer le fichier *secret*

7.2 Reconstitution de la clé de déchiffrement

Nous avons à présent l'ensemble des octets qui composent la clé pour déchiffrer le fichier *secret* que nous venons de reconstituer. Il ne nous reste plus qu'à déterminer l'ordre dans lequel les saisir, puis vérifier si notre intuition sur la nature de celui-ci était correcte.

Les 8 premiers octets de la clé de déchiffrement – `fd4185ff66a94afd` – obtenus par cryptanalyse du *byte-code* python *check.pyc* sont indépendants de l'architecture de la machine. On peut alors garder telle quelle cette partie de la clé obtenue. En revanche, les 8 octets suivants – `dfe5e3943df63789` – que nous avons obtenus par une attaque de type force-brute sont dépendants de l'arrangement des octets en mémoire, et donc de l'architecture du processeur que nous avons utilisé. Nous avons effectué les calculs sur une machine de type Intel (architecture de type *little-endian*) et l'application *ssticrypt* doit être exécuté sur un processeur MIPS (architecture de type *bi-endian* utilisé en *big-endian*). Comme l'arrangement des octets en mémoire est différente sur les deux processeurs, il peut être nécessaire d'intervertir l'ordre de ces 8 derniers octets. Il est possible de déterminer l'ordre de ces 8 octets en déchiffrant le fichier *secret* avec tous arrangements possibles (et intelligents) de ceux-ci. À la place de cette approche, nous préférons décrire une solution plus élégante et réfléchie dans cette sous-section.

Pour déterminer l'arrangement des 8 derniers octets de la clé, nous allons nous focaliser sur la fonction *set_my_key()* de l'application *ssticrypt*. Nous rappelons que cette fonction se charge de découper les 8 derniers octets de la clé founie à *ssticrypt* en morceaux de 4 octets, de ré-organiser ces derniers avant de les copier à l'adresse `0xA000` dans la mémoire utilisée pour communiquer avec le périphérique USB. Les transformations effectuées sur ces morceaux de 4 octets sont représentées sur la figure 11. Nous considérons, sur cette figure, une exécution de l'application *ssticrypt* sur son architecture native, c'est-à-dire *big-endian*. L'organisation des octets, suite aux transformations effectuées par la fonction *set_my_key()*, serait différente sur une architecture *little-endian*.

Afin de faciliter la lecture, nous avons également rappelé sur la figure 11 les octets (déjà transformés) que nous avons identifiés par l'attaque de type force-brute (cf. section 2). Il nous suffit alors d'appliquer la transformation inverse pour déterminer l'arrangement adéquat pour les 8 derniers octets de la clé de déchiffrement. Ces 8 derniers octets devraient correspondre ainsi à `e5df94e3f63d8937`.

Nous déchiffrons alors le fichier *secret* pour valider notre raisonnement (cf. listing 35). Comme nous n'avons pas a notre disposition le périphérique USB nécessaire à la validation des 8 derniers octets, nous décidons de ré-implémenter l'application *ssticrypt* sans les fonctions de vérification de la clé. Cette dernière est identifiée par *ssticrypt-rc4* dans le listing 35. Elle a été obtenue par rétro-ingénierie et son code source est disponible en annexe B. Comme nous l'avons intuité, le fichier déchiffré *secret.dec* correspond effectivement à une image de disque. Notre raisonnement pour l'arrangement des 8 derniers octets est, par conséquent, correct et la clé de déchiffrement à utiliser pour le fichier *secret* est `fd4185ff66a94afd e5df94e3f63d8937`.

```
sh-3.2$ gcc -Wall ssticrypt-rc4.c -lcrypto -o ssticrypt-rc4
```

```
sh-3.2$ ./ssticrypt-rc4
usage: ./ssticrypt-rc4 [-d|-e] <key> <secure container>
sh-3.2$ ./ssticrypt-rc4 -d fd4185ff66a94afde5df94e3f63d8937 secret
Using keys fd4185ff66a94afd / e5df94e3f63d8937 ...
sh-3.2$ ls -al
drwxr-xr-x  2 root     root 1728512 May 20 21:06 .
drwxrwxrwt 10 root     root    4096 May 20 19:25 ..
-rwxr-xr-x  1 root     root 1048592 May 16 01:32 secret
-rw-r--r--  1 root     root 1048576 May 20 21:06 secret.dec
-rwxr-xr-x  1 root     root   11971 May 19 22:15 ssticrypt-rc4
-rw-r--r--  1 root     root    2695 May 16 01:32 ssticcrypt-rc4.c
sh-3.2$ file secret.dec
secret.dec: Linux rev 1.0 ext2 filesystem data
```

Listing 35 – Vérification de la clé et déchiffrement du fichier *secret*

7.3 Analyse du fichier *secret* déchiffré : *secret.dec*

Dans cette section, nous allons analyser le contenu de cette image de disque en esperant y trouver dans le système de fichier un ou plusieurs documents contenant l'adresse e-mail que nous recherchons.

Nous chargeons l'image du disque *secret.dec* afin d'explorer son contenu (cf. listing 36). Nous procédons pour cela de la même façon que précédemment, lorsque nous avions chargé le fichier *dump.img* pour l'analyser (cf. section 2).

```
sh-3.2$ sudo modprobe loop
sh-3.2$ sudo losetup /dev/loop0 secret.dec
sh-3.2$ sudo mount /dev/loop0 /mnt
sh-3.2$ ls /mnt
sh-3.2$ ls -al /mnt
drwxr-xr-x  3 root     root       1024 19 mars  17:41 .
drwxr-xr-x 22 root     root       4096 30 mars  09:18 ..
-rw-r--r--  1 root     root     922290 19 mars  17:41 lobster
drwx------  2 root     root      12288 19 mars  17:40 lost+found
sh-3.2$ file /mnt/*
/mnt/lobster:    RIFF (little-endian) data, AVI, 352 x 264, ~15 fps, ...
/mnt/lost+found: directory
```

Listing 36 – Lecture de la vidéo *lobster* et extraction d'une image

Nous remarquons que le fichier *secret.dec* contient un fichier vidéo *lobster* au format RIFF et un dossier *lost+found* vide, tous deux crées le 19 mars 2012, soit datant de quelques jours avant la mise en ligne du Challenge SSTIC (26 mars 2012). Nous rappelons que le dossier *lost+found* est inhérent aux partitions de type étendue (partitions *ext*) et contient généralement les fichiers corrompus que l'utilitaire *fsck*[7] a pu reconstituer suite à un arrêt anormal du système, par exemple suite à un *crash*, ou lorsqu'une partition n'a pas été démontée correctement à l'arrêt du système. En supposant que l'image *secret.dec* a été correctement démontée, il est donc normal que ce dossier soit vide. Nous allons donc nous focaliser, dans la suite, sur le fichier vidéo *lobster*.

7.4 Chargement de la vidéo *lobster* et récupération de l'adresse e-mail

Intuition

Le fichier vidéo *lobster* correspond probablement à l'ultime étape de l'épreuve. Il convient de se rappeler que l'adresse e-mail à retrouver pour le concours de l'année dernière était intégrée à une vidéo. La nature du fichier *lobster* nous laisse penser qu'il est probable que cela soit également le cas pour l'épreuve de cette année. Notre intuition s'avère bonne lorsqu'on charge la vidéo dans un lecteur multimédia (cf. listing 46). On distingue en effet l'adresse e-mail recherchée à partir de la 32e trame de la vidéo (cf. figure 12).

```
sh-3.2$ mplayer /mnt/lobster
MPlayer SVN-r34799-4.6.3 (C) 2000-2012 MPlayer Team
183 audio & 398 video codecs
[...]
sh-3.2$ ffmpeg -i /mnt/lobster -r 15 -f image2 /tmp/images%03d.png
ffmpeg version 0.10.2 Copyright (c) 2000-2012 the FFmpeg developers
  built on Mar 17 2012 08:51:02 with gcc 4.6.3
```

7. *File System Check* ou *File System consistency Check* est un outil sous UNIX pour vérifier la cohérence d'un système de fichiers.

```
[...]
sh-3.2$ feh /tmp/images032.png
```

Listing 37 – Lecture de la vidéo *lobster* et extraction d'une image

FIGURE 12 – Image *images032.png* extraite de la vidéo *lobster*

8 Conclusion

Cette épreuve a été une expérience enrichissante. Elle aborde différents aspects de la sécurité informatique tels que l'investigation informatique légale, la rétro-ingénierie d'applications, la cryptanalyse et, dans une certaine mesure, l'ingénierie sociale. La résolution de ce concours a été l'occasion pour nous de revoir la théorie de la décompilation, de créer de nouveaux outils, et réfléchir aux différentes approches possibles (typiques mais également atypiques) pour résoudre un problème donné. Pour ces raisons, nous avons trouvé ce concours à la fois original, complet et intéressant. Il convient cependant de noter que cette épreuve a été beaucoup plus difficile que celle des années précédentes. Cela a probablement découragé de nombreux participants à terminer le concours.

Remerciements

Nous tenons à remercier les organisateurs de cette édition du Challenge SSTIC – Florent Marceau et Fabien Perigaud du Cert-Lexsi et Axel Tillequin d'EADS Innovation Works – pour cette épreuve qui nous a donné beaucoup de fil à retordre. Merci également à toute l'équipe *Tolérance aux fautes et sûreté de Fonctionnement informatique* (TSF) du *Laboratoire d'Analyse et d'Architecture des Systèmes* (LAAS-CNRS), qui nous ont soutenu (et distrait maintes fois) tout le long de notre périple. Finalement, des remerciements plus particuliers à Maxime Lastera, Vincent Nicomette et Yves Deswarte, pour leurs conseils et leurs remarques constructives qui ont permis d'améliorer la qualité de cet article.

Références

[1] aurel32. Répertoire d'Aurélien Jarno - /qemu/mips, January 2011. http://people.debian.org/~aurel32/qemu/mips/.

[2] Des. Data encryption standard. In *FIPS PUB 46, Federal Information Processing Standards Publication*, pages 46–2, 1977.

[3] L. Peter Deutsch. GZIP file format specification version 4.3, December 1996. http://www.gzip.org/zlib/rfc-gzip.html.

[4] Yoann Guillot et Julien Tinnes. The METASM assembly manipulation suite, October 2010. http://metasm.cr0.org/.

[5] Etienne Dublé. Le monde merveilleux de LD_PRELOAD. *OpenSilicium*, 4 :58–64, October-November-December 2011.

[6] GDB developers. GDB : The GNU Project Debugger, 5 October 2011. http://sources.redhat.com/gdb/.

[7] Christophe Grenier. Photo Recovery - PhotoRec, 15 November 2011. http://www.cgsecurity.org/wiki/PhotoRec_FR.

[8] Haypo. Hachoir, July 2010. https://bitbucket.org/haypo/hachoir/wiki/Home.

[9] Hex-Rays. IDA, 5 October 2011. http://www.hex-rays.com/products/ida/index.shtml.

[10] John Hyde. USB Multi-Role Device – Design by example, 2003. http://www.cypress.com/?docID=14347.

[11] S. Josefsson. The Base16, Base32, and Base64 Data Encodings, October 2006. http://tools.ietf.org/html/rfc4648.

[12] MIPS Technologies. MIPS Architecture, 2012. http://www.mips.com/products/product-materials/processor/mips-architecture/.

[13] OpenSSL Project. OpenSSL – Cryptography and SSL/TLS toolkit, 10 May 2012. http://www.openssl.org/.

[14] Projet Debian. Informations sur la version « Lenny » de Debian, 10 March 2012. http://www.debian.org/releases/lenny/.

[15] R. Rivest. MD5 Algorithm, 2 April 1992. http://www.kleinschmidt.com/edi/md5.htm.

[16] Cypress Semiconductor. CY16 – USB Host/Slave Controller/16-Bit – RISC Processor Programmers Guide, 2003. http://www.cypress.com/?docID=14345.

[17] David Sterndark. RC4 Algorithm revealed, 15 September 1994. http://groups.google.com/group/sci.crypt/msg/10a300c9d21afca0.

[18] STIC. Wiki de la communauté SSTIC - Challenge SSTIC 2012, 22 March 2012. http://wiki.sstic.org/ChallengeSSTIC2012.

[19] Tool Interface Standard (TIS). Portable Formats Specification – version 1.1, October 1993. http://refspecs.linuxbase.org/elf/TIS1.1.pdf.

A Code source de *ssticrypt* obtenu par rétro-ingénierie

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <arpa/inet.h>
#include <usb.h>
#include <openssl/rc4.h>
#include <openssl/md5.h>

unsigned char  *check_pyc, *init_rom, *stage2_rom;
unsigned int   check_pyc_len = 0x4457d, init_rom_len = 0x44,
    stage2_rom_len = 0xA76;

unsigned char  *ram, *layer1, *layer2, *layer3, *blob, *blah;
unsigned int   o,layer_len[3] = { 0x65A, 0xEE2, 0x6E3 };
void           *all_layers[3] = { &layer1, &layer2, &layer3 };

usb_dev_handle *devCam = NULL;

// Quelques fonctions helpers

int swap_word(short num) { return (num << 8) | (num >> 8); }

void swap_key(int *key32b) {
  char b3 = (char)(*key32b >> 24);
  char b2 = (char)(*key32b >> 16);
  char b1 = (char)(*key32b >>  8);
  char b0 = (char)(*key32b >>  0);

  *key32b = (b2 << 24) | (b3 << 16) | (b0 <<  8) | (b1 <<  0);
}

// Fonctions pour la évrification de la écl par le ééppriphrique USB

usb_dev_handle *vicpwn_init(int idProduct)
{
    usb_dev_handle *devCam;

    usb_init();
    usb_find_busses();
    usb_find_devices();
    struct usb_bus *busses = usb_get_busses();
    while (busses != NULL) {
        struct usb_device *dev = busses->devices;
        while (dev != NULL) {
            if ((dev->descriptor.idVendor == 0x04c1) &&
                (dev->descriptor.idProduct == idProduct))
                devCam = usb_open(dev);
            dev = dev->next;
        }
        busses = busses->next;
    }

    return devCam;
}

void vicpwn_close(usb_dev_handle *devCam) { usb_close(devCam); };
void vicpwn_quit(usb_dev_handle *devCam) { vicpwn_close(devCam); exit(0);
     }

void vicpwn_sendbuf(void *buf, int buflen)
{
    devCam = vicpwn_init(157);
    if (devCam == NULL) {
        exit(-1);
    }
    usb_control_msg(devCam, 64, 0xFF, 0, 0, buf, buflen, 1000);
    vicpwn_close(devCam);
}

void load_layer(void *layers, int cnt, int offset)
{
    memcpy(&ram[offset], layers, layer_len[cnt]);
    if (cnt <= 0) free(layers);
}

void set_my_key(char *key, int cnt, int offset) {
    int lkey[4];
    int key32b;

    strcpy((char *)lkey,key);
    lkey[cnt+2] = '\0';

    key32b = htonl(strtoul((char *)&lkey[cnt], NULL, 16));
    swap_key(&key32b);
```

```
    memcpy(&ram[offset],&key32b,4);

    if (cnt == 1)
        memcpy(&ram[offset+16], blob, 0x100);
    else if (cnt == 2)
        memcpy(&ram[offset+16], blah, 0x20);
}

void *vicpwn_check(int cnt, int offset, char *key) {
    int key32b, new_key32b;
    char ascii_key[16], ksa[256];
    int aux, i, j, index;
    int *decoded_layer, *layer;
    int len = layer_len[cnt+1];

    layer = all_layers[cnt+1];
    decoded_layer = calloc(1, len);

    memcpy(&new_key32b, &ram[offset], 4);
    swap_key(&new_key32b);

    strncpy(ascii_key,key,16);
    ascii_key[8] = '\0';
    key32b = htonl(strtoul(ascii_key, NULL, 16));

    if (cnt == 0) {
        decoded_layer[0] = layer[0] ^ new_key32b;
        index = 1;

        do {
            decoded_layer[index] = layer[index] ^ decoded_layer[index-1];
            index++;
        } while (index < (len/4));

        if (new_key32b == key32b) {
            puts("Error: bad key2");
            exit(1);
        }
    } else if (cnt == 1) {
        memcpy(ksa, &ram[offset+16], 0x100);

        if (ksa[0xfe] == 0xff && ksa[0xff] == 0xff) {
            printf("Error: bad key2\n");
            exit(-1);
        }

        i = 0; j= 0; index = 0;
        while (index < len) { // RC4 avec KSA precalculee
            i += 1;
            j += ksa[i+4];
            aux = ksa[i+4];
            ksa[i+4] = ksa[j+4];
            ksa[j+4] = aux;
            ((char *)decoded_layer)[index] = ((char *)layer)[index] ^ (
                ksa[4+((ksa[i+0x4] + ksa[j+0x4]) & 0xff)]);
            index++;
        }
    } else if (cnt == 2) {
        // base64.decode('V29vdCAhISBTbWVsbHMgZ29vZCA6KQ==') = 'Woot !!
            Smells good :)'
        if (strncmp((char *)&ram[offset+16], "
            V29vdCAhISBTbWVsbHMgZ29vZCA6KQ==", 32)) {
            printf("Error: bad key2\n");
            exit(-1);
        }
        free(decoded_layer);
    }
    return decoded_layer;
}

void vicpwn_handle(char *key) {
    char *decoded_layer, *buf = malloc(20);
    unsigned int cnt, bytes, waddr;

    o = 0;
    devCam = vicpwn_init(157);
    if (devCam == NULL) {
        printf("No Dev found\n");
        exit(-1);
    }

    decoded_layer = all_layers[0];
    for (cnt = 0; cnt < 3; cnt++) {
        load_layer(decoded_layer, cnt, 0);
        set_my_key(key, cnt, 0xa000);
        while (*(int*)&ram[0x8000] == 0 || o != 0) {
            o &= 0xfff0;
            if ((o & 0xfff0) != 0xfff0) {
                memcpy(buf, o + ram, 16);
                usb_control_msg(devCam, 64, 81, o, 1, buf, 16, 1000);
            }
            memset(buf, 0, 20);
            bytes = usb_control_msg(devCam, 192, 86, 0, 0, buf, 20, 1000)
                ;
```

```
            waddr = swap_word(ntohs(*(short*)&buf[bytes-4]) & 0xffff);
            o = swap_word(ntohs(*(short*)&buf[bytes-2]) & 0xffff);
            bytes = 20;
            if (((waddr & 0xfff0) == 0xfff0) || ((waddr & 1) == 0))
                break;

            waddr &= 0xfff0;
            memcpy(ram + waddr, buf, 16);
        }
        decoded_layer = vicpwn_check(cnt, 0xa000, key);
        cnt++;

        *(int*)&ram[0x8000] = 0;
    }
    usb_release_interface(devCam, 0);
    free(buf);
    vicpwn_close(devCam);
}

// Fonction de relative à la verification de la écl par le code python
void extract_pyc(void)
{
    int fd = open("./check.pyc", 257, 493);
    if (fd == -1) {
        perror("open");
        exit(-1);
    }
    write(fd, check_pyc, check_pyc_len);
    close(fd);
}


int check_key(char *key, int id) {
    if (id == 1) {
        char command[40];

        extract_pyc();
        sprintf(command, "python ./check.pyc %16s", key);
        int ret = system(command);
        unlink("./check.pyc");
        if (ret != 0) {
            puts("Error: bad key1");
            exit(1);
        }
    } else { // id == 2
        signal(SIGINT, (void (*)(int))(vicpwn_quit));
        ram = malloc(0x10000);
        vicpwn_sendbuf(init_rom, init_rom_len);
        vicpwn_sendbuf(stage2_rom, stage2_rom_len);
        vicpwn_handle(key);
    }
    return 0;
}

void show_key(char *key1, char *key2) {
    printf("Using keys %s / %s ...\n", key1, key2);
}

void check_duck(char *key1, char *key2) {
    if (!strcmp(key1, "5132397062694567") &&
        !strcmp(key2, "513239706269453d")) {
        printf("Error: You're a duck\n");
        exit(1);
    }
}

void usage(char *prog) {
    printf("usage: %s [-d|-e] <key> <secure container>\n", prog);
    exit(-1);
}


int main(int argc, char **argv) {

    int off = 0, mode = 0;
    int fd, size, index;
    unsigned char md5f[16], md5c[16];
    unsigned char *inbuf, *outbuf;
    char key1[20], key2[20];
    char *filename;

    RC4_KEY key;

    printf("--> SSTICRYPT <--\n");

    if (argc != 4)
        usage(argv[0]);

    if (strlen(argv[2]) != 32) {
      printf("Error: key should be a 128-bits hexadecimal string\n");
      exit(1);
    }

    if (strcmp(argv[1], "-e") == 0)
        mode = 1;
```

```
    fd = open(argv[3], O_RDONLY);
    if (fd < 0) {
      perror("open");
      exit(1);
    }

    size = lseek(fd, 0, 2);
    lseek(fd, 0, 0);

    if (mode == 0) { // Decipher mode
      read(fd, md5f, 16);
      size -= 16;
    }

    inbuf = calloc(size, 1);
    if (inbuf == NULL) {
        perror("calloc");
        exit(1);
    }

    do {
        off += read(fd, inbuf, size);
    } while (off != size);
    close(fd);

    if (mode == 0) { // Decipher mode
        MD5(inbuf, size, md5c);
        if (memcmp(md5f, md5c, 16) != 0)
            puts("Warning: MD5 mismatch for container");
    }

    outbuf = calloc(size, 1);
    strncpy(key1, argv[2], 16);
    strncpy(key2, argv[2] + 16, 16);
    key1[16] = '\0'; key2[16] = '\0';

    show_key(key1, key2);
    check_duck(key1, key2);

    if (mode == 0) {
        check_key(key1, 1);
        check_key(key2, 2);
    }

    RC4_set_key(&key, 32, (unsigned char*) argv[2]);

    if (mode == 0) {
        index = 1;
        while (index < size) {
            inbuf[index-1] = (inbuf[index-1] ^ inbuf[index]) & 0xFF;
            index++;
        }
    }

    RC4(&key, size, inbuf, outbuf);

    if (mode != 0) { // Cipher mode
        index = size - 1;
        while (index > 0) {
            outbuf[index-1] = (outbuf[index-1] ^ outbuf[index]) & 0xFF;
            index--;
        }
    }

    filename = calloc(strlen(argv[3]) + 5, 1);
    sprintf(filename, mode ? "%s.enc" : "%s.dec", argv[3]);

    fd = open(filename, 257, 493);
    if (fd < 0) {
        perror("open");
        exit(-1);
    }

    if (mode != 0) { // Cipher mode
        MD5(outbuf, size, md5c);
        write(fd, md5c, 16);
    }

    write(fd, outbuf, size);
    close(fd);

    free(filename);
    free(outbuf);
    free(inbuf);

    return 0;
}
```

Listing 38 – Code source de *ssticrypt* obtenu par rétro-ingénierie

B Code source de *ssticrypt-rc4.c*

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <openssl/rc4.h>
#include <openssl/md5.h>
#include <sys/stat.h>
#include <fcntl.h>

void show_key(char *key) {
    char key1[20], key2[20];

    strncpy(key1, key, 16);
    key1[16] = '\0';

    strncpy(key2, key + 16, 16);
    key2[16] = '\0';

    printf("Using keys %s / %s ...\n", key1, key2);
}

void usage(char *prog) {
    printf("usage: %s [-d|-e] <key> <secure container>\n", prog);
    exit(1);
}

int main(int argc, char **argv) {

    int off = 0, mode = 0;
    int fd, size, index;
    unsigned char md5f[16], md5c[16];
    unsigned char *inbuf, *outbuf;
    char *filename;

    RC4_KEY key;

    if (argc != 4)
        usage(argv[0]);

    if (strlen(argv[2]) != 32) {
      puts("Error: key should be a 128-bits hexadecimal string");
      exit(1);
    }

    if (strcmp(argv[1], "-e") == 0)
        mode = 1;

    fd = open(argv[3], O_RDONLY);
    if (fd < 0) {
      perror("open");
      exit(1);
    }

    size = lseek(fd, 0, 2);
    lseek(fd, 0, 0);

    if (mode == 0) { // Decipher mode
      read(fd, md5f, 16);
      size -= 16;
    }

    inbuf = calloc(size, 1);
    if (inbuf == NULL) {
        perror("calloc");
        exit(1);
    }

    do {
        off += read(fd, inbuf+off, size);
    } while (off != size);
    close(fd);

    if (mode == 0) { // Decipher mode
        MD5(inbuf, size, md5c);
        if (memcmp(md5f, md5c, 16) != 0)
            puts("Warning: MD5 mismatch for container");
    }

    outbuf = calloc(size, 1);
    show_key(argv[2]);

    RC4_set_key(&key, 32, (unsigned char*) argv[2]);

    if (mode == 0) { // Decipher mode
        index = 1;
        while (index < size) {
            inbuf[index-1] = (inbuf[index-1] ^ inbuf[index]) & 0xFF;
            index++;
```

```
        }
    }

    RC4(&key, size, inbuf, outbuf);

    if (mode != 0) { // Cipher mode
        index = size - 1;
        while (index > 0) {
            outbuf[index-1] = (outbuf[index-1] ^ outbuf[index]) & 0xFF;
            index--;
        }
    }

    filename = calloc(strlen(argv[3]) + 5, 1);
    if (mode != 0)
        sprintf(filename, "%s.enc", argv[3]);
    else
        sprintf(filename, "%s.dec", argv[3]);

    fd = open(filename, O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP
        );
    if (fd < 0) {
        perror("open");
        exit(1);
    }

    if (mode != 0) { // Cipher mode
        MD5(outbuf, size, md5c);
        write(fd, md5c, 16);
    }

    write(fd, outbuf, size);
    close(fd);

    free(filename);
    free(outbuf);
    free(inbuf);

    return 0;
}
```

Listing 39 – Ré-implémentation de *ssticrypt* sans les fonctions de vérification

C Code source de *retrieve-secret.py*

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

import hashlib
import sys

def usage():
    print("usage: %s <secret.inc>")
    sys.exit(1)

if __name__ == '__main__':

    # Verification des parametres
    if len(sys.argv) != 2:
        usage()

    # Lecture du fichier qui contient secret
    secret = open(sys.argv[1], 'rb')
    data = secret.read()
    secret.close()

    # Verification sur le fichier
    if len(data) < 16:
        sys.exit(1)

    # Extraction de l'empreinte recherchee
    md5_ref = data[:16]

    # Reconstruction du fichier secret
    md5 = hashlib.md5()
    offset = 16

    for c in data[16:]:
        md5.update(c)
        md5_calc = md5.digest()

        if md5_calc == md5_ref:
            f = open('%s.complet' % sys.argv[1], 'wb')
            f.write(data[:offset])
            f.close()
            sys.exit(0)

        offset += 1

    sys.exit(1)
```

Listing 40 – Script pour la reconstitution du fichier *secret*

D Code source de *decompilateur.py*

```
import opcode
import inspect
import marshal
import string
import dis
import pickle

class Instruction(object):
  def __init__(self, op, arg=None, real_idx=None, new_idx=None):
    self.op = op
    self.arg = arg
    self.real_idx = real_idx
    self.new_idx = new_idx

  def opname(self):
    return opcode.opname[self.op]

  def __eq__(self, other):
    if not isinstance(other, Instruction):
      return NotImplemented
    return self.opname == other.opname and self.arg == other.arg

  def finish(self, instructions):
    if self.op in opcode.hasjabs or self.op in opcode.hasjrel:
      for idx, instr in enumerate(instructions):
        if instr.real_idx == self.arg:
          self.arg = idx
          break
      else:
        assert False

def parse_bytecode(code):
  i = 0
  bytecodes = map(ord, code.co_code)
  instructions = []
  # UNPACK_SEQUENCE, MAKE_FUNCTION, BUILD_TUPLE, BUILD_LIST,
      CALL_FUNCTION,
  # COMPARE_OP, RAISE_VARARGS, BUILD_SLICE or hasjabs
  hasnum = [92, 132, 102, 103, 131, 106, 130, 133] + opcode.hasjabs
  while i < len(bytecodes):
    op = bytecodes[i]
    opidx = i
    i += 1
    arg = None
    if op >= opcode.HAVE_ARGUMENT:
      oparg = bytecodes[i] + (bytecodes[i + 1] << 8)
      i += 2
      if op in opcode.hasconst:
        arg = code.co_consts[oparg]
      elif op in opcode.hasname:
        arg = code.co_names[oparg]
      elif op in hasnum:
        arg = oparg
      elif op in opcode.hasjrel:
        arg = i + oparg
      elif op in opcode.haslocal:
        arg = code.co_varnames[oparg]
      else:
        print "!", opcode.opname[op]
        raise NotImplementedError
    instructions.append(Instruction(op, arg, opidx, len(instructions),))
  for instr in instructions:
    instr.finish(instructions)
  return instructions

class BasicBlock(object):

  def __init__(self):
    self.instructions = []

class BasicBlockFinder(object):

  def __init__(self, instructions):
    self.instructions = instructions
    self.pending_basic_blocks = {}
    self.starting_basic_block = self.pending_basic_blocks[0] = BasicBlock
        ()

  def find_basic_blocks(self):
    current_basic_block = None
    for instr in self.instructions:
      if instr.new_idx in self.pending_basic_blocks:
        new_block = self.pending_basic_blocks.pop(instr.new_idx)
        current_basic_block = new_block
      handler = "handle_%s" % opcode.opname[instr.op].replace("+", "_")
      handler = getattr(self, handler)
      current_basic_block.instructions.append(instr)
      handler(instr)
```

```
    return self.starting_basic_block

  def get_basic_block(self, idx):
    return self.pending_basic_blocks.setdefault(idx, BasicBlock())

  def handle_simple_op(self, instr):
    pass

  handle_LOAD_CONST = handle_LOAD_FAST = handle_LOAD_ATTR = \
    handle_LOAD_GLOBAL = handle_STORE_FAST = handle_STORE_SUBSCR = \
    handle_BUILD_LIST = handle_BUILD_TUPLE = handle_CALL_FUNCTION = \
    handle_JUMP_ABSOLUTE = handle_POP_BLOCK = handle_BINARY_ADD = \
    handle_DUP_TOP = handle_ROT_THREE = handle_ROT_TWO = \
    handle_BINARY_SUBTRACT = handle_BINARY_RSHIFT = handle_BINARY_LSHIFT
        = \
    handle_BUILD_SLICE = handle_BINARY_XOR = handle_BINARY_OR = \
    handle_COMPARE_OP = handle_BINARY_FLOOR_DIVIDE = handle_STORE_SLICE_3
         = \
    handle_NOP = handle_SLICE_3 = handle_SLICE_2 = handle_BINARY_SUBSCR =
         \
    handle_RETURN_VALUE = handle_POP_TOP = handle_IMPORT_NAME = \
    handle_IMPORT_FROM = handle_STORE_NAME = handle_MAKE_FUNCTION = \
    handle_LOAD_NAME = handle_PRINT_NEWLINE = handle_PRINT_ITEM = \
    handle_BUILD_CLASS = handle_UNPACK_SEQUENCE = handle_BINARY_AND = \
    handle_BINARY_MODULO = handle_STORE_ATTR = handle_SLICE_1 = \
    handle_BINARY_MULTIPLY = handle_INPLACE_ADD = handle_simple_op

  def handle_SETUP_LOOP(self, instr):
    instr.true_block = self.get_basic_block(instr.new_idx + 1)
    instr.false_block = self.get_basic_block(instr.arg)

  def handle_JUMP_IF_TRUE(self, instr):
    instr.false_block = self.get_basic_block(instr.new_idx + 1)
    instr.true_block = self.get_basic_block(instr.arg)

  def handle_POP_JUMP_IF_FALSE(self, instr):
    instr.true_block = self.get_basic_block(instr.new_idx + 1)
    instr.false_block = self.get_basic_block(instr.arg)
  handle_JUMP_IF_FALSE = handle_POP_JUMP_IF_FALSE

  def handle_JUMP_FORWARD(self, instr):
    instr.fallthrough = self.get_basic_block(instr.arg)

  def handle_RAISE_VARARGS(self, instr):
    instr.fallthrough = self.get_basic_block(-1)

  def handle_GET_ITER(self, instr):
    instr.loop_var = self.instructions[instr.new_idx + 2].arg
    self.instructions[instr.new_idx + 2].op = opcode.opmap["NOP"]
    instr.loop = self.get_basic_block(instr.new_idx + 1)

  def handle_FOR_ITER(self, instr):
    instr.fallthrough = self.get_basic_block(instr.arg)

class Interpreter(object):

  def __init__(self, basic_block, indent_level=1):
    self.basic_blocks = [basic_block]
    self.ops = []
    self.buf = []
    self.indent_level = indent_level

  def get_and_clear_buf(self, expected_len):
    assert len(self.buf) >= expected_len
    buf = self.buf[(len(self.buf) - expected_len):len(self.buf)]
    del self.buf[(len(self.buf) - expected_len):len(self.buf)]
    return buf

  def emit(self, op):
    if isinstance(op, Interpreter):
      self.ops.append(op.evaluate())
    else:
      self.ops.append("  " * self.indent_level + op)

  def indent(self):
    self.indent_level += 1
    try:
      yield
    finally:
      self.indent_level -= 1

  def evaluate(self):
    while self.basic_blocks:
      basic_block = self.basic_blocks.pop()
      self.handle_basic_block(basic_block)
    return "\n".join(self.ops)

  def handle_basic_block(self, basic_block):
    for instr in basic_block.instructions:
      handler = "handle_%s" % opcode.opname[instr.op].replace("+", "_")
      handler = getattr(self, handler)
      handler(instr)

  def handle_NOP(self, instr):
    pass
  handle_POP_BLOCK = handle_NOP
```

```
def handle_literal(self, instr):
  if isinstance(instr.arg, str):
    self.buf.append("%s" % str(instr.arg))
  else:
    self.buf.append(str(instr.arg))
handle_LOAD_CONST = handle_LOAD_FAST = handle_LOAD_GLOBAL =
    handle_literal

def handle_BUILD_SLICE(self, instr):
  if instr.arg == 2:
    a, b = self.get_and_clear_buf(instr.arg)
    self.buf.append("range(%s,%s)" % (a, b))
  else:
    a, b, c = self.get_and_clear_buf(instr.arg)
    self.buf.append("range(%s,%s,%s)" % (a, b, c))

def handle_LOAD_ATTR(self, instr):
  [obj] = self.get_and_clear_buf(1)
  self.buf.append("%s.%s" % (obj, instr.arg))

def handle_STORE_ATTR(self, instr):
  [tos1, tos] = self.get_and_clear_buf(2)
  self.buf.append("%s.%s = %s" % (tos, instr.arg, tos1))

def handle_UNPACK_SEQUENCE(self, instr):
  seq = self.get_and_clear_buf(1)
  for i in range(instr.arg):
    self.buf.append("%s[%d]" % (seq, i))

def handle_ROT_TWO(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append(b)
  self.buf.append(a)

def handle_ROT_THREE(self, instr):
  a, b, c = self.get_and_clear_buf(3)
  self.buf.append(c)
  self.buf.append(a)
  self.buf.append(b)

def handle_DUP_TOP(self, instr):
  [a] = self.get_and_clear_buf(1)
  self.buf.append(a)
  self.buf.append(a)

def handle_BINARY_XOR(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("%s ^ %s" % (a, b))

def handle_STORE_SLICE_3(self, instr):
  a, b, c, d = self.get_and_clear_buf(4)
  self.emit("%s[%s:%s] = %s" % (b, c, d, a))

def handle_BINARY_SUBSCR(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("%s[%s]" % (a, b))

def handle_BUILD_LIST(self, instr):
  obj = self.get_and_clear_buf(instr.arg)
  self.buf.append(str(obj))

def handle_BUILD_TUPLE(self, instr):
  obj = self.get_and_clear_buf(instr.arg)
  self.buf.append("tuple(" + str(obj) + ")")

def handle_MAKE_FUNCTION(self, instr):
  self.buf.append("'%s'" % self.get_and_clear_buf(1)[0])

def handle_BUILD_CLASS(self, instr):
  code = self.get_and_clear_buf(1)
  self.get_and_clear_buf(1)
  self.buf.append("1#class %s: %s" % ("".join(self.get_and_clear_buf(1)
      ), code))

def handle_LOAD_NAME(self, instr):
  self.buf.append(instr.arg)

def handle_PRINT_ITEM(self, instr):
  item = self.get_and_clear_buf(1)
  self.emit("print('%s')" % (item[0]))

def handle_PRINT_NEWLINE(self, instr):
  self.emit("print('\\n')")

def handle_STORE_FAST(self, instr):
  [obj] = self.get_and_clear_buf(1)
  nv = "tmp"
  swap = False
  for i, v in enumerate(self.buf):
    if instr.arg in v:
      swap = True
    self.buf[i] = v.replace(instr.arg, nv)
  if swap:
    self.emit("%s = %s" % ("tmp", instr.arg))
  self.emit("%s = %s" % (instr.arg, obj))
```

```
def handle_IMPORT_NAME(self, instr):
  [level, fromlist] = self.get_and_clear_buf(2)
  i = "__import__(%s, fromlist=%s, level=%s)" % (instr.arg, fromlist,
      level)
  self.buf.append(i)

def handle_IMPORT_FROM(self, instr):
  a = self.get_and_clear_buf(1)
  self.buf.append(a[0])
  if a[0].startswith("__import__("):
    self.buf.append("%s.%s" % (a[0], instr.arg))
  else:
    self.buf.append("import %s from %s" % (instr.arg, ''.join(a)))

def handle_STORE_NAME(self, instr):
  a = self.get_and_clear_buf(1)
  right = ''.join(a)
  right = string.replace(''.join(a), '\n', '\\n')
  #if len(right) > 80:
  #  right = right[:75] + "..."
  self.emit("%s = %s" % (instr.arg, right))

def handle_BINARY_FLOOR_DIVIDE(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("(%s/%s)" % (a, b))

def handle_BINARY_OR(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("(%s|%s)" % (a, b))

def handle_COMPARE_OP(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("%s %s %s" % (a, dis.cmp_op[int(instr.arg)], b))

def handle_BINARY_SUBTRACT(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("(%s-%s)" % (a, b))

def handle_BINARY_LSHIFT(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("(%s<<%s)" % (a, b))

def handle_BINARY_RSHIFT(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("(%s>>%s)" % (a, b))

def handle_BINARY_ADD(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("(%s+%s)" % (a, b))
handle_INPLACE_ADD = handle_BINARY_ADD

def handle_BINARY_AND(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("(%s&%s)" % (a, b))

def handle_BINARY_MULTIPLY(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("(%s * %s)" % (a, b))

def handle_BINARY_MODULO(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("(%s %s %s)" % (a, "%", b))

def handle_SLICE_1(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("%s[%s:]" % (a, b))

def handle_SLICE_2(self, instr):
  a, b = self.get_and_clear_buf(2)
  self.buf.append("%s[:%s]" % (a, b))

def handle_SLICE_3(self, instr):
  a, b, c = self.get_and_clear_buf(3)
  self.buf.append("%s[%s:%s]" % (a, b, c))

def handle_STORE_SUBSCR(self, instr):
  value, obj, idx = self.get_and_clear_buf(3)
  self.emit("%s[%s] = %s" % (obj, idx, value))

def handle_GET_ITER(self, instr):
  [obj] = self.get_and_clear_buf(1)
  self.emit("for %s in %s:" % (instr.loop_var, obj))
  self.indent_level += 1
  self.handle_basic_block(instr.loop)
  self.indent_level -= 1

def handle_FOR_ITER(self, instr):
  self.basic_blocks.append(instr.fallthrough)

def handle_CALL_FUNCTION(self, instr):
  global pickles
  args = self.get_and_clear_buf(instr.arg + 1)
  func = args[0]
  args = args[1:]
  if func == "pickle.loads":
```

```
      pickles.append(args[0])
      self.buf.append('%s(%s)' % (func, ", ".join(args)))
      ###self.buf.append('%s("""%s""")' % (func, ", ".join(args)))
    else:
      self.buf.append("%s(%s)" % (func, ", ".join(args)))

  def handle_RETURN_VALUE(self, instr):
    [obj] = self.get_and_clear_buf(1)
    self.emit("return %s" % obj)

  def handle_POP_TOP(self, instr):
    [obj] = self.get_and_clear_buf(1)
    self.emit("%s # Pop from stack" % obj)

  def handle_SETUP_LOOP(self, instr):
    self.handle_basic_block(instr.true_block)
    self.handle_basic_block(instr.false_block)

  def handle_JUMP_IF_FALSE(self, instr):
    obj = self.buf[-1]
    buf_copy = self.buf[:]
    self.emit("if %s:" % obj)
    self.indent_level += 1
    self.handle_basic_block(instr.true_block)
    self.indent_level -= 1
    self.buf = buf_copy
    self.handle_basic_block(instr.false_block)

  def handle_JUMP_IF_TRUE(self, instr):
    obj = self.buf[-1]
    buf_copy = self.buf[:]
    self.emit("if not %s:" % obj)
    self.indent_level += 1
    self.handle_basic_block(instr.false_block)
    self.indent_level -= 1
    self.buf = buf_copy
    self.handle_basic_block(instr.true_block)

  def handle_RAISE_VARARGS(self, instr):
    self.emit("raise %s" % instr.arg)

  def handle_JUMP_FORWARD(self, instr):
    self.handle_basic_block(instr.fallthrough)

  def handle_JUMP_ABSOLUTE(self, instr):
    self.emit("continue")

def decompile(function):
  instructions = parse_bytecode(function.func_code)
  start_bblock = BasicBlockFinder(instructions).find_basic_blocks()
  body = Interpreter(start_bblock).evaluate()
  args = function.func_code.co_varnames[:function.func_code.co_argcount]
  args = ",".join(args)
  name = function.__name__
  header = "def %(name)s(%(args)s):\n" % { "name": name, "args": args }
  print(header + body)
  return header + body

def decompile_object(o):
  for i in dir(o):
    p = getattr(o, i)
    print("# %s %s" % (i, str(type(p))))
    if inspect.isfunction(p) or inspect.ismethod(p):
      decompile(p)
    elif isinstance(p, list):
      print("%s = %s" % (i, str(p)))
    print

pickles = []
mod = __import__("check")
decompile_object(mod)
# Pour decompiler le "main", on est oblige de lire le fichier pyc.
f = open("check.pyc", "rb")
f.read(8) # On ignore la version et la date.
c = marshal.load(f)
instructions = parse_bytecode(c)
start_bblock = BasicBlockFinder(instructions).find_basic_blocks()
print("# Main")
print(Interpreter(start_bblock, indent_level=0).evaluate())
f.close()
print

print("######################")
print("# FROM pickle.loads #")
print("######################")
for m in pickles:
  decompile_object(pickle.loads(m))
```

Listing 41 – Décompilateur de *bytecode python*

E Décompilation de *check.pyc*

```
# Bits <type 'classobj'>

# E <type 'function'>
def E(L):
  if not L.size == 32:
    L.size == 32 # Pop from stack
    raise 1
  L.size == 32 # Pop from stack
  table = ['31', '0', '1', '2', '3', '4', '3', '4', '5', '6', '7', '8', '
      7', '8',
  return L[table]

# ECB <type 'type'>

# F <type 'function'>
def F(R,k,r):
  RE = E(R)
  Z = Bits(0, 32)
  fk = subkey(k, r)
  s = RE ^ fk
  ri = ['(0, 0)'][1]
  ro = ['(0, 0)'][0]
  for n in range(8):
    nri = (ri+6)
    nro = (ro+4)
    x = s[ri:nri]
    i = x[(5, 0)].ival
    j = x[(4, 3, 2, 1)].ival
    Z[ro:nro] = Bits(S(n, ((i<<4)+j)), 4)[range(None,None,-1)]
    ri = nri
    ro = nro
    continue
  return P(Z)

# IP <type 'function'>
def IP(M):
  if not M.size == 64:
    M.size == 64 # Pop from stack
    raise 1
  M.size == 64 # Pop from stack
  table = ['57', '49', '41', '33', '25', '17', '9', '1', '59', '51', '43'
      , '35',
  return M[table]
```

```
# IPinv <type 'function'>
def IPinv(M):
  if not M.size == 64:
    M.size == 64 # Pop from stack
    raise 1
  M.size == 64 # Pop from stack
  table = ['39', '7', '47', '15', '55', '23', '63', '31', '38', '6', '46'
      , '14',
  return M[table]

# P <type 'function'>
def P(s):
  if not s.size == 32:
    s.size == 32 # Pop from stack
    raise 1
  s.size == 32 # Pop from stack
  table = ['15', '6', '19', '20', '28', '11', '27', '16', '0', '14', '22'
      , '25',
  return s[table]

# PC1 <type 'function'>
def PC1(K):
  table = ['56', '48', '40', '32', '24', '16', '8', '0', '57', '49', '41'
      , '33',
  return K[table]

# PC2 <type 'function'>
def PC2(K):
  if not K.size == 56:
    K.size == 56 # Pop from stack
    raise 1
  K.size == 56 # Pop from stack
  table = ['13', '16', '10', '23', '0', '4', '2', '27', '14', '5', '20',
      '9','22'
  return K[table]

# S <type 'function'>
def S(n,x):
  if 0 <= n:
    0 <= n # Pop from stack
    if not n < 8:
      n < 8 # Pop from stack
      raise 1
    n < 8 # Pop from stack
```

```
    if 0 <= x:
      0 <= x # Pop from stack
      if not x < 64:
        x < 64 # Pop from stack
        raise 1
      x < 64 # Pop from stack
      boxes = ["['14', '4', '13', '1', '2', '15', '11', '8', '3', '10',
          '6', '12'
      return Bits(boxes[n][x], 4)
    x # Pop from stack
  n # Pop from stack

# WhiteDES <type 'type'>

# __builtins__ <type 'dict'>

# __doc__ <type 'NoneType'>

# __file__ <type 'str'>

# __name__ <type 'str'>

# a2b_hex <type 'builtin_function_or_method'>

# b2a_hex <type 'builtin_function_or_method'>

# dec <type 'function'>
def dec(K,C):
  if not C.size == 64:
    C.size == 64 # Pop from stack
    raise 1
  C.size == 64 # Pop from stack
  k = PC1(K)
  blk = IP(C)
  L = blk[0:32]
  R = blk[32:64]
  for r in range(16)[range(None,None,-1)]:
    fout = F(R, k, r)
    L = L ^ fout
    tmp = L
    L = R
    R = tmp
    continue
  tmp = L
  L = R
  R = tmp
  M = Bits(0, 64)
  M[0:32] = L
  M[32:64] = R
  return IPinv(M)

# enc <type 'function'>
def enc(K,M):
  if not M.size == 64:
    M.size == 64 # Pop from stack
    raise 1
  M.size == 64 # Pop from stack
  k = PC1(K)
  blk = IP(M)
  L = blk[0:32]
  R = blk[32:64]
  for r in range(16):
    fout = F(R, k, r)
    L = L ^ fout
    tmp = L
    L = R
    R = tmp
    continue
  tmp = L
  L = R
  R = tmp
  C = Bits(0, 64)
  C[0:32] = L
  C[32:64] = R
  return IPinv(C)

# floor <type 'builtin_function_or_method'>

# log <type 'builtin_function_or_method'>

# pickle <type 'module'>

# random <type 'module'>

# struct <type 'module'>

# subkey <type 'function'>
def subkey(k,r):
  C = k[0:28]
  D = k[28:56]
  shifts = ['1', '1', '2', '2', '2', '2', '2', '2', '1', '2', '2', '2',
      '2', '2',
  s = sum(shifts[:(r+1)])
  C = ((C>>s)|(C<<(28-s)))
  D = ((D>>s)|(D<<(28-s)))
  return PC2((C/D))
```

```
# sys <type 'module'>

# Main
floor = __import__(math, fromlist=('floor', 'log'), level=-1).floor
log = __import__(math, fromlist=('floor', 'log'), level=-1).log
__import__(math, fromlist=('floor', 'log'), level=-1) # Pop from stack
b2a_hex = __import__(binascii, fromlist=('b2a_hex', 'a2b_hex'), level=-1)
    .b2a_hex
a2b_hex = __import__(binascii, fromlist=('b2a_hex', 'a2b_hex'), level=-1)
    .a2b_hex
__import__(binascii, fromlist=('b2a_hex', 'a2b_hex'), level=-1) # Pop
    from stack
sys = __import__(sys, fromlist=None, level=-1)
struct = __import__(struct, fromlist=None, level=-1)
random = __import__(random, fromlist=None, level=-1)
pickle = __import__(pickle, fromlist=None, level=-1)
Bits = 1#class Bits: ['\'<code object Bits at 0x100430a08, file "check.py
    ", line
ECB = 1#class ECB: ['\'<code object ECB at 0x100430c60, file "check.py",
    line 219
enc = '<code object enc at 0x100430d50, file "check.py", line 256>'
dec = '<code object dec at 0x100430e40, file "check.py", line 272>'
subkey = '<code object subkey at 0x100430eb8, file "check.py", line 288>'
F = '<code object F at 0x100430f30, file "check.py", line 297>'
IP = '<code object IP at 0x100433030, file "check.py", line 312>'
IPinv = '<code object IPinv at 0x1004330a8, file "check.py", line 324>'
PC1 = '<code object PC1 at 0x100433120, file "check.py", line 336>'
PC2 = '<code object PC2 at 0x100433198, file "check.py", line 347>'
E = '<code object E at 0x100433210, file "check.py", line 359>'
P = '<code object P at 0x100433288, file "check.py", line 371>'
S = '<code object S at 0x100433300, file "check.py", line 381>'
WhiteDES = 1#class WhiteDES: ['\'<code object WhiteDES at 0x100433648,
    file "chec
if __name__ == '__main__':
  __name__ == '__main__' # Pop from stack
  WT = pickle.loads(ccopy_reg\n_reconstructor\np0\n(ccheck\nWhiteDES\np1\
      nc__buil
  if len(sys.argv) == 1:
    len(sys.argv) == 1 # Pop from stack
    print('Usage: python check.pyc <key>')
    print('\n')
    print('    - key: a 64 bits hexlify-ed string')
    print('\n')
    print('Example: python check.pyc 0123456789abcdef')
    print('\n')
    return None
  len(sys.argv) == 1 # Pop from stack
  K = Bits(a2b_hex(sys.argv[1]), 64)
  if not K[range(7, 64, 8)] == 175:
    K[range(7, 64, 8)] == 175 # Pop from stack
    raise 1
  K[range(7, 64, 8)] == 175 # Pop from stack
  M = Bits(random.getrandbits(64), 64)
  if hex(WT._cipher(M, 1)) == hex(enc(K, M)):
    hex(WT._cipher(M, 1)) == hex(enc(K, M)) # Pop from stack
    exit(0) # Pop from stack
    return None
  hex(WT._cipher(M, 1)) == hex(enc(K, M)) # Pop from stack
  exit(1) # Pop from stack
__name__ == __main__ # Pop from stack

####################
# FROM pickle.loads #
####################
# FX <type 'instancemethod'>
def FX(self,v):
  res = Bits(0, 96)
  for b in range(96):
    res[b] = ((v&self.tM2[b]).hw() % 2)
    continue
  return res

# KT <type 'list'>
KT = [((8L, 27L, 6L, 28L, 1L, 30L, 12L, 19L, 2L, 23L, 5L, 25L, 15L, 16L,
    10L, 21L

# __class__ <type 'type'>

# __delattr__ <type 'method-wrapper'>

# __dict__ <type 'dict'>

# __doc__ <type 'NoneType'>

# __getattribute__ <type 'method-wrapper'>

# __hash__ <type 'method-wrapper'>

# __init__ <type 'instancemethod'>
def __init__(self,KT,tM1,tM2,tM3):
  return None

# __module__ <type 'str'>

# __new__ <type 'builtin_function_or_method'>
```

```
# __reduce__ <type 'builtin_function_or_method'>

# __reduce_ex__ <type 'builtin_function_or_method'>

# __repr__ <type 'method-wrapper'>

# __setattr__ <type 'method-wrapper'>

# __str__ <type 'method-wrapper'>

# __weakref__ <type 'NoneType'>

# _cipher <type 'instancemethod'>
def _cipher(self,M,d):
  if not M.size == 64:
    M.size == 64 # Pop from stack
    raise 1
  M.size == 64 # Pop from stack
  if d == 1:
    d == 1 # Pop from stack
    blk = M[self.tM1]
    for r in range(16):
      t = 0
      for n in range(12):
        nt = (t+8)
        blk[t:nt] = self.KT[r][n][blk[t:nt].ival]
        t = nt
        continue
      blk = self.FX(blk)
      continue
    return blk[self.tM3]
    if d == -1:
      d == -1 # Pop from stack
      raise 1
      return None
    d == -1 # Pop from stack
  d == 1 # Pop from stack

# dec <type 'instancemethod'>
def dec(self,C):
  tmp = n
  n = ['divmod(len(C), 8)'][1]
  p = ['divmod(letmp(C), 8)'][0]
  if not p == 0:
    p == 0 # Pop from stack
    raise 1
  p == 0 # Pop from stack
  M = []
  for b in range(n):
    M.append(hex(self._cipher(Bits(C[0:8]), -1))) # Pop from stack
    C = C[8:]
    continue
  if not len(C) == 0:
    len(C) == 0 # Pop from stack
    raise 1
  len(C) == 0 # Pop from stack
  return .join(M)

# enc <type 'instancemethod'>
def enc(self,M):
  tmp = n
  n = ['divmod(len(M), 8)'][1]
  p = ['divmod(letmp(M), 8)'][0]
  if p > 0:
    p > 0 # Pop from stack
    M = (M+(chr((8-p)) * (8-p)))
    n = (n+1)
    C = []
    for b in range(n):
      C.append(hex(self._cipher(Bits(M[0:8]), 1))) # Pop from stack
      M = M[8:]
      continue
    if not len(M) == 0:
      len(M) == 0 # Pop from stack
      raise 1
    len(M) == 0 # Pop from stack
    return .join(C)
  p > 0 # Pop from stack

# pad <type 'instancemethod'>
def pad(self,M):
  m = (%s % M)
  tmp = n
  n = ['divmod(len(m), 8)'][1]
  p = ['divmod(letmp(m), 8)'][0]
  if p > 0:
    p > 0 # Pop from stack
    m = (m+(chr((8-p)) * (8-p)))
    n = (n+1)
    return m
  p > 0 # Pop from stack

# tM1 <type 'list'>
tM1 = [6, 56, 48, 40, 32, 24, 57, 49, 32, 24, 16, 8, 0, 58, 41, 33, 0,
    58, 50, 42
```

```
# tM2 <type 'list'>
tM2 = [247588007857104202477495910 4L, 134217792L, 1152L, 34359754752L,
    1099511660
```

```
# tM3 <type 'list'>
tM3 = [31, 16, 63, 32, 75, 48, 91, 0, 30, 71, 62, 79, 74, 87, 90, 95, 23,
     70, 55,
```

Listing 42 – Résultat de la décompilation de *check.pyc*

F Code source de *reverse.py*

```
import random
import sys

from tables import *

def bitLenCount(int_type):
   count = 0
   while (int_type):
     count += (int_type & 1)
     int_type >>= 1
   return count

def bitPos(n):
   pos = []
   i = 0
   while (n):
     if n % 2 == 1:
       pos = pos + [i]
     i = i + 1
     n = n >> 1
   return pos

def tabToInt(v):
  s = 0
  for b in range(len(v) - 1, -1, -1):
    s = s * 2 + v[b]
  return s

def intToTab(i, s):
  r = []
  for j in range(s):
    if i % 2 == 1:
      r = r + [1]
    else:
      r = r + [0]
    i = i / 2
  return r

def intToTab2(i, s):
  r = []
  for j in range(s):
    if i % 2 == 1:
      r = [1] + r
    else:
      r = [0] + r
    i = i / 2
  return r

def FX(v):
  res = [0L] * 96
  s = tabToInt(v)
  for b in range(96L):
    res[b] = bitLenCount(s & tM2[b]) % 2L
  return res

def _cipher(M, d):
    if not len(M) == 64:
        raise 1
    if d == 1:
        blk = [M[i] for i in tM1]
        for r in range(16):
            t = 0L
            for n in range(12):
                nt = (t+8)
                l = KT[r][n]
                blk[t:nt] = intToTab(l[tabToInt(blk[t:nt])], nt-t)
                t = nt
                continue
            blk = FX(blk)
            continue
        return [blk[i] for i in tM3]
    if d == -1:
        raise 1
    return None

def laleatoire(s):
  return [random.sample([0, 1], 1)[0] for i in range(s)]

def l2s(l):
  return "".join([str(i) for i in l])

def product(*args, **kwds):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
    pools = map(tuple, args) * kwds.get('repeat', 1)
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
```

```
    for prod in result:
        yield tuple(prod)

# Identification des positions des bits de Li dans blki.
blk2li = [0] * 32
zero64 = [0] * 64
for i in range(32):
  zero64[i] = 1
  input = [zero64[j] for j in IPinv]
  blk = [input[j] for j in tM1]
  bits_set_in_blk = [j for j in range(len(blk)) if blk[j] == 1]
  blk2li[i] = bits_set_in_blk[0]
  zero64[i] = 0
# Identification des positions des bits de Ri dans blki.
candidates = dict([(i, list(range(96))) for i in range(96)])
for i in range(400):
  ablk0 = laleatoire(96)
  ablk1 = ablk0[:]
  t = 0L
  for n in range(12):
      nt = t + 8
      l = KT[0][n]
      ablk1[t:nt] = intToTab(l[tabToInt(ablk1[t:nt])], nt - t)
      t = nt
  ablk1 = FX(ablk1)
  # Dans ces boucles, si le j-eme bit de ablk1 n'est pas egale
  # au k-eme bit de ablk0, alors k n'est pas un candidat pour
  # la generation du j-eme bit de ablk1.
  for j in range(96):
    for k in candidates[j][:]:
      if ablk0[k] != ablk1[j] and k in candidates[j]:
        candidates[j].remove(k)
blk2ri = [0] * 32
for i in range(len(blk2li)):
  blk2ri[i] = candidates[blk2li[i]][0]
# Recuperation des indices permettant d'obtenir f(Ri,Ki+1) depuis tM2.
blk2pfi = [0] * 32
for j in range(len(blk2ri)):
  i = blk2ri[j]
  # A ce niveau, bitLenCount(tM2[i]) == 2, sinon c'est pas un bit de Ri.
  cont = True
  pos = bitPos(tM2[i])
  while cont:
    input = laleatoire(64)
    blk = [input[n] for n in tM1]
    l0 = [blk[n] for n in blk2li]
    # Iteration de l'etage 1.
    r = 0
    t = 0L
    for n in range(12):
      nt = t + 8
      l = KT[0][n]
      blk[t:nt] = intToTab(l[tabToInt(blk[t:nt])], nt - t)
      t = nt
    cont = blk[pos[0]] == blk[pos[1]]
  if blk[pos[0]] == l0[j]:
    blk2pfi[j] = pos[1]
  else:
    blk2pfi[j] = pos[0]
# Recuperation des clefs intermediaires.
Kq = [None] * 16
for q in range(16):
  # 48 premiers bits de Kq
  candidatesKq = dict([(i, map(list, list(product(range(2), repeat=6))))
      for i in range(8)])
  #candidatesKqb1a6 = map(list, list(product(range(2), repeat=6)))
  last = sum([len(candidatesKq[i]) for i in range(8)])
  while last > 8:
    # Pour un nombre, recuperation de L0, R0, L1 et R1.
    # Il faut que les Li et Ri soient coherents, on part donc de input.
    input = laleatoire(64)
    blk0 = [input[i] for i in tM1]
    # Iterations des etages 1 a q.
    for x in range(q + 1):
      blk1 = blk0[:]
      t = 0L
      for n in range(12):
        nt = t + 8
        l = KT[x][n]
        blk1[t:nt] = intToTab(l[tabToInt(blk1[t:nt])], nt-t)
        t = nt
      blk1_before_xor = blk1[:]
      blk1 = FX(blk1)
      if x < q:
        blk0 = blk1
    l0 = [blk0[i] for i in blk2li]
    r0 = [blk0[i] for i in blk2ri]
    l1 = [blk1[i] for i in blk2li]
    r1 = [blk1[i] for i in blk2ri]
    fr0k1 = [int(r1[i] ^ l0[i]) for i in range(32)]
    # Calcul de f(R,K) pour chacun des candidats et comparaison avec
        fr0k1.
    r0e = [r0[i] for i in E]
    for ci in candidatesKq:
      ii = ci * 6
      io = ci * 4
```

```
      for k in candidatesKq[ci][:]:
        r0exk = [int(r0e[ii + i] ^ k[i]) for i in range(6)]
        m = (r0exk[0] << 1) + r0exk[5]
        n = (r0exk[1] << 3) + (r0exk[2] << 2) + (r0exk[3] << 1) + r0exk
            [4]
        r0exks = S[ci][(m << 4) + n]
        r0exks = intToTab2(r0exks, 4)
        r0exkss = ([2] * (io)) + r0exks + ([2] * (32 - 4 - io))
        r0exkssp = [r0exkss[i] for i in P]
        for i in range(32):
          # Pour le bit i, si dans r0exkssp la valeur est 2, c'est que ce
              bloc
          # n'engendre pas ce bit. On ignore ce test pour l'instant.
          if r0exkssp[i] != 2 and r0exkssp[i] != fr0k1[i]:
            candidatesKq[ci].remove(k)
            last = last - 1
            break
    Kqq = []
    for i in range(8):
      Kqq = Kqq + candidatesKq[i][0]
    Kq[q] = Kqq

f = open("out.Kqq", "wb")
import pickle
pickle.dump(Kq, f)
f.close()
```

Listing 43 – Code source de *reverse.py*

G Code source de *subkey2key.py*

```
import pickle
from binascii import b2a_hex, a2b_hex

def tabToInt(v):
  s = 0
  for b in range(len(v) - 1, -1, -1):
    s = s * 2 + v[b]
  return s

f = open("out.Kqq", "rb")
Kqq = pickle.load(f)
f.close()

PC1 = [56, 48, 40, 32, 24, 16, 8, 0, 57, 49, 41, 33, 25, 17, 9,
1, 58, 50, 42, 34, 26, 18, 10, 2, 59, 51, 43, 35, 62, 54, 46,
38, 30, 22, 14, 6, 61, 53, 45, 37, 29, 21, 13, 5, 60, 52, 44,
36, 28, 20, 12, 4, 27, 19, 11, 3]

PC2 = [13, 16, 10, 23, 0, 4, 2, 27, 14, 5, 20, 9, 22, 18, 11,
3, 25, 7, 15, 6, 26, 19, 12, 1, 40, 51, 30, 36, 46, 54, 29,
39, 50, 44, 32, 47, 43, 48, 38, 55, 33, 52, 45, 41, 49, 35, 28,
31]

shifts = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

PC2Inv = [None] * 56
for i in range(len(PC2)):
  PC2Inv[PC2[i]] = i

PC1Inv = [None] * 64
for i in range(len(PC1)):
  PC1Inv[PC1[i]] = i
```

```
r = 15
K = [None] * 56
while r >= 0:
  Kqp = [None] * 56
  for i in range(56):
    if not PC2Inv[i] == None:
      Kqp[i] = Kqq[r][PC2Inv[i]]
  Test = [Kqp[i] for i in PC2]
  if Test != Kqq[r]:
    print "ERROR!"
  for i in range(len(Kqp)):
    if not Kqp[i] == None:
      if not K[i] == None:
        if Kqp[i] != K[i]:
          print "ERROR!"
      else:
        K[i] = Kqp[i]
  G = K[:]
  G[:28] = K[(28-shifts[r]):28] + K[:(28-shifts[r])]
  G[28:56] = K[(56-shifts[r]):56] + K[28:(56-shifts[r])]
  K = G
  r = r - 1
G = [None] * 64
for i in range(64):
  if not PC1Inv[i] == None:
    G[i] = K[PC1Inv[i]]
parity = [1, 1, 1, 1, 0, 1, 0, 1]
for i in range(8):
  G[i * 8 + 7] = parity[i]
G = G[::-1]
print "%16x" % tabToInt(G)
```

Listing 44 – Code source de *subkey2key.py*

H Code source de *load_rom.py*

```
import sys, struct

# memory of the usb device
memory = []
vector = {}

vdata  = {}

def usage():
    print 'usage: %s memory [files]' % sys.argv[0]
    sys.exit(1)

def write_rom():
    print '[*] Dumping memory to file %s' % sys.argv[1]
    f = open(sys.argv[1], 'wb')
    f.write(''.join(map(chr, memory)))
    f.close()

def handle_opcode(opcode, data):
    if opcode == 0x2:
        count = len(data) - 1
        vec   = ord(data[0])
        print '+ Asking BIOS to allocate %d bytes on vector %d' % (count,
             vec)
        print '  Copying data to this vector %d' % vec
        vector[vec] = len(memory)
        memory.extend(map(ord, data[1:2+count]))
        vdata[vec] = data[1:2+count]
    elif opcode == 0x6:
        vec = ord(data[0])
        print '+ Execute instruction at vector %d' % vec
        print '  Address in vector %d is %#.4x' % (vec, vector[vec])
    elif opcode == 0x3:
        vcode = ''
        vec = ord(data[0])
        print '+ Opcode 3 vec %d' % vec
        for i in xrange(1, len(data),2):
            print '%.2x%.2x' % tuple(map(ord,(data[i+1], data[i])))
            offset = ord(data[i+0]) + (ord(data[i+1]) << 8)
            vcode = vcode + vdata[vec][offset:offset+2]
        f = open('prout', 'w')
        f.write(vcode)
        f.close()
    else:
        print '- Opcode not yet implemented'

def parse_scans(stream):
    i = 0
    while i < len(stream):
        if ord(stream[i]) == 0xb6 and ord(stream[i+1]) == 0xc3:
            dataLen = ord(stream[i+2]) + (ord(stream[i+3]) << 8)
            #dataLen = struct.unpack('h', stream[i+2:i+4])[0]
            opcode  = ord(stream[i+4])
            print opcode
            data    = stream[i+5:i+5+dataLen]
            print data.encode('hex')
            handle_opcode(opcode, data)
            i       += 5 + dataLen
        else:
            print 'Cochonneries'
            sys.exit(1)

if __name__ == '__main__':
    if len(sys.argv) < 2:
        usage()

    for i in xrange(2,len(sys.argv)):
        print '[*] Handling rom file %s' % sys.argv[i]
        f = open(sys.argv[i], 'rb')
        parse_scans(f.read())
        f.close()

    write_rom()
```

Listing 45 – Code source de *load_rom.py*

I Code source de *interpreter.c*

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "data.h"

typedef unsigned int uint;

uint key_valid;
uint fatal;
uint stop;
uint r4 = -1;

uint key_layer1;
uint key_layer2;
uint key_layer3;

//#define DEBUG 0
//#define DUMP 1

unsigned char* layer;

#define RAMSIZE (1024 * 65)

#define GET_SHORT_FROM_RAM(a) (uint)((ram[(a) + 1] << 8) | ram[a])
#define SET_SHORT_INTO_RAM(a, v) \
  do { \
    uint ca = (a); \
    uint cv = (v); \
    ram[(ca)] = (cv) & 0xFF; \
    ram[(ca) + 1] = ((cv) >> 8) & 0xFF; \
  } while (0)

unsigned char* ram;

uint pow2[] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048};

void create(void) {
  ram = malloc(RAMSIZE);
}

void init(void) {
  memset(ram, 0, RAMSIZE);
}

void destroy(void) {
  free(ram);
}

uint loc_1268h(uint nb) {
  uint used = ram[0x111e];
  uint index = GET_SHORT_FROM_RAM(0x111c);
  uint result = 0;
  if (8 - used >= nb) {
    result = (layer[index] >> (8 - used - nb)) & (pow2[nb] - 1);
  } else {
    result = layer[index] & (pow2[8 - used] - 1);
    nb = nb - (8 - used);
    used = 0;
    index = index + 1;
    while (nb >= 8) {
      result = (result << 8) | layer[index];
      index = index + 1;
      nb = nb - 8;
    }
    if (nb > 0) {
      result = (result << nb) | ((layer[index] >> (8 - nb)) & (pow2[nb] -
          1));
    }
  }
  used = used + nb;
  ram[0x111e] = used & 0x7;
  index = index + (used >> 3);
  SET_SHORT_INTO_RAM(0x111c, index);
#ifdef DEBUG
  printf("%04x ", result);
#endif
  return result;
}

#define Z 1
#define C 2
#define O 4
#define S 8
```

```
uint test_cccc_masks[]  = {Z, Z, C, C, S, S, O, O, Z+C, Z+C};
uint test_cccc_values[] = {Z, 0, C, 0, S, 0, O, 0, 0,   Z+C};

uint test_cccc(uint cccc, uint flags) { // TODO: évrifier
  if (cccc <= 9) {
    return (test_cccc_masks[cccc] & flags) == test_cccc_values[cccc];
  } else if (cccc == 0xA) {
    return ( (((flags & (O+S)) == (O+S)) || ((flags & (O+S)) == 0))  &&
        ((flags & Z) == 0) );
  } else if (cccc == 0xB) {
    return ((flags & (O+S)) == (O+S)) || (((flags & (O+S)) == 0));
  } else if (cccc == 0xC) {
    return ((flags & (O+S)) == O) || (((flags & (O+S)) == S));
  } else if (cccc == 0xD) {
    return (((flags & (O+S)) == O) || (((flags & (O+S)) == S))) && ((
        flags & Z) == Z);
  } else if (cccc == 0xF) {
    return 1;
  } else {
    printf("Bad cccc\n");
    fatal = 1;
  }
  return 0;
}

uint loc_11a6h(uint r0, uint r1, uint r2) {
  if (r1 < 2) {
    ram[r0] = r2 & 0xFF;
    if (r1 == 1) {
      ram[r0 + 1] = (r2 >> 8) & 0xFF;
    }
    return 0;
  } else if (r1 == 4) {
    return ram[r0] | (ram[r0 + 1] << 8);
  } else {
    return ram[r0];
  }
}

uint loc_1166h(uint r0, uint r1, uint r2) {
  if ((r1 != 0) && (r1 != 2) && ((r0 & 0xF) == 0xF)) {
    r1 = r1 >> 1;
    if (r1 != 0) {
      //TODO
      /*SET_SHORT_INTO_RAM(0x1072, loc_11a6h(r0, r1, r2));
      uint r0 = loc_11a6h(r0 + 1, r1, r2);
      r0 = (r0 << 8) & ram[0x1072];
      return r0;*/
      printf("loc_1166h: error p2: %x %x %x\n", r0, r1, r2);
      fatal = 1;
    }
    // TODO
    printf("loc_1166h: error p1: %x %x %x\n", r0, r1, r2);
    fatal = 1;
  }
  return loc_11a6h(r0, r1, r2);
}

uint loc_1162h(uint r0, uint r1, uint r2) {
  return loc_1166h(r0, (r1 + 1) * 2, r2);
}

uint rol(int value, int shift) {
  return ((value << shift) | (value >> (16-shift))) & 0xffff;
}

int ror(int value, int shift) {
  return ((value >> shift) | (value << (16-shift))) & 0xffff;
}

uint loc_184ch(uint r0, uint r1, uint r2, uint r3, uint r4, uint r6) {
  uint r5 = (((((r2 << 6) + r2) << 4) + r2) & 0xFFFF) ^ 0x464d;
  r2 = r0 ^ 0x6c38;
  r0 = (r0 + r1) & 0xFFFF;
  r1 = (r1 + 2) & 0xFFFF;
  uint r7;
  uint r8;
  do {
    r7 = r6;
    r5 = rol(r5, 1) & 0xFFFF; // TODO: juste un édcalage suffit ?
    r2 = (ror(r2, 2) + r5) & 0xFFFF; // TODO: juste un édcalage suffit ?
    // r5 = (r5 << 1) & 0xFFFF; // TODO: juste un édcalage suffit ?
    // r2 = ((r2 >> 2) + r5) & 0xFFFF; // TODO: juste un édcalage suffit
        ?
    r5 = (r5 + 2) & 0xFFFF;
    r6 = r2 ^ r5;
    r8 = (r6 >> 8) & 0xFF;
    r6 = (r6 & 0xFF) ^ r8;
    r1 = (r1 - 1) & 0xFFFF;
  } while (r1 > 0);
  r6 = ((r6 << 8) & 0xFFFF) | r7;
  r1 = r3;
  r2 = r4 ^ r6;
  r0 = loc_1166h(r0, r1, r2) ^ r6;
  r3 = r3 & 4;
  if (r3 == 0) {
```

```
    r0 = r0 & 0xFF;
  }
  return r0;
}

uint loc_1848h(uint r0, uint r1, uint r2, uint r3, uint r4, uint r6) {
  r3 = (r3 + 1) << 1;
  return loc_184ch(r0, r1, r2, r3, r4, r6);
}

//void loc_13c4h(uint r4, int r4_initialised, uint r6) {
void loc_13c4h(uint r6) {
  uint word_112eh = loc_1268h(1);
  SET_SHORT_INTO_RAM(0x112e, word_112eh);
  uint word_1130h = loc_1268h(2);
  SET_SHORT_INTO_RAM(0x1130, word_1130h);
  if (word_1130h == 1) { // 14c2
    if (word_112eh == 0) {
      SET_SHORT_INTO_RAM(0x1126, loc_1268h(8));
    } else {
      SET_SHORT_INTO_RAM(0x1126, loc_1268h(16));
    }
  } else if (word_1130h == 0) { // 14a8
    uint reg = loc_1268h(4);
    SET_SHORT_INTO_RAM(0x1128, reg);
    SET_SHORT_INTO_RAM(0x1126, GET_SHORT_FROM_RAM(0x10fe + (reg * 2)));
  } else if (word_1130h == 2) { // 1430
    uint word_1132h = loc_1268h(2);
    SET_SHORT_INTO_RAM(0x1132, word_1132h);
    if (word_1132h == 1) { // 149a
      uint result = loc_1268h(16);
      SET_SHORT_INTO_RAM(0x112a, result);
      result = loc_1162h(result, word_112eh, -1);
      SET_SHORT_INTO_RAM(0x1126, result);
    } else if (word_1132h == 0) { // 1482
      uint word_1128h = loc_1268h(4);
      SET_SHORT_INTO_RAM(0x1128, word_1128h);
      uint word_112ah = GET_SHORT_FROM_RAM(0x10fe + (word_1128h * 2));
      SET_SHORT_INTO_RAM(0x112a, word_112ah);
      uint r1 = word_112eh;
      uint r0 = word_112ah;
      uint word_1126h = loc_1162h(r0, r1, -1);
      SET_SHORT_INTO_RAM(0x1126, word_1126h);
    } else if (word_1132h == 3) { // 1446
      uint r1 = word_112eh;
      uint r0 = GET_SHORT_FROM_RAM(0x112a);
      uint word_1126h = loc_1162h(r0, r1, -1);
      SET_SHORT_INTO_RAM(0x1126, word_1126h);
    } else { // 1454, word_1132h == 2
      //printf("Through bug in loc_13c4h of c7df of loc_13c4h\n");
      uint word_112ah = loc_1268h(16);
      SET_SHORT_INTO_RAM(0x112a, word_112ah);
      uint word_1128h = loc_1268h(4);
      SET_SHORT_INTO_RAM(0x1128, word_1128h);
      SET_SHORT_INTO_RAM(0x112a, GET_SHORT_FROM_RAM(0x112a) +
          GET_SHORT_FROM_RAM(0x10fe + (word_1128h * 2)));
      uint r1 = word_112eh;
      uint r0 = GET_SHORT_FROM_RAM(0x112a);
      uint word_1126h = loc_1162h(r0, r1, -1);
      SET_SHORT_INTO_RAM(0x1126, word_1126h);
    }
  } else { // 13ee, word_1130h == 3
    //printf("Through bug in loc_13c4h of r4\n");
    // if (r4_initialised == 0) {
    if (r4 == -1) {
      // TODO
      printf("loc_13c4h: error p2: %x %x\n", ram[0x112e], ram[0x1130]);
      fatal = 1;
    }
    uint word_112ah = loc_1268h(16);
    SET_SHORT_INTO_RAM(0x112a, word_112ah);
    uint word_1128h = loc_1268h(4);
    SET_SHORT_INTO_RAM(0x1128, word_1128h);
    uint word_112ch = loc_1268h(6);
    SET_SHORT_INTO_RAM(0x112c, word_112ch);
    uint r2 = word_112ch;
    uint r3 = word_112eh;
    uint r1 = GET_SHORT_FROM_RAM(0x10fe + (word_1128h * 2));
    uint r0 = word_112ah;
    uint word_1126h = loc_1848h(r0, r1, r2, r3, r4, r6);
    SET_SHORT_INTO_RAM(0x1126, word_1126h);
  }
}

inline void sub_1346h(void) {
  memcpy(&ram[0x1134], &ram[0x1126], 12);
}

void sub_135eh(uint r6) {
  uint word_113eh = GET_SHORT_FROM_RAM(0x113e);
  if (word_113eh == 0) { // 1374
    uint word_113ch = GET_SHORT_FROM_RAM(0x113c);
    uint r9 = GET_SHORT_FROM_RAM(0x1136);
    ram[0x10fe + r9 * 2] = ram[0x1126];
    if (word_113ch != 0) {
      ram[0x10fe + r9 * 2] = ram[0x1126];
```

```
      ram[0x10fe + r9 * 2 + 1] = ram[0x1127];
    }
  } else if (word_113eh == 2) { // 13b2
    uint r0 = GET_SHORT_FROM_RAM(0x1138);
    uint r2 = GET_SHORT_FROM_RAM(0x1126);
    uint r1 = ram[0x113c];
    loc_1166h(r0, r1, r2);
  } else if (word_113eh == 3) { // 1392
    uint r9 = GET_SHORT_FROM_RAM(0x1136) * 2;
    uint r3 = ram[0x113c];
    /* uint */ r4 = GET_SHORT_FROM_RAM(0x1126); // TODO: pas éé protg par
        PUSH/POP
    uint r2 = GET_SHORT_FROM_RAM(0x113a);
    uint r1 = GET_SHORT_FROM_RAM(0x10fe + r9);
    uint r0 = GET_SHORT_FROM_RAM(0x1138);
    loc_184ch(r0, r1, r2, r3, r4, r6);
  } else { // 1372
    //TODO
    /*return;*/
    printf("sub_135eh: error p1\n");
    fatal = 1;
  }
}

void loc_1814h(uint r6, uint r14) {
  if ((r6 != 0) && ((r14 & 0x2) == 0)) {
    uint r0 = ram[0xc000] & 0xF;
    uint r12 = GET_SHORT_FROM_RAM(0x1144);
    if (r14 == 0) {
      r0 = r0 << 4;
      r12 = r12 & 0x0F;
    } else {
      r12 = r12 & 0xF0;
    }
    SET_SHORT_INTO_RAM(0x1144, r12 | r0);
  }
}

void loc_14dah(void) {
  //TODO Verifier que c'est bien inutil.
  //memcpy(&ram[0x1120], &ram[0x111a], 6);
  uint r14 = loc_1268h(1);
  uint r11 = loc_1268h(8);
  if (r11 == 0xFF) {
    stop = 1;
    return;
  }
  uint r12 = GET_SHORT_FROM_RAM(0x1144);
  if (r14 == 0) {
    r11 = r11 >> 4;
    r12 = r12 >> 4;
  }
  if (!test_cccc(r11 & 0xF, r12 & 0xF)) {
    r14 = r14 + 2;
  }
  r11 = 0;
  uint r13 = loc_1268h(3);
  uint r6 = loc_1268h(1);
  SET_SHORT_INTO_RAM(0x1126, r11);
  if (r13 == 4) { // 160ah
    // loc_13c4h(r4, 1, r6); // Fernand: Pourquoi on indique que c'est
        éinitialis ?
    loc_13c4h(r6);
    /*uint*/ r4 = ram[0x1130];
    uint r3 = GET_SHORT_FROM_RAM(0x1128);
    if ((r4 == r11) && (r3 == 0xF)) {
      r11 = r11 + 1;
    }
    if ((r3 == 0xE) && ((r4 == 0x3) || ((r4 == 2) && (GET_SHORT_FROM_RAM
        (0x1132) == 0)))) { // 163ch
      r12 = r14 & 0x2;
      if (r12 == 0) {
        SET_SHORT_INTO_RAM(0x111a, GET_SHORT_FROM_RAM(0x111a) - 2);
        SET_SHORT_INTO_RAM(0x112a, GET_SHORT_FROM_RAM(0x112a) - 2);
      }
    } // 164ch
    sub_1346h();
    loc_13c4h(r6);
    r12 = r14 & 0x2;
    if (r12 != 0) {
      return;
    }
    if (GET_SHORT_FROM_RAM(0x1128) != 0xE) {
      // 15aeh
      if ((r14 & 0x2) == 0) {
        sub_135eh(r6);
      }
    } else {
      // 1668h - TODO
      printf("loc_14dah: error p2 %x %x\n", r13, r6);
      fatal = 1;
    }
  } else if (r13 < 2) { // 1574h
    //loc_13c4h(r4, 1, r6); // Fernand: Pourquoi on indique que c'est
        éinitialis ?
    loc_13c4h(r6);
```

```
    sub_1346h();
    //loc_13c4h(r4, 1, r6); // Fernand: Pourquoi on indique que c'est
        éinitialis ?
    loc_13c4h(r6);
    uint r0;
    if (r13 != r11) {
      r0 = GET_SHORT_FROM_RAM(0x1134) | GET_SHORT_FROM_RAM(0x1126);
    } else {
      r0 = GET_SHORT_FROM_RAM(0x1134) & GET_SHORT_FROM_RAM(0x1126);
    }
    // Update flags
    if (r0 & 0x8000) {
      ram[0xc000] = ram[0xc000] | S;
    } else {
      ram[0xc000] = ram[0xc000] & (~S);
    }
    if (r0) {
      ram[0xc000] = ram[0xc000] & (~Z);
    } else {
      ram[0xc000] = ram[0xc000] | Z;
    }
    // 1592h
    r12 = GET_SHORT_FROM_RAM(0x1144) & 0xFFEE;
    loc_1814h(r6, r14);
    uint r2 = (GET_SHORT_FROM_RAM(0x1144) & 0xFF11) | r12;
    SET_SHORT_INTO_RAM(0x1144, r2);
    SET_SHORT_INTO_RAM(0x1126, r0);
    if ((r14 & 0x2) == 0) {
      sub_135eh(r6);
    }
  } else if (r13 == 2) { // 15beh
    //loc_13c4h(r4, 1, r6); // Fernand: Pourquoi on indique que c'est
        éinitialis ?
    loc_13c4h(r6);
    sub_1346h();
    ram[0x1126] = (~ram[0x1126]) & 0xFF;
    ram[0x1127] = (~ram[0x1127]) & 0xFF;
    uint r0 = GET_SHORT_FROM_RAM(0x1126);
    // Update flags
    if (r0 & 0x8000) {
      ram[0xc000] = ram[0xc000] | S;
    } else {
      ram[0xc000] = ram[0xc000] & (~S);
    }
    if (r0) {
      ram[0xc000] = ram[0xc000] & (~Z);
    } else {
      ram[0xc000] = ram[0xc000] | Z;
    }
    loc_1814h(r6, r14);
    // 15aeh
    if ((r14 & 0x2) == 0) {
      sub_135eh(r6);
    }
  } else if (r13 == 3) { // 15d0h
    uint r10 = loc_1268h(1);
    r12 = loc_1268h(8);
    //loc_13c4h(r4, 1, r6); // Fernand: Pourquoi on indique que c'est
        éinitialis ?
    loc_13c4h(r6);
    sub_1346h();
    uint r0 = GET_SHORT_FROM_RAM(0x1126);
    uint nc = 0;
    if (r10 != r11) {
      r0 = r0 >> (r12 - 1);
      nc = r0 & 0x1;
      r0 = r0 >> 1;
    } else {
      r0 = r0 << r12;
      nc = r0 & 0x10000;
    }
    // Update flags
    if (r0 & 0x8000) {
      ram[0xc000] = ram[0xc000] | S;
    } else {
      ram[0xc000] = ram[0xc000] & (~S);
    }
    if (nc != 0x0) {
      ram[0xc000] = ram[0xc000] | C;
    } else {
      ram[0xc000] = ram[0xc000] & (~C);
    }
    r0 = r0 & 0xFFFF;
    if (r0) {
      ram[0xc000] = ram[0xc000] & (~Z);
    } else {
      ram[0xc000] = ram[0xc000] | Z;
    }
    loc_1814h(r6, r14);
    // 15aah
    SET_SHORT_INTO_RAM(0x1126, r0);
    if ((r14 & 0x2) == 0) {
      sub_135eh(r6);
    }
  } else { // 1702h
    ram[0xc000] = ram[0xc000] | C; // TODO: On a besoin de faire stc ?
```

```
    r12 = loc_1268h(6);
    uint r10 = loc_1268h(3);
    //TODO Test correct ?
    if (r12 == 0) {
      //loc_13c4h(r4, 1, r6); // Fernand: Pourquoi on indique que c'est
          éinitialis ?
      loc_13c4h(r6);
      if (GET_SHORT_FROM_RAM(0x1130) != 1) { // 1726h
        // TODO
        printf("loc_14dah: error p6: %x %x\n", r13, r6);
        fatal = 1;
      }
    } // 173ah
    r14 = r14 & 0x2;
    if (r14) {
      return;
    }
    if (r13 != 6) {
loc_1748h:
//TODO Test correct ?
      if (r12) {
        r11 = r12 & 0x20;
        r12 = r12 & 0x1F;
        if (r11 == 0) { // 175ah
          uint r3 = GET_SHORT_FROM_RAM(0x111c) + r12;
          /*uint*/ r4 = GET_SHORT_FROM_RAM(0x111e) + r10;
          SET_SHORT_INTO_RAM(0x111c, r3 + ((r4 >> 3) & 0x1FFF));
          SET_SHORT_INTO_RAM(0x111e, r4 & 0x7);
          return;
        } else { // 1780h
          uint r0 = GET_SHORT_FROM_RAM(0x111c) - r12;
          if (GET_SHORT_FROM_RAM(0x111e) < r10) {
            r0 = r0 - 1;
            SET_SHORT_INTO_RAM(0x111e, GET_SHORT_FROM_RAM(0x111e) + 8);
          } // 1796h
          SET_SHORT_INTO_RAM(0x111c, r0);
          SET_SHORT_INTO_RAM(0x111e, GET_SHORT_FROM_RAM(0x111e) - r10);
          return;
        }
      } // 17a0h
      ram[0x111c] = ram[0x1126];
      ram[0x111d] = ram[0x1127];
      SET_SHORT_INTO_RAM(0x111e, r10);
      return;
    } // 17aeh

    uint r2 = GET_SHORT_FROM_RAM(0x1140);
    if (r2 == 0) {
      uint r5 = (GET_SHORT_FROM_RAM(0x111a) - 2) & 0xFFFF;
      r2 = GET_SHORT_FROM_RAM(0x111c);
      uint r1 = 1;
      uint r0 = r5;
      loc_1166h(r0, r1, r2);
      r5 = r5 - 2;
      SET_SHORT_INTO_RAM(0x111a, r5);
      r2 = GET_SHORT_FROM_RAM(0x111e);
      r1 = 1;
      r0 = r5;
      loc_1166h(r0, r1, r2);
      goto loc_1748h;
    } else { // 177eh
      // TODO
      printf("loc_14dah: error p3 %x %x %x\n", r13, r6, r2);
      fatal = 1;
    }
  }
}

void execute_layer1(uint from, uint to) {
#ifdef DEBUG
  uint step = to - from;
  uint check = from + (step / 1000);
  printf("# %u %u\n", from, to);
#endif
  uint key = from;
  uint r4_save = r4;
  unsigned char* ram_save = malloc(RAMSIZE);
  memcpy(ram_save, ram, RAMSIZE);
  while (key < to) {
#ifdef DEBUG
    if (key > check) {
      printf("# %3lf/100  %u  %u/%u\n", (key - from) * 100.0 / step, (
          uint)key, key -from, step);
      check = check + (step / 1000);
    }
#endif
    memcpy(ram, ram_save, RAMSIZE);
    r4 = r4_save;
    ram[0xa000] = key & 0xFF;
    ram[0xa001] = (key >>  8) & 0xFF;
    ram[0xa002] = (key >> 16) & 0xFF;
    ram[0xa003] = (key >> 24) & 0xFF;
    fatal = 0;
    stop = 0;
    uint cont = 1;
```

```
    while (cont && (fatal == 0) && (stop == 0)) {
#ifdef DEBUG
      printf("%04x %04x ", GET_SHORT_FROM_RAM(0x111c), GET_SHORT_FROM_RAM
          (0x111e));
#endif
      loc_14dah();
#ifdef DEBUG
      printf("\n");
#endif
      cont = ! (GET_SHORT_FROM_RAM(0x111c) == 0x577);
      //cont = ((ram[0x8000] + (ram[0x8001] << 8) + (ram[0x8002] << 16) +
           (ram[0x8003] << 24)) == 0);
    }
    if (fatal != 0) {
      printf("fatal with: %16x\n", (uint)key);
    }
    uint result1 = ram[0xa000] + (ram[0xa001] << 8) + (ram[0xa002] << 16)
         + (ram[0xa003] << 24);
    uint result2 = ram[0x8000] + (ram[0x8001] << 8) + (ram[0x8002] << 16)
         + (ram[0x8003] << 24);
#ifdef DEBUG
    printf("%16x - %16x - %16x\n", key, result1, result2);
#endif
    if (key != result1) {
      printf("valid: %16x - %16x - %16x\n", (uint)key, result1, result2);
      key_layer1 = key;
      key_valid = 1;
    }
    key++;
  }
}

void execute_layer2(uint from, uint to) {
  memcpy(&ram[0xa000 + 16], &blob, 0x100);
  ram[0xa000] = (key_layer1 >> 16) & 0xFF;
  ram[0xa001] = (key_layer1 >> 24) & 0xFF;
  ram[0x8000] = 0x0;
  ram[0x8001] = 0x0;
  ram[0x8002] = 0x0;
  ram[0x8003] = 0x0;
  ram[0x111c] = 0x0;
  ram[0x111d] = 0x0;
  ram[0x111e] = 0x0;
  ram[0x111f] = 0x0;
  uint r4_save = r4;
  unsigned char* ram_save = malloc(RAMSIZE);
  memcpy(ram_save, ram, RAMSIZE);
  uint key = from;
  while (key < to) {
    memcpy(ram, ram_save, RAMSIZE);
    r4 = r4_save;
    ram[0xa002] = (key >> 0) & 0xFF;
    ram[0xa003] = (key >> 8) & 0xFF;
    fatal = 0;
    stop = 0;
    uint cont = 1;
    while (cont && (fatal == 0) && (stop == 0)) {
#ifdef DEBUG
      printf("%04x %04x ", GET_SHORT_FROM_RAM(0x111c), GET_SHORT_FROM_RAM
          (0x111e));
#endif
      loc_14dah();
#ifdef DEBUG
      printf("\n");
#endif
      cont = ((ram[0x8000] + (ram[0x8001] << 8) + (ram[0x8002] << 16) + (
          ram[0x8003] << 24)) != 0xbeefdead);
      //cont = ! (GET_SHORT_FROM_RAM(0x111c) == 0x0e09);
    }
    if (fatal != 0) {
      printf("fatal with: %16x\n", (uint)key);
    }
    uint result1 = ram[0xa000] + (ram[0xa001] << 8) + (ram[0xa002] << 16)
         + (ram[0xa003] << 24);
    uint result2 = ram[0x8000] + (ram[0x8001] << 8) + (ram[0x8002] << 16)
         + (ram[0x8003] << 24);
    printf("? key: %16x - %x - %x\n", key, ram[0xa010+0xFE] & 0xFF, ram[0
        xa010+0xFF] & 0xFF);
    if ((ram[0xa010 + 0xFF] != 0xFF) && (ram[0xa010 + 0xFE] != 0xFF)) {
      printf("valid: %16x - %16x - %16x\n", (uint)key, result1, result2);
      key_layer2 = result1;
      key_valid = 1;
    }
    key++;
  }
}

void execute_layer3(uint from, uint to) {
  memcpy(&ram[0xa000 + 16], &blah, 0x20);
  ram[0xa000] = (0xf63d >> 0) & 0xFF;
  ram[0xa001] = (0xf63d >> 8) & 0xFF;
  ram[0x8000] = 0x0;
  ram[0x8001] = 0x0;
  ram[0x8002] = 0x0;
  ram[0x8003] = 0x0;
  ram[0x111c] = 0x0;
```

```
  ram[0x111d] = 0x0;
  ram[0x111e] = 0x0;
  ram[0x111f] = 0x0;
  uint r4_save = r4;
  unsigned char* ram_save = malloc(RAMSIZE);
  memcpy(ram_save, ram, RAMSIZE);
  uint key = from;
  while (key < to) {
    memcpy(ram, ram_save, RAMSIZE);
    r4 = r4_save;
    ram[0xa002] = (key >> 0) & 0xFF;
    ram[0xa003] = (key >> 8) & 0xFF;
    fatal = 0;
    stop = 0;
    uint cont = 1;
    while (cont && (fatal == 0) && (stop == 0)) {
#ifdef DEBUG
      printf("%04x %04x ", GET_SHORT_FROM_RAM(0x111c), GET_SHORT_FROM_RAM
          (0x111e));
#endif
      loc_14dah();
#ifdef DEBUG
      printf("\n");
#endif
      cont = ((ram[0x8000] + (ram[0x8001] << 8) + (ram[0x8002] << 16) + (
          ram[0x8003] << 24)) == 0);
    }
    if (fatal != 0) {
      printf("fatal with: %16x\n", (uint)key);
    }
    uint result1 = ram[0xa000] + (ram[0xa001] << 8) + (ram[0xa002] << 16)
        + (ram[0xa003] << 24);
    uint result2 = ram[0x8000] + (ram[0x8001] << 8) + (ram[0x8002] << 16)
        + (ram[0x8003] << 24);
    printf("? key:%04x  - %c %c\n", key, ram[0xa000 + 16], ram[0xa000 +
        17]);
    if (!strncmp(ram+0xa010, "V29vdCAhISBTbWVsbHMgZ29vZCA6KQ==", 32)) {
    //if ((ram[0xa010] == 'V') && (ram[0xa011] == '2')) {
      printf("valid: %16x - %16x - %16x\n", (uint)key, result1, result2);
      key_layer2 = result1;
      key_valid = 1;
    }
    key++;
  }
}

uint swap_key(uint key32b) {
  unsigned char b0, b1, b2, b3;
  b3 = (char)(key32b >> 24) & 0xFF;
  b2 = (char)(key32b >> 16) & 0xFF;
  b1 = (char)(key32b >>  8) & 0xFF;
  b0 = (char)(key32b >>  0) & 0xFF;
  return (b2 << 24) | (b3 << 16) | (b0 <<  8) | (b1 <<  0);
}

int main(int argc, char** argv) {
  create();
  init();

  // LAYER1

  key_valid = 0;
  layer = layer1;
  execute_layer1(0x94e3e5df, 0x94e3e5e0);
  if (key_valid == 0) {
    printf("ERROR Layer1\n");
    exit(2);
  }

  // LAYER2

  layer = layer2;
  uint key_from_layer1 = ram[0xa000] + (ram[0xa001] << 8) + (ram[0xa002]
      << 16) + (ram[0xa003] << 24);
  key_from_layer1 = swap_key(key_from_layer1);
  int order[] = {0, 8, 16, 24};

  layer[0] = layer[0] ^ ((key_from_layer1 >> order[0]) & 0xFF);
  layer[1] = layer[1] ^ ((key_from_layer1 >> order[1]) & 0xFF);
  layer[2] = layer[2] ^ ((key_from_layer1 >> order[2]) & 0xFF);
  layer[3] = layer[3] ^ ((key_from_layer1 >> order[3]) & 0xFF);

  int index = 4;
  do {
    layer[index] = layer[index] ^ layer[index - 4];
    index++;
  } while (index < sizeof(layer2));

  key_valid = 0;

  // execute_layer2(0xc795, 0xc796);
  execute_layer2(0xf63d, 0xf63e);

  if (key_valid == 0) {
    printf("ERROR Layer2\n");
    exit(2);
```

```
  }

  // LAYER3

  layer = layer3;
  index = 0;
  int len = 0x6E3;
  int i = 0;

  unsigned char* ksa = &ram[0xa000 + 16];
  unsigned char* buf1 = layer;
  int j = 0;
  int tmp;
  while (index < len) {
    i = (i + 1) & 0xFF;
    j = (j + ksa[i]) & 0xFF;
    tmp = ksa[i];
    ksa[i] = ksa[j];
    ksa[j] = tmp;
    buf1[index] = layer[index] ^ ksa[(ksa[i] + ksa[j]) & 0xFF];
    index++;
  }

#if DUMP
  int fd;
  fd = open("layer3.dec", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR | S_IRGRP |
       S_IROTH);
  write(fd, layer, 0x6E3);
  close(fd);
#endif

  /*
  printf("\nlayer3 = [");
  index = 0;
  while (index < sizeof(layer2)) {
    printf("0x%x, ", layer[index]);
    index++;
    if (index % 8 == 0) {
      printf("\n");
    }
  }
  printf("]\n");
  */

  execute_layer3(0, 65536);
  destroy();
  return 0;
}
```

Listing 46 – Code source de *interpreter.c*