Solution du challenge SSTIC 2015

Aurélien Deharbe
aurelien.deharbe@lip6.fr

27 avril 2015

Résumé

Depuis plusieurs années, la conférence SSTIC sur le thème de la sécurité de l'information propose un challenge dont le but est de trouver une adresse e-mail à partir d'un fichier fourni. Ce challenge se découpe en plusieurs étapes, invitant à la rétro-ingénierie de plusieurs formats de fichiers, technologies, langages de programmation, ou encore d'algorithmes de chiffrement de données. Dans ce rapport, nous présentons la démarche qui nous a permis de venir à bout de l'édition 2015 de ce challenge.

Le défi de cette année est composé d'une succession de 6 "stages". L'organisation de ce document suit donc naturellement ce découpage. Dans la première partie, nous découvrons le challenge et récupérons un fichier injecté dans une machine à l'aide d'une clé USB un peu particulière. Le second stage nous emmène dans le jeu vidéo OpenArena, dans lequel une clé de chiffrement a été dissimulée. Une fois cette clé retrouvée, nous pourrons déchiffrer le fichier du stage suivant. Il s'agit alors d'étudier des échanges de données capturés entre une souris USB et un système, afin de reproduire les mouvements de souris effectués par une personne ayant cachée une nouvelle clé de chiffrement dans une image sous Paint. Le quatrième stage traite d'une technique d'obfuscation de code en Javascript. L'avant-dernier stage nous plonge dans le langage assembleur d'un microcontrôleur de la famille ST20, via un *firmware* que nous devrons étudier pour en reproduire le comportement, dans le but de déchiffrer un nouveau fichier. Enfin, le dernier stage, nous offrira un panorama sur plusieurs techniques de dissimulation de données dans des images. C'est à l'issue de ces étapes de stéganographie que nous découvrirons finalement la précieuse adresse e-mail.

La liste des outils utilisés figure en annexe de ce document.

Table des matières

Listings

Stage 1 : Duckyscript

1.1 Découverte de *inject.bin*

Le fichier de départ nous est présenté comme l'image disque d'une "carte microSD qui était insérée dans une clé USB étrange". Nous commençons par vérifier l'intégrité du fichier téléchargé et par confirmer son format.

```
$ wget --quiet http://static.sstic.org/challenge2015/challenge.zip
$ shasum -a 256 challenge.zip
bd0df75a1d6591e01212e6e28848aec94b514e17e4ac26155696a9a76ab2ea31  challenge.zip
$ unzip challenge.zip
Archive:  challenge.zip
  inflating: sdcard.img
$ file sdcard.img
sdcard.img: x86 boot sector, mkdosfs boot message display, code offset 0x3c, OEM-ID "mkfs.fat", sec
tors/cluster 4, root entries 512, Media descriptor 0xf8, sectors/FAT 244, heads 64, sectors 250000
(volumes > 32 MB) , serial number 0xe50d883b, unlabeled, FAT (16 bit)
```

Il s'agit de l'image d'un disque formaté en FAT 16 bits, système de fichiers couramment utilisé pour les cartes SD.

```
$ mkdir sdcard
$ sudo mount -t msdos -o loop sdcard.img sdcard
$ ls -a sdcard
.           inject.bin
..
$ file sdcard/inject.bin
inject.bin: data
```

Dans l'immédiat, nous n'avons aucun renseignement sur la nature du seul fichier *inject.bin* présent dans l'image du disque. Mais une recherche sur le web nous révèle qu'il s'agit du *payload* encodé depuis un script Duckyscript, destiné à être embarqué sur une clé USB Rubber Ducky [1]. Ce type de clé USB a la particularité de se comporter comme un clavier, en envoyant automatiquement (et donc à très haute vitesse) des signaux de touches au système sur lequel elle est branchée. Ce gadget peut alors être utilisé à des fins diverses et variées ; on citera par exemple la récupération de données ou bien l'injection de fichiers sur une machine. Cette information concorde avec le descriptif du challenge, le *payload* d'un Rubber Ducky étant embarqué sur une carte microSD, elle-même insérée dans la clé USB. Le dépôt de code officiel du Rubber Ducky [2] contient un programme

1. http://hakshop.myshopify.com/products/usb-rubber-ducky-deluxe
2. https://github.com/hak5darren/USB-Rubber-Ducky

Java permettant d'encoder du Duckyscript. Le décodage consiste donc à effectuer la transformation inverse. Nous utilisons pour cela le script fourni dans un des nombreux forks du projet et disponible à l'adresse https://github.com/midnitesnake/USB-Rubber-Ducky/blob/master/Decode/ducky-decode.pl.

```
$ perl ducky-decode.pl -f inject.bin > ducky.txt
$
```

Le listing 1.1 présente le début du fichier Duckyscript ainsi obtenu ; nous l'étudions dans la section suivante.

1.2 Reconstruction du fichier injecté

Listing 1.1– ducky.txt : résultat du décodage de *inject.bin*

```
00ff 007d
GUI R

DELAY 500

ENTER

DELAY 1000
 c m d
ENTER

DELAY 50
 p o w e r s h e l l
SPACE
 - e n c
SPACE
 Z g B 1 A G 4 A Y w B 0 A G k A b w B u A C A A d w B y A G k A d A B l A F 8 A Z g B p A G w
 A Z Q B f A G I A e Q B 0 A G U A c w B 7 A H A A Y Q B y A G E A b Q A o A F s A Q g B 5 A H
 Q A Z Q B b A F 0 A X Q A g A C Q A Z g B p A G w A Z Q B f A G I A e Q B 0 A G U A c w A s A
 C A A W w B z A H Q A c g B p A G 4 A Z w B d A C A A J A B m A G k A b A B l A F 8 A c A B h
 [...]
 B B A E g A S Q B B A C c A K Q A p A D s A f Q A = 00a0
ENTER
 p o w e r s h e l l
SPACE
 - e n c
SPACE
  Z g B 1 A G 4 A Y w B 0 A G k A b w B u A C A A d w B y A G k A d A B l A F 8 A Z g B p A G w
  [...]
```

La syntaxe Duckyscript est la suivante : un script se compose de commandes (en majuscules), suivies d'éventuels arguments. Les commandes qui nous intéressent ici sont :

— `GUI R` émule la touche "windows",
— `DELAY` n delaye la prochaine instruction de n millisecondes,
— `STRING` str émule les touches des caractères qui constituent la chaîne str,
— `ENTER`, `SPACE` émulent les touches spéciales correspondantes.

La commande `STRING` a été omise lors du décodage du *payload*, on identifie toutefois les touches devants être directement émulées par un espace en début de ligne.

Après avoir lancé une invite de commandes, ce script va donc entrer la commande `powershell -enc ZgB1AG4...` puis l'exécuter, avant de poursuivre avec la commande Powershell suivante. L'option "-enc" du Powershell de Windows est la version courte de l'option "-EncodedCommand", qui attend en argument suivant des instructions encodées en base64 à exécuter dans la session Powershell. Quelques commandes dans un terminal nous permettent de reconstituer le premier groupe d'instructions à exécuter :

```
$ cat ducky.txt | grep "Z g B 1" | head -n 1 | base64 --decode
function write_file_bytes{param([Byte[]] $file_bytes, [string] $file_path = ".\stage2.zip");$f = [i
o.file]::OpenWrite($file_path);$f.Seek($f.Length,0);$f.Write($file_bytes,0,$file_bytes.Length);$f.C
lose();}function check_correct_environment{$e=[Environment]::CurrentDirectory.split("\");$e=$e[$e.L
ength-1]+[Environment]::UserName;$e -eq "challenge2015sstic";}if(check_correct_environment){write_f
ile_bytes([Convert]::FromBase64String('UEsDBAoDAAAAADaKeUbfS/XdEKUHABClBwAJAAAAZW5jcnlwdGVkfL/V3iij
vVZtKEk8fNyka1PvkbrI0KkNNLxpo0np9mik2+VSO8DpYpziIGfM8vObQ6eCnQeaSrcOE9Lp9sRkxSyqYp/V1tjWu5V1NeJs4Vo
m37rm7Pdoym+SoGdEyrFay1pbK1rzSNAfJiLMvtDBPZSwPcYnDg6Irh1U5U+o5MmS+LhV5LFRRTqcbi8INTNCfig/l1xFdK+/wF
Gai22ni90Mkzxzz0rV+09nBCHnm1+h5CAr2foL9oe8e11GNNKZt1mUmf4OB+/kw93C+Bya4HCBjJ3n7v+fBTwZq3D6BrlqRS1cn
yZsEGllLopvLxJUSMwUlL+uD2VBoaRodD/ZCYh2YEMRtTgNF+pN/2O2lQK79RsyO6EQL6RfejxdPYsPxqvPfw7UsnfjLydxKoPL
PMOSxORYo1tcdh81ozeR4Iv3ZaYFUJlTFYSWpCBQtZFEuQfhdUTsVTCf0LcVG7fFtMxG9mwr+kez7Wt7VyANSQsKzxiSEU+2x+x
AG0A1U5rDUqOOhdupHI1+TLggOHlbHP2KKbW0dYtRYKomNEWhApdZWhHSU+QF2KYhFYDx8xGiEUpId8kVvif1tAbNpA+WzmvXyH
lofmb3SMXqbngAC8KkHM28Nmjx8D+6kpyhawnR8YRxJEbNdJOSLnfAGgCqRFQExQqHl4FfDpPQUGM2s1rw4mFDN6xiwXFKoXtBV
UH2EESo67u5TmWmpBUOqPA28iXC/4ZVdn1nqOmOCR2aHXYQBiQ9uJG0QOukor8XphyV4mfWHgCnytpVhF61W0kDsid7cullAe52
+m0Ze5s28bF22xAjRdy0lY+0ALp1UKLUUqJbEYG46h++8VKzNEbKdHjkeSoaDtqFoen0e6C+W67qmMWxGlZQC4nM1nkH9fkFiOW
lPYsDHWHxaBXAJ0HeW3HjJ3Ec0SMNuW7KTM5mH2JU5BpFshzbbKAh9fa+7GMT9F3ebLaW2KEGQuvkfx5XJQwnc+FPUAyTkf9yoL
l4vlG1ieC547zY4kBenCy09IeWOxWCWTyKak+kYAmKsohUEYkF5dKArhFCnZZYmUCn1kDXftQommJWrGuya69nR+ZvtuQAmSDlg
orWgJl/Q89vA7OA+KXB8eTPtBO7HkU6xRbIg+It3fChNGOLM6LJvtC/V6FbiXqeXGcYUTiX4/sKlCqbWIllxKQi0jc4DHZKmani
ZGDhtJKxX1zfv1t6OcI0TU5oNdv2GcqWIn0WBATN/spOiFsY9etTfjHu7pHGq+khCnRB3REA4AvKyKYfPb+nXHdCvsY4S+s8AJF
W2UwdXtOh6gUsBzWnXw==’));}else{write_file_bytes([Convert]::FromBase64String(’VAByAHkASABhAHIAZABlAH
IA’));}
```

Au total, 3390 groupes d'instructions Powershell seront exécutées par le Rubber Ducky contenant ce *payload*. Seul le code du dernier groupe diffère des autres par son fonctionnement. Les deux types de scripts Powershell sont présentés dans le listing 1.2.

Listing 1.2– inject.ps1 : scripts Powershell

```
function write_file_bytes {
  param([Byte[]] $file_bytes, [string] $file_path = ".\stage2.zip");
  $f = [io.file]::OpenWrite($file_path);
  $f.Seek($f.Length ,0);
  $f.Write($file_bytes ,0,$file_bytes.Length);
  $f.Close();
}

function check_correct_environment {
  $e=[Environment]::CurrentDirectory.split("\");
  $e=$e[$e.Length-1] + [Environment]::UserName;
  $e -eq "challenge2015sstic";
}

if (check_correct_environment) {
  write_file_bytes([Convert]::FromBase64String(’UEsDBAoDAAAAADaKe[...]==’));
} else {
  write_file_bytes([Convert]::FromBase64String(’VAByAHkASABhAHIAZABlAHIA’));
}

[...]

function hash_file {
  param([string] $filepath);
  $sha1 = New-Object -TypeName System.Security.Cryptography.SHA1CryptoServiceProvider;
  $h = [System.BitConverter]::ToString(
    $sha1.ComputeHash([System.IO.File]::ReadAllBytes($filepath))
  );
  $h
}

$h = hash_file(".\stage2.zip");
if ($h -eq "EA-9B-8A-6F-5B-52-7E-72-65-20-19-31-3C-25-B5-6A-D2-7C-7E-C6") {
  echo "You WIN";
} else {
  echo "You LOSE";
}
```

L'exécution du script complet effectue une suite d'écriture dans le fichier "stage2.zip", puis calcule le condensat SHA-1 de ce fichier avant de le comparer avec un condensat de référence pour afficher un message de réussite ou d'échec d'injection de fichier. Chaque nouvelle écriture est faite à la fin du fichier, après le déplacement du curseur de fichier via la méthode `Seek`. On

remarque un test conditionnel avant l'appel à la fonction d'écriture : si le nom d'utilisateur de la machine n'est pas "challenge2015sstic", c'est la chaîne "TryHarder" (en base64 : "VAByAHkA-SABhAHIAZABlAHIA") qui est écrite dans le fichier. Cette protection a certainement été introduite par les concepteurs du challenge pour que les possesseurs d'un USB Rubber Ducky ne puisse pas simplement rejouer l'injection du fichier sur leurs propres machines.

Le listing 1.3, en Python, nous permet d'extraire le contenu de chaque packet à écrire dans le fichier stage2.zip et ainsi terminer cette première épreuve.

Listing 1.3– to-stage2.py : reconstruction de *stage2.zip*

```
import base64
import re

st2file = open('stage2.zip', 'wb')

with open('ducky.txt') as f:
    data = f.readlines()

    for i in range(16, len(data), 6):
        cmd64 = "".join(data[i].split('␣'))
        cmd = base64.b64decode(cmd64)
        cmd = str("".join(str(cmd).split('\\x00')))

        m = re.search('FromBase64String(.*)\}else\{write_file', cmd)

        if m:
            chunk = base64.b64decode(m.groups()[0][3:-5])
            st2file.write(chunk)

st2file.close()
```

```
$ python to-stage2.py
$ shasum stage2.zip
ea9b8a6f5b527e72652019313c25b56ad27c7ec6  stage2.zip
```

Le condensat du fichier *stage2.zip* correspond bien à celui permettant d'afficher "You WIN" lors du dernier test du script Powershell. Nous découvrons son contenu dans le prochain chapitre.

Stage 2 : OpenArena

```
$ unzip stage2.zip
Archive:  stage2.zip
 extracting: encrypted
  inflating: memo.txt
  inflating: sstic.pk3
$ cat memo.txt
Cipher: AES-OFB
IV: 0x5353544943323031352d537461676532
Key: Damn... I ALWAYS forget it. Fortunately I found a way to hide it into my favorite game !

SHA256: 91d0a6f55cce427132fc638b6beecf105c2cb0c817a4b7846ddb04e3132ea945 - encrypted
SHA256: 845f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187 - decrypted
```

Nous voici maintenant face à un fichier encrypté. D'après le mémo contenu également dans l'archive, il s'agirait d'un fichier encrypté à l'aide du chiffrement par bloc AES-OFB. Le vecteur d'initialisation utile au déchiffrement nous est fourni (et n'est nul autre que la chaîne de caractères 'SSTIC2015-Stage2'), ainsi que le condensat du fichier *decrypted* que nous devons obtenir. Il nous reste donc à retrouver la clé, qui, d'après le mémo, serait cachée dans un jeu. Le dernier fichier présent dans l'archive est le fichier *sstic.pk3*.

2.1 Le format de fichier PK3

Une recherche sur le web nous apprend que le format de fichier PK3 [1] est employé par les jeux reposant sur le moteur *Quake III Engine*. Un fichier PK3 est en fait une archive au format ZIP renommée, et contenant des maps, des modèles 3D, des textures, des sons, ou encore des scripts nécessaires à la description de niveaux de jeu.

```
$ mv sstic.pk3 sstic.zip
$ unzip sstic.zip
Archive:  sstic.zip
  inflating: AUTHORS
   creating: levelshots/
  inflating: levelshots/sstic.tga
   creating: maps/
  inflating: maps/sstic.bsp
  inflating: README
   creating: scripts/
  inflating: scripts/sstic.arena
```

1. http://en.wikipedia.org/wiki/PK3_%28file_extension%29

```
   creating: sound/
   creating: sound/world/
  inflating: sound/world/bj3.wav
   creating: textures/
   creating: textures/sstic/
  inflating: textures/sstic/01.tga
  inflating: textures/sstic/02.tga
  inflating: textures/sstic/103336131.tga
  inflating: textures/sstic/1036082074.tga
  inflating: textures/sstic/1039223422.tga
  inflating: textures/sstic/1042015984.tga
  [...]
```

On remarque la présence de nombreux fichiers images dans un dossier de textures nommé "sstic". En faisant abstraction des logos, on identifie deux types d'images, dont un exemple de chaque est donné en figure 2.1. L'une représente trois parties possibles de clé de déchiffrement, identifiées par une icône ainsi qu'une couleur (orange, blanc ou bleu). L'autre type d'image est ce que nous appelerons un "sélecteur", composé d'une icône et d'une couleur, qui nous permettra de sélectionner une image et une partie de clé.

FIGURE 2.1 – Exemples de textures : à gauche, une partie de clé, et à droite, un sélecteur.

80 images contenant des parties de clé et 30 sélecteurs figurent dans le dossier des textures. L'étape suivante consiste donc à isoler les images composant la clé de déchiffrement du fichier *encrypted*. Pour cela, nous étudions maintenant directement le fichier de la map *sstic.bsp*.

2.2 Le format de fichier BSP

Une documentation non officielle détaillée sur le format de fichier BSP est présente à l'adresse suivante : `http://www.mralligator.com/q3/`. Un fichier BSP décrit entre autres donées des faces 3D constituant une map, composées de sommets dans l'espace 3D (*vertices*). Ces faces sont ensuite rangées par groupe dans les nœuds ou dans les feuilles d'un arbre BSP. Cette technique est couramment utilisée dans les moteurs d'affichage 3D afin de séparer une scène en plusieurs parties, et de ne calculer le rendu que des objets du sous-arbre visible depuis la caméra. Outre ces informations de modélisation de la map, le fichier BSP contient également des données utiles à la détection de collision, à l'habillage et au rendu (textures et lumières), et aux événements de jeux. Toutes ces données sont organisées dans des "*lumps*" ; sortes de *chunks* de données. Enfin, la commande `strings` sur le fichier *sstic.bsp* nous confirme la présence de la clé à un certain endroit de la map, via un message trouvé dans le *lump* "*entities*" du fichier :

```
$ strings maps/sstic.bsp
[...]
"message" "Yes!\n You found my key !"
[...]
```

Le header du fichier BSP présente ainsi (les entiers et flottants sont encodées dans l'ordre *little-endian*, sur 4 octets) :

1. 4 octets représentent le "nombre magique" du fichier, ici les caractères "IBSP",
2. 4 octets forment le numéro de version,
3. 17 fois 8 octets sont utilisés par la table des *lumps* : pour chaque type de *lump*, on y retrouve son offset de départ dans le fichier ainsi que sa taille.

Notre objectif étant de repérer les textures effectivement utilisées lors du rendu de la map dans le jeu, 3 *lumps* nous intéressent particulièrement :

— le 2ème *lump* contient une table des textures disponibles pour cette map,
— le 13ème *lump* contient une table des *vertices* décrivant les faces de la scène,
— le 14ème *lump* contient une table de toutes les faces composant la scène.

Un premier script (listing 2.1) nous permet de parcourir ces données et d'en extraire toutes les faces. Chaque face contient un entier faisant référence à une texture dans la table des textures. On filtre ensuite les textures référencées par des faces pour ne conserver que celles dont le chemin de fichier contient la chaîne de caractères "sstic".

Listing 2.1– bsp-inspect.py : repérage des textures utilisées

```
import struct

textures = []
faces = []

def readInt(f):
    return struct.unpack("<i", f.read(4))[0]

with open('maps/sstic.bsp', 'r') as f:
    magic = f.read(4)
    assert(magic == "IBSP")
    version = readInt(f)
    assert(version == 0x2e)

    # parsing lumps directory
    lumpsdir = []
    for i in range(17):
        offset = readInt(f)
        size   = readInt(f)
        lumpsdir.append({ "offset": offset, "size": size })

    # parsing textures
    f.seek(lumpsdir[1]["offset"])
    for i in range(lumpsdir[1]["size"] / 72):
        textures.append({ "name": f.read(64).split("\x00")[0] })
        f.read(8) # skip other texture data

    # parsing faces
    f.seek(lumpsdir[13]["offset"])
    for i in range(lumpsdir[13]["size"] / 104):
        faces.append({ "textureId": readInt(f) })
        f.read(100) # skip other face data

for face in faces:
    t = textures[face["textureId"]]
    if "sstic" in t["name"]:
        print(t["name"])
```

```
$ python bsp-inspect.py | sort | uniq
extures/sstic/01
```

```
textures/sstic/02
textures/sstic/1219191984
textures/sstic/123771438
textures/sstic/1429015180
textures/sstic/1509983054
textures/sstic/1604401524
textures/sstic/1749623519
textures/sstic/1773100136
textures/sstic/19281330
textures/sstic/2036414783
textures/sstic/31570422
textures/sstic/381650771
textures/sstic/457615433
textures/sstic/52809216
textures/sstic/643008245
textures/sstic/71921642
textures/sstic/747908186
textures/sstic/838783866
textures/sstic/855646320
textures/sstic/86520831
textures/sstic/logo
```

Parmi ces 22 textures figurent :

— 1 logo du SSTIC,

— 2 images d'un diablotin,

— 1 image vide au fond rouge,

— 8 sélecteurs,

— 10 images contenant 3 parties de clé chacune.

8 sélecteurs permettent de sélectionner $8 * 4 = 32$ octets de clé, soit une longueur de clé correspondant pour un déchiffrement AES-256.

Parmi les 10 dernières images citées figure exactement une pour chaque sélecteur, sauf pour l'icône du disque dur et celle de la goutte d'eau pour lesquelles deux images sont sélectionnables. Afin de réduire encore le champ des clés possibles, nous recherchons maintenant l'ordre d'affichage des sélecteurs à l'écran. Pour cela, nous exploitons le *lump* des *vertices* contenant les coordonnées des sommets 3D des coins des faces. On n'observe désormais plus que les faces correspondant à nos sélecteurs de parties de clé, et on en affiche les coordonnées des *vertices* les construisant (listing 2.2).

Listing 2.2– bsp-inspect-bis.py : recherche des coordonnées des sélecteurs

```
import struct

textures = []
vertices = []
faces = []

def readInt(f):
    return struct.unpack("<i", f.read(4))[0]

def readFloat(f):
    return struct.unpack("<f", f.read(4))[0]

with open('maps/sstic.bsp', 'r') as f:
    magic = f.read(4)
    assert(magic == "IBSP")
    version = readInt(f)
    assert(version == 0x2e)

    # parsing lumps directory
    lumpsdir = []
    for i in range(17):
        offset = readInt(f)
        size   = readInt(f)
        lumpsdir.append({ "offset": offset, "size": size })
```

```
    # parsing textures
    f.seek(lumpsdir[1]["offset"])
    for i in range(lumpsdir[1]["size"] / 72):
        textures.append({ "name": f.read(64).split("\x00")[0] })
        f.read(8) # skip other texture data

    # parsing vertices
    f.seek(lumpsdir[10]["offset"])
    for i in range(lumpsdir[10]["size"] / 44):
        x = readFloat(f)
        y = readFloat(f)
        z = readFloat(f)
        vertices.append((x, y, z))
        f.read(32)

    # parsing faces
    f.seek(lumpsdir[13]["offset"])
    for i in range(lumpsdir[13]["size"] / 104):
        t = readInt(f)
        f.read(8)
        v = readInt(f)
        n = readInt(f)
        faces.append({ "textureId": t, "vertId": v, "nbVerts": n })
        f.read(84)

selecs = ["19281330", "123771438", "381650771", "747908186",
          "1219191984", "1429015180", "1749623519", "2036414783"]

for face in faces:
    t = textures[face["textureId"]]
    if "sstic" in t["name"]:
        if t["name"].split("/")[-1] in selecs:
            print(t["name"] + ": ")
            for vi in range(face["vertId"], face["vertId"] + face["nbVerts"]):
                print("x = %f,\t y = %f,\t z = %f" % vertices[vi])
```

```
$ python bsp-inspect-bis.py
textures/sstic/123771438:
x = -2611.250000,        y = 2401.750000,        z = -352.500000
x = -2680.000000,        y = 2401.750000,        z = -352.500000
x = -2611.250000,        y = 2401.750000,        z = -321.250000
x = -2680.000000,        y = 2401.750000,        z = -321.250000
textures/sstic/747908186:
x = -2210.750000,        y = 2401.750000,        z = -352.500000
x = -2279.500000,        y = 2401.750000,        z = -352.500000
x = -2210.750000,        y = 2401.750000,        z = -321.250000
x = -2279.500000,        y = 2401.750000,        z = -321.250000
textures/sstic/381650771:
x = -2290.500000,        y = 2401.750000,        z = -352.500000
x = -2359.250000,        y = 2401.750000,        z = -352.500000
x = -2290.500000,        y = 2401.750000,        z = -321.250000
x = -2359.250000,        y = 2401.750000,        z = -321.250000
textures/sstic/19281330:
x = -2371.250000,        y = 2401.750000,        z = -352.500000
x = -2440.000000,        y = 2401.750000,        z = -352.500000
x = -2371.250000,        y = 2401.750000,        z = -321.250000
x = -2440.000000,        y = 2401.750000,        z = -321.250000
textures/sstic/2036414783:
x = -2450.250000,        y = 2401.750000,        z = -352.500000
x = -2519.000000,        y = 2401.750000,        z = -352.500000
x = -2450.250000,        y = 2401.750000,        z = -321.250000
x = -2519.000000,        y = 2401.750000,        z = -321.250000
textures/sstic/1749623519:
x = -2530.750000,        y = 2401.750000,        z = -321.250000
x = -2599.500000,        y = 2401.750000,        z = -352.500000
x = -2599.500000,        y = 2401.750000,        z = -321.250000
x = -2530.750000,        y = 2401.750000,        z = -352.500000
textures/sstic/1429015180:
x = -2130.250000,        y = 2401.750000,        z = -321.250000
x = -2199.000000,        y = 2401.750000,        z = -352.500000
x = -2199.000000,        y = 2401.750000,        z = -321.250000
x = -2130.250000,        y = 2401.750000,        z = -352.500000
textures/sstic/1219191984:
```

```
x = -2691.000000,        y = 2401.750000,        z = -321.250000
x = -2759.750000,        y = 2401.750000,        z = -352.500000
x = -2759.750000,        y = 2401.750000,        z = -321.250000
x = -2691.000000,        y = 2401.750000,        z = -352.500000
```

Le résultat est encourageant : la composante y ne varie pas, ce qui signifie que les sélecteurs sont bien tous sur le même plan (probablement un mur de la map), et la composante z n'a que deux positions possibles (le haut et le bas des faces), ce qui confirme que les sélecteurs sont bien alignés sur ce plan. On peut maintenant recomposer 2 ordres possibles (dans les 2 sens) pour nos 8 sélecteurs, en triant les faces sur lesquelles ils apparaissent selon leurs coordonnées en x. Nous disposons désormais de suffisamment d'informations pour retrouver la clé complète : il y a 2 sens possibles, et 2 des sélecteurs peuvent faire référence à 2 textures différentes, ce qui nous donne un total de 8 clés seulement à tester. Le script présenté dans le listing 2.3 effectue ces tests. Les critères de succès d'une clé sont les suivants :

1. Le fichier décrypté est très probablement une nouvelle archive au format ZIP, on teste donc si le nombre magique du fichier résultant correspond bien à celui d'un fichier ZIP ("PK").
2. On dispose du condensat du fichier décrypté afin de vérifier l'intégrité du fichier du stage suivant. Les blocs de l'AES-256 étant de 32 octets de long, on vérifie la somme SHA-256 en supprimant jusqu'à 31 caractères du dernier bloc, afin de supprimer un éventuel *padding*.

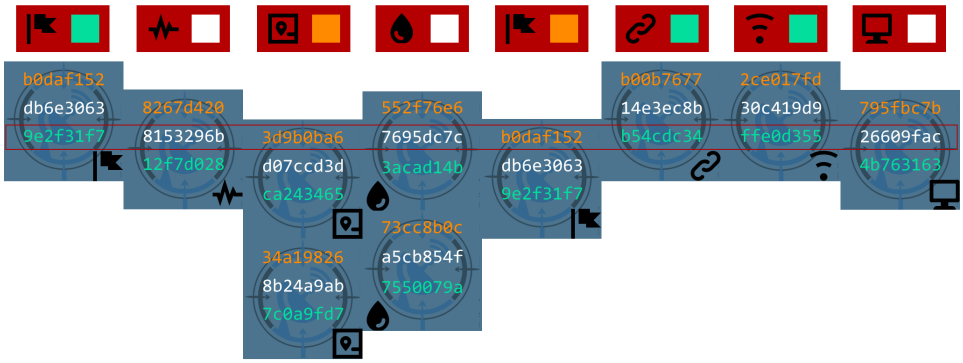
Listing 2.3– bf-st2key.py : déchiffrement de *stage3.zip*

```
from Crypto.Cipher import *
import hashlib
import binascii

with open('encrypted', 'r') as f:
    data = f.read()
assert(hashlib.sha256(data).hexdigest() ==
       "91d0a6f55cce427132fc638b6beecf105c2cb0c817a4b7846ddb04e3132ea945")

kparts1 = ["\x9e\x2f\x31\xf7", "\x81\x53\x29\x6b"]
opts2   = ["\x34\xa1\x98\x26", "\x3d\x9b\x0b\xa6"]
opts3   = ["\xa5\xcb\x85\x4f", "\x76\x95\xdc\x7c"]
kparts2 = ["\xb0\xda\xf1\x52", "\xb5\x4c\xdc\x34",
           "\xff\xe0\xd3\x55", "\x26\x60\x9f\xac"]

iv = "\x53\x53\x54\x49\x43\x32\x30\x31\x35\x2d\x53\x74\x61\x67\x65\x32"

def decrypt(key):
    cipher = AES.new("".join(key), AES.MODE_OFB, iv)
    return cipher.decrypt(data)

def check_data(key, dec):
    if dec[0:2] == 'PK':
        for i in range(32):
            h = hashlib.sha256(dec[0:-i]).hexdigest()
            if h == "845f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187":
                with open("stage3.zip", "w") as f:
                    f.write(dec[0:-i])
                print("key : " + binascii.hexlify("".join(kl)))
                print("file decrypted (stage3.zip)")


for i in opts2:
    for j in opts3:
        kl = kparts1 + [i, j] + kparts2
        check_data(kl, decrypt(kl))

        kl.reverse()
        check_data(kl, decrypt(kl))
```

```
$ python bf-st2key.py
key : 9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac
file decrypted (stage3.zip)
```

La composition de la clé de déchiffrement finalement utilisée est décrite dans la figure ci-dessous.

FIGURE 2.2 – Composition de la clé de déchiffrement du stage3



2.3 Solution alternative

Avant d'explorer le stage 3, nous présentons dans cette courte section une solution alternative et vidéoludique pour retrouver la clé de déchiffrement cachée dans une map BSP. Le fichier *README* présent dans l'archive *sstic.pk3* nous livre quelques informations supplémentaires :

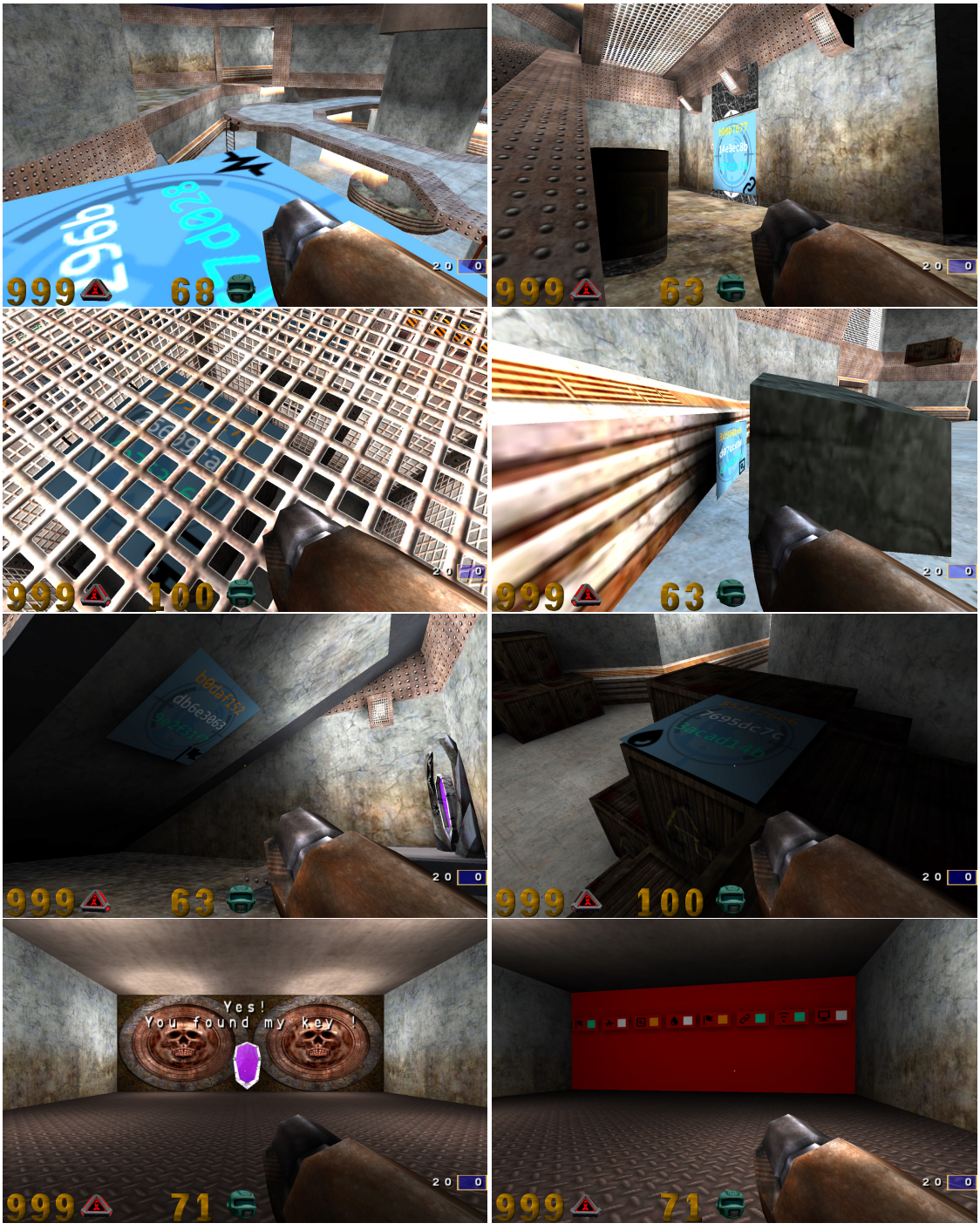
```
$ cat README
Copy the pk3 in your baseoa directory.
In the game, open the console (^2) and type \map sstic.
```

Un copié/collé de ces instructions dans un moteur de recherche nous envoie sur une page du wiki du jeu OpenArena, un *deathmatch FPS* open-source reposant sur le moteur Quake III Engine.

En maîtrisant quelques techniques de jeu avancées (comme le "Rocket Jump" pour pouvoir lire la partie de clé située sur une caisse en hauteur), il est possible d'explorer la map *sstic* et de trouver d'une part, les parties de clé, et d'autre part, la salle secrète dans laquelle sont affichés les sélecteurs retrouvés dans la section précédente. De plus, le jeu OpenArena permet l'utilisation de codes de triches, dont les suivants peuvent se révéler pratiques :

— l'entrée de `/give flight` dans la console permet de voler,
— l'entrée de `/noclip` permet de passer à travers les murs.

Ci-dessous, quelques captures d'écrans du jeu dans la map *sstic*.


999 68
999 63
999 100
999 63
999 63
999 100
Yes!
You found my key !
999 71
999 71

Stage 3 : protocole de souris USB

```
$ unzip stage3.zip
Archive:  stage3.zip
 extracting: encrypted
  inflating: memo.txt
  inflating: paint.cap
$ cat memo.txt
Cipher: Serpent-1-CBC-With-CTS
IV: 0x5353544943323031352d537461676533
Key: Well, definitely can't remember it... So this time I securely stored it with Paint.

SHA256: 6b39ac2220e703a48b3de1e8365d9075297c0750e9e4302fc3492f98bdf3a0b0 - encrypted
SHA256: 7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2 - decrypted
```

Nous faisons face cette fois encore à un fichier *encrypted*. Celui-ci a été chiffré à l'aide de l'algorithme Serpent1, en mode bloc CBC avec vol de texte CTS. Le vecteur d'initialisation nous est également fourni, et le mémo nous informe que la clé a été stockée dans le logiciel Paint [1], certainement sous la forme d'une image donc.

3.1 Analyse de la capture USB

```
$ file paint.cap
paint.cap: tcpdump capture file (little-endian) - version 2.4, capture length 262144)
```

Les fichiers CAP renferment des captures de données, et peuvent être visualisés dans le logiciel Wireshark. La figure 3.1 présente les premières trames du fichier *paint.cap* dans ce logiciel.

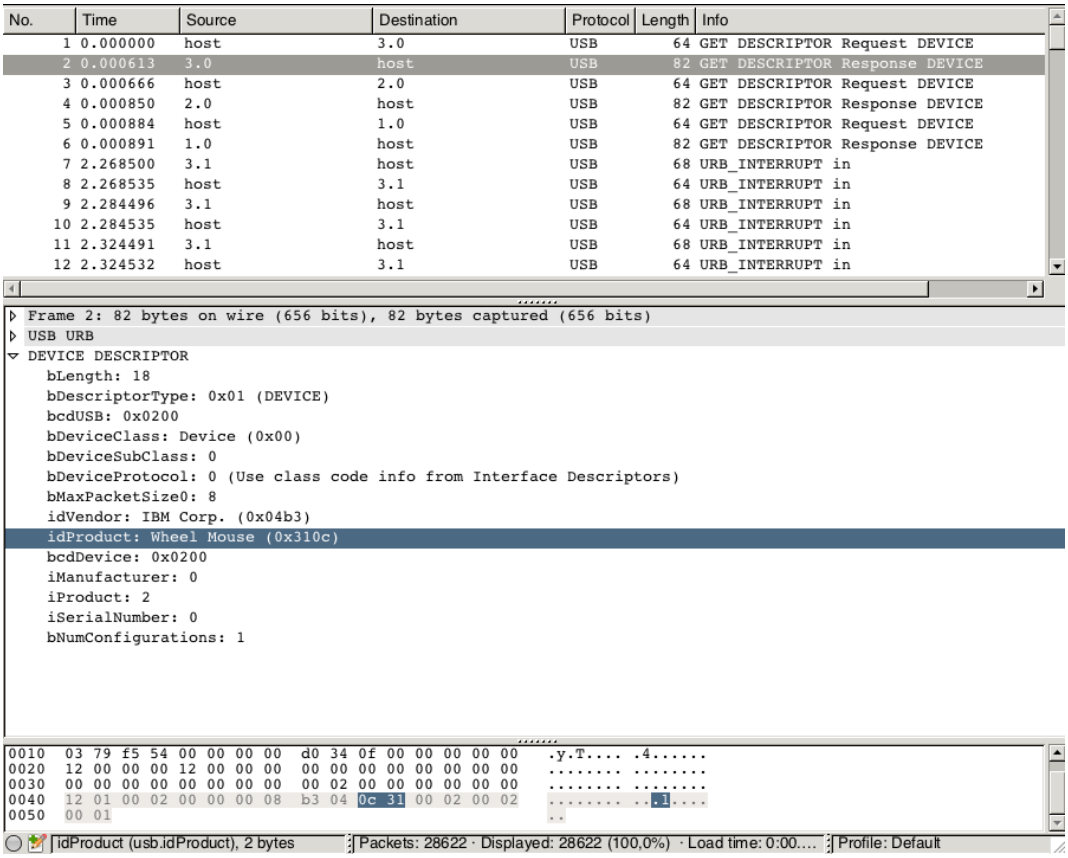
On identifie immédiatement la nature du protocole respecté par les données ici présentes, grâce à la colonne du même nom : il s'agit d'un protocole USB. Plus précisément, le fichier contient 6 paquets de type "GET DESCRIPTOR DEVICE", permettant d'établir le dialogue entre le dispositif USB et le système hôte, puis 28616 paquets de type "URB_INTERRUPT" véhiculant les données effectives.

Les 6 premiers paquets forment 3 couples requête/réponse. Le système interroge des dispositifs USB afin de mieux les identifier, et ainsi savoir avec quel pilote interpréter les *payloads* des interruptions suivantes.

Un premier dispositif signale qu'il est une souris USB (IBM, cf. figure 3.1), deux autres signalent qu'il s'agit de HUBs USB. C'est le premier dispositif qui nous intéresse ici : les communications

1. http://windows.microsoft.com/fr-fr/windows7/products/features/paint

FIGURE 3.1 – Capture d'écran de Wireshark - paquets `GET DESCRIPTOR`



interceptées vont nous permettrent de reproduire les mouvement (et les clics) de souris effectués sous Paint, et ainsi retrouver la clé de déchiffrement tracée dans Paint.

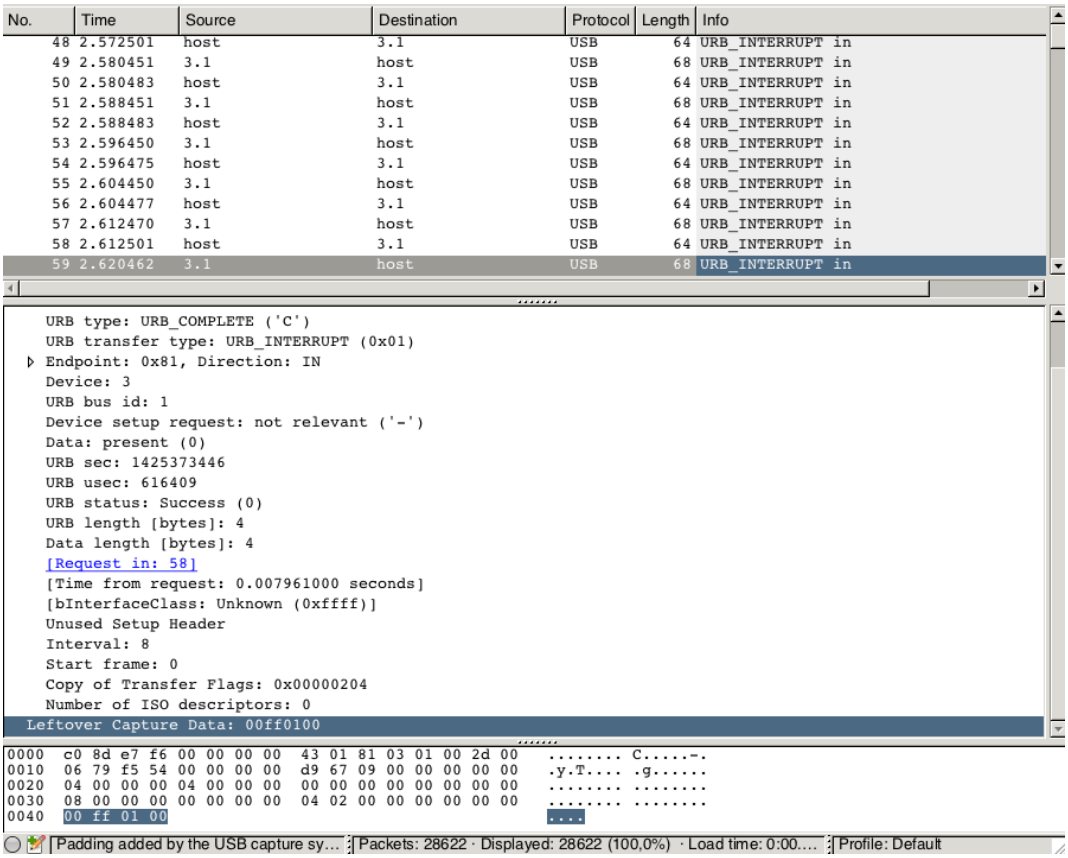
3.2 Protocole de communication des souris USB

La figure 3.2 présente cette fois les paquets de données de type URB_INTERRUPT. Un URB [2] (*USB Request Block*) est une structure de données décrivant les détails d'une requête d'un dispositif USB au système, ainsi que les détails de la réponse de ce dernier. Dans le cas d'une souris USB, un URB contenant les informations de déplacement et d'état des boutons est soumis à intervalle régulier.

Le champ marqué par l'étiquette "Leftover Capture Data" dans Wireshark contient 4 octets de données. La page Wikipedia sur les souris présente le découpage suivant :

2. https://msdn.microsoft.com/en-us/library/windows/hardware/ff537056%28v=vs.85%29.aspx

FIGURE 3.2 – Capture d'écran de Wireshark - paquets URB_INTERRUPT



	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Octet 1	$overflow_y$	$overflow_x$	$sign_y$	$sign_x$	1	MB	RB	LB
Octet 2	(unsigned char)δ_x							
Octet 3	(unsigned char)δ_y							

Les bits $overflow$ indiquent un dépassement de capacité tandis que les bits $sign$ informe sur le signe des valeurs de déplacement δ_x et δ_y. MB (*Middle Button*), RB (*Right Button*) et LB (*Left Button*) renseignent sur l'état des boutons de la souris (1 pour appuyé, 0 pour relâché).

Toutefois, cette représentation n'est pas satisfaisante puisqu'on ne retrouve que les valeurs suivantes dans la capture fournie :

— Pour l'octet 1 : 0 ou 1,
— Pour l'octet 2 : 0x01, 0x02, 0xFE ou 0xFF,
— Pour l'octet 3 : 0x01, 0x02, 0xFE ou 0xFF.

Notre souris ne se déplacerait toujours que vers un coin de l'écran puisque les bits de signe sont à 0 pour tous les paquets. De ces valeurs présentes, on infère que les octets 2 et 3 sont en fait

représentés directement signés en complément à deux. La disposition des données utiles serait donc la suivante :

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Octet 1	0	0	0	0	0	0	0	LB
Octet 2	(char)δ_x							
Octet 3	(char)δ_y							

Nous pouvons désormais tenter de reconstruire l'image. Pour cela, nous utilisons le module Turtle disponible en Python. Son but premier est de permettre la découverte de la programmation en offrant des primitives simples de déplacements d'une tortue dans une fenêtre, à la manière des langages de la famille Logo. Elle nous permettra ici de rejouer le déplacement de la souris, et ainsi reconstruire le dessin de la clé de déchiffrement tracée initiatialement dans Paint.

Listing 3.1– redraw.py : reconstruction de l'image Paint

```
import turtle
import struct

with open('paint.cap', 'r') as f:
    data = f.read()

turtle.penup()
turtle.setpos(-600, 0)
turtle.speed(0)

for i in range(0x27e, len(data), 164):
    byte1 = ord(data[i])
    byte2 = struct.unpack('b', data[i+1])[0]
    byte3 = struct.unpack('b', data[i+2])[0]

    (x,y) = turtle.position()

    if byte1 == 0x00:
        turtle.penup()
    if byte1 == 0x01:
        turtle.pendown()

    turtle.setpos(x+byte2, y-byte3)

input("waiting a key press")
```

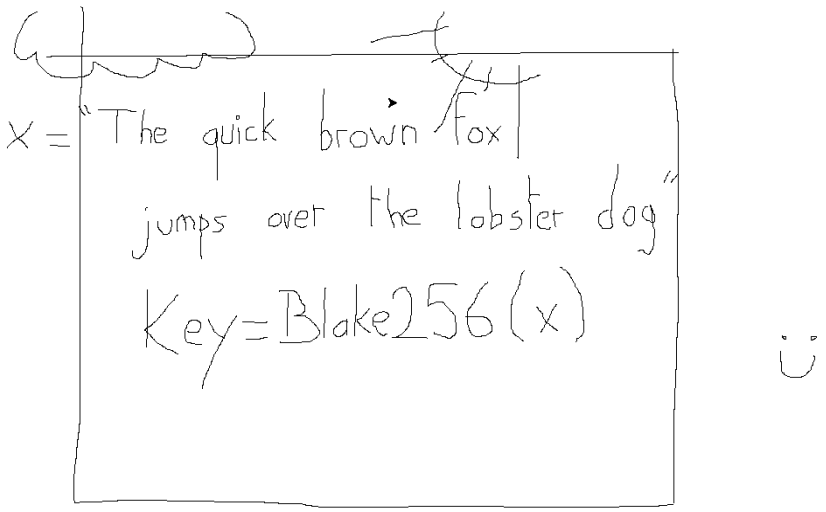
La figure 3.3 présente le dessin reconstruit. On y découvre l'information que nous recherchions : la chaîne de caractères "The quick brown fox jumps over the lobster dog" est passée en argument à une fonction `Blake256` pour former en retour la clé de déchiffrement.

3.3 Déchiffrement du fichier *encrypted*

L'algorithme Blake256 [3] est un algorithme de hachage de données. On utilise la librairie C libkripto disponible sur Github implémente cet algorithme.

3. http://en.wikipedia.org/wiki/BLAKE_%28hash_function%29

FIGURE 3.3 – Image reconstituée depuis la capture USB



Listing 3.2– blake-hash.c : hachage Blake256 de la clé de déchiffrement

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <kripto/hash.h>
#include <kripto/hash/blake256.h>

int main(int argc, char **argv)
{
        uint8_t hash[32];
        unsigned int i;

        kripto_hash_all(kripto_hash_blake256, 0, "The␣quick␣brown␣fox␣jumps␣over␣the␣lobster␣dog",
                  46, hash, 32);
        for(i = 0; i < 32; i++) printf("%.2x", hash[i]);
        putchar('\n');

        return 0;
}
```

```
$ gcc test/hash/blake256.c lib/libkripto.a -I include
$ ./a.out
66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d
```

Nous sommes désormais en possession de la clé de déchiffrement. La librarie C++ Crypto++ offre dans son API de nombreux algorithmes cryptographiques, ainsi que de nombreux modes de chiffrement par blocs. On y retrouve l'algorithme Serpent, et également l'enchaînement des blocs CBC avec vol de texte (CTR).

Listing 3.3– serpent-decrypt.cpp : déchiffrement du fichier *encrypted*

```
#include <cryptopp/serpent.h>
#include <cryptopp/serpentp.h>
#include <cryptopp/osrng.h>
#include <cryptopp/modes.h>
#include <cryptopp/hex.h>
#include <cryptopp/secblock.h>
#include <iostream>
#include <fstream>


int main(int argc, char **argv)
{
  using namespace CryptoPP;
  using namespace std;

  AutoSeededRandomPool prng;

  byte srcbytes[] = {0x66,0xc1,0xba,0x5e,0x8c,0xa2,0x9a,0x8a,
                     0xb6,0xc1,0x05,0xa9,0xbe,0x9e,0x75,0xfe,
                     0x0b,0xa0,0x79,0x97,0xa8,0x39,0xff,0xea,
                     0xe9,0x70,0x0b,0x00,0xb7,0x26,0x9c,0x8d};
  SecByteBlock key(sizeof(srcbytes));
  byte* dstbytes = key.BytePtr();
  for (int i = 0 ; i < 32 ; i++)
    dstbytes[i] = srcbytes[i];

  byte iv[Serpent::BLOCKSIZE] = {0x53,0x53,0x54,0x49,0x43,0x32,0x30,0x31,
                                 0x35,0x2d,0x53,0x74,0x61,0x67,0x65,0x33};

  ifstream ifs("encrypted");
  string cipher((istreambuf_iterator<char>(ifs)),
                (istreambuf_iterator<char>()));
  string recovered;

  try
  {
    CBC_CTS_Mode< Serpent >::Decryption d;
    d.SetKeyWithIV(key, key.size(), iv);

    StringSource ss3(cipher, true,
      new StreamTransformationFilter(d,
        new StringSink(recovered)
      )
    );

    cout << recovered;
  }
  catch(const CryptoPP::Exception& e)
  {
    cerr << e.what() << endl;
    exit(1);
  }

  return 0;
}
```

Il ne reste qu'à compiler et exécuter, puis à vérifier l'intégrité du fichier ainsi obtenu.

```
$ g++ serpent-decrypt.cpp -I/opt/local/include -lcryptopp -L/opt/local/lib
$ ./a.out > stage4.zip
$ shasum -a 256 stage4.zip
7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2  stage4.zip
```

Stage 4 : désobfuscation de code Javascript

```
$ unzip stage4.zip
Archive:  stage4.zip
  inflating: stage4.html
```

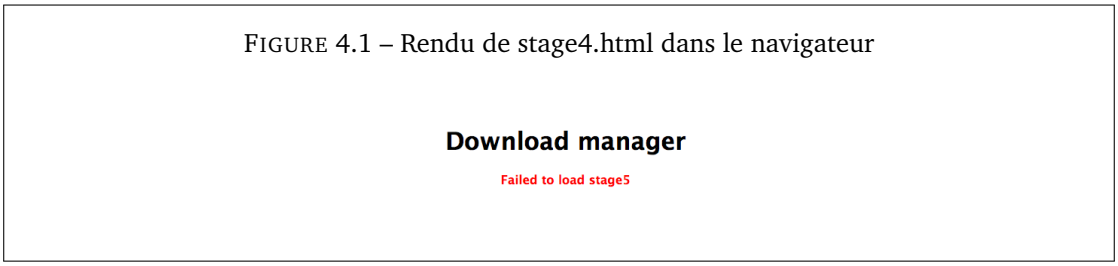
L'archive ne contient pas de mémo, mais un unique fichier HTML, dont le code source est le suivant :

```
<html>
<head>
<style>
        * { font-family: Lucida Grande,Lucida Sans Unicode,Lucida Sans,Geneva,Verdana,sans-serif;
      text-align:center; }
        #status { font-size: 16px; margin: 20px; }
        #status a { color: green; }
        #status b { color: red; }
</style>
</head>
<body>
        <script>
                var data = "2b1f25cf8db5d243f59b065da6b56753b72e28f16eb271b9ad1[...]";
                var hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c";
                $=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_:(![]+"")[$],_$_:++$,$_$$:({}+"")
    [$],$$_$:($[$]+"")[$],_$$:++$,$$$_:(!""+"")[$],$__:++$,$_$:++$,$$__:({}+"")[$],$$_:++$,$$$:
    ++$,$___:++$,$__$:++$};$.$_=($.$_=$+"")[$.$_$]+($._$=$.$_[$.__$])+($.$$=($.$+"")[$.__$])+((
    !$)+"")[$._$$]+($.__=$.$_[$.$$_])+($.$=(!""+"")[$.__$])+($._=(!""+"")[$._$_])+$.$_[$.$_$]+$
    .__+$._$+$.$;$.$$=$.$+(!""+"")[$._$$]+$.__+$._+$.$+$.$$;$.$=($.___)[$.$_][$.$_];$.$($.$($.$
    [...]
        </script>
</body>
</html>
```

Nous tentons de l'ouvrir dans un navigateur, et obtenons le rendu de la figure 4.1.

Nous étudions donc le code Javascript présent dans la page, afin de comprendre pourquoi le stage 5 n'est pas chargé, et d'y remédier. Cette étude commence par une étape de désobfuscation du script.

FIGURE 4.1 – Rendu de stage4.html dans le navigateur



4.1 Désobfuscation du Javascript

Nous commençons par réindenter le code Javascript. De nombreux outils permettent d'automatiser cette tâche ; nous utilisons JSBeautifier [1]. Le résultat est un script constitué de 6 instructions seulement (nous les avons numérotés dans le listing ci-dessous). La première de ces instructions ainsi que les nombreuses occurences des caractères `$`, _ et + guident notre recherche sur le web : le chargeur du stage 5 a été obfusqué à l'aide de l'outil JJEncode [2] [3]. Nous montrons dans cette section comment fonctionne ce type d'encodage, en le décodant "manuellement" étape par étape. Il est toutefois également possible de le décoder automatiquement à l'aide d'outils [4] [5].

```
1: $ = ~[];
2: $ = {
       ___: ++$,
       $$$$: (![] + "")[$],
       __$: ++$,
       $_$_: (![] + "")[$],
       _$_: ++$,
       $_$$: ({} + "")[$],
       $$_$: ($[$] + "")[$],
       _$$: ++$,
       $$$_: (!"" + "")[$],
       $__: ++$,
       $_$: ++$,
       $$__: ({} + "")[$],
       $$_: ++$,
       $$$: ++$,
       $___: ++$,
       $__$: ++$
   };
3: $.$_ = ($.$_ = $ + "")[$.$_$] + ($._$ = $.$_[$.__$]) + ($.$$ = ($.$ + "")[$.__$]) + ((!$) + "")[
   $._$$] + ($.__ = $.$_[$.$$_]) + ($.$ = (!"" + "")[$.__$]) + ($._ = (!"" + "")[$._$_]) + $.$_[$.$
   _$] + $.__ + $._$ + $.$;
4: $.$$ = $.$ + (!"" + "")[$._$$] + $.__ + $._ + $.$ + $.$$;
5: $.$ = ($.___)[$.$_][$.$_];
6: $.$($.$($.$$ + "\"" + "__=" + $.$$_$ + $._$ + $.$$__ + $._ + "\\" + [...])())();
```

Javascript est un langage permissif, notamment en ce qui concerne le nommage des variables. Par exemple, les caractères '$' ou '_' peuvent former des noms de variables légaux, même situés au début du nom. Ainsi, la première instruction n'est autre qu'un affectation de la négation bit à bit d'un tableau vide (en Javascript, -1) à la variable `$`.

Durant la seconde instruction, cette variable est utilisée (et modifiée par des incrémentations successives) afin de construire un ensemble de valeurs. Ces valeurs sont affectées aux champs aux noms toujours aussi exotiques d'un objet, lui-même placé finalement dans la variable `$`. L'objet contient finalement les valeurs de l'ensemble des caractères hexadécimaux :

1. http://jsbeautifier.org/
2. http://utf-8.jp/public/jjencode.html
3. http://pferrie2.tripod.com/papers/jjencode.pdf
4. https://github.com/jacobsoo/Decoder-JJEncode
5. https://github.com/crackinglandia/python-jjdecoder

```
$ = {
  ___  : 0,
  $$$$ : "f",
  __$  : 1,
  $_$_ : "a",
  _$_  : 2,
  $_$$ : "b",
  $$_$ : "d",
  _$$  : 3,
  $$$_ : "e",
  $__  : 4,
  $_$  : 5,
  $$__ : "c",
  $$_  : 6,
  $$$  : 7,
  $___ : 8,
  $__$ : 9
};
```

On peut désormais substituer certaines variables à leur valeur dans la suite du script, ce qui donne pour l'instruction 3 :

```
$.$_ = ($.$_ = $ + "")[5] + ($._$ = $.$_[1]) + ($.$$ = ($.$ + "")[1]) + ((!$) + "")[3] + ($.__ = $.
$_[6]) + ($.$ = (!"" + "")[1]) + ($._ = (!"" + "")[2]) + $.$_[5] + $.__ + $._$ + $.$;
```

Les instructions 3 et 4 construisent donc également, par un jeu de concaténations de chaînes et de conversions de types, de nouvelles valeurs, qui viennent s'ajouter dans l'objet `$` :

```
$ = {
  ___  : 0,
  $$$$ : "f",
  __$  : 1,
  $_$_ : "a",
  _$_  : 2,
  $_$$ : "b",
  $$_$ : "d",
  _$$  : 3,
  $$$_ : "e",
  $__  : 4,
  $_$  : 5,
  $$__ : "c",
  $$_  : 6,
  $$$  : 7,
  $___ : 8,
  $__$ : 9,
  $_   : "constructor",
  _$   : "o",
  $$   : "n",
  __   : "t",
  _    : "u",
  $$   : "return",
};
```

Ensuite, l'instruction 5 une fois réécrite correspond à :

```
$.$ = (0)["constructor"]["constructor"];
```

Il s'agit là d'une manière de créer dynamiquement une fonction Javascript. La variable $.$ appliquée à une chaîne de caractères retournera une fonction dont le corps sera dicté par cette chaîne. C'est ce que fait l'instruction 6, en construisant une chaîne de caractères à partir de caractères littéraux en notation hexadécimale et des champs de l'objet `$`, puis en l'appliquant à l'aide de `()`. Le listing suivant présente le contenu de cette chaîne reconstruit et réindenté :

```
__ = document;
$$$ = 'stage5';
$$_$ = 'load';
$_$$ = ' ';
_$$$$ = 'user';
_$$$ = 'div';
$$_$$$ = 'navigator';
$$_$$ = 'preferences';
```

```
$_$$$ = 'to';
$$$_$ = 'href';
$$$$_ = '=';
$$$$$ = 'chrome';
_$$$$ = '"';
$_$$$$ = 'Agent';
$$$_$$ = 'down';
$$$$_$ = 'import';
$ = '<b>Failed' + $_$$ + $_$$$ + $_$$ + $$_$ + $_$$ + $$$ + '</b>';
___ = 'write';
____ = 'getElementById';
_$_ = "raw";
$$ = window;
__$ = $$.crypto.subtle;
__$_ = 'decrypt';
___$ = 'status';
$____ = $$$$_$ + 'Key';
_______ = 0;
__$__ = 'then';
_$____ = 'digest';
__$___ = 'innerHTML';
___$__ = {
    name: 'SHA-1'
};
____$_ = data;
_____$ = hash;
_$_____ = Blob;
___$___ = URL;
____$__ = 'createObjectURL';
______$ = 'type';
$_______ = 'application/octet-stream';
_$______ = 'name';
__$_____ = 'AES-CBC';
___$____ = 'iv';
____$___ = '<a' + $_$$ + $$$_$ + $$$$_ + _$$$$;
_____$__ = '"' + $_$$ + $$$_$$ + $$_$ + $$$$_ + _$$$$ + $$$ + '.zip' + _$$$$ + '>' + $$$_$$ + $$_$
+ $_$$ + $$$ + '</a>';
______$_ = '(';
_______$ = ')';
$________ = 'setTimeout';
_________ = parseInt;
__________ = $$[$$_$$$][_$$$$$ + $_$$$$];
___________ = 'length';
____________ = 'substr';
_____________ = _______ + 1;
______________ = ______________ * 2;
_______________ = ______________ * 4;
________________ = ________________ * ________________;
$_$ = 125 * ________________;
_________________ = 'indexOf';
__________________ = 'charCodeAt';
___________________ = 'push';
____________________ = Uint8Array;
_____________________ = '';
______________________ = 'byteLength';
_______________________ = $_$$$ + 'String';
__[___]('<h' + _____________ + '>Down' + $$_$ + $_$$ + 'manager</h' + _____________ + '>');
__[___]('<' + _$$$ + $_$$ + 'id' + $$$$_ + _$$$$ + ___$ + _$$$$ + '><i>' + $$_$ + 'ing...</i></' +
_$$$ + '>');
__[___]('<' + _$$$ + $_$$ + 'style' + $$$$_ + _$$$$ + 'display:none' + _$$$$ + '><a' + $_$$ + 'tar
get' + $$$$_ + _$$$$ + 'blank' + _$$$$ + $_$$ + $$$_$ + $$$$_ + _$$$$ + $$$$$ + '://browser/conten
t/' + $$_$$ + '/' + $$_$$ + '.xul' + _$$$$ + '>Back' + $_$$ + $_$$$ + $_$$ + $$_$$ + '</a></' + _$
$$ + '>');

function _____(______) {
    _ = [];
    for (___________ = _______; ___________ < ______[___________]; ++___________) _[_____________
    _______](______[_____________](___________));
    return new _____________________(_);
}

function _____________________(______) {
    _ = [];
```

```
    for (____________ = ________; ____________ < _______[_____________] / _________________; ++__________
    _) _[_____________________](__________(_______[_____________](____________ * _________________, ____
    ___________), ____________________));
    return new ____________________________(_);
}

function ____________________________(_________) {
    _______ = ________________________________;
    for (____________ = ________; ____________ < _________[_________________________________]; ++___________)
    {
        ___ = _________[____________][_________________________________](_____________________);
        if (___[_____________] < ___________________) _______ += ________;
        _______ += ___;
    }
    return _______;
}

function ___________________________________() {
    $_ = ______(___________[_____________](___________[_____________________](_______$_) + _________________
    , ____________________));
    _$__ = ______(___________[_____________](___________[_____________________](________$) - _____________
    _____, ____________________));
    _$ = {};
    _$[_$_______] = __$______;
    _$[___$_____] = $_;
    _$[_____________] = _$__[_____________] * ____________________;
    __$[$_____](_$_, _$__, _$, false, [__$_])[__$__](function(_$___) {
        __$[__$_](_$, _$___, _____________________(_____$_))[__$__](function(___$_) {
            _____$ = new ____________________________(___$_);
            __$[_$_____](___$__, _____$)[__$__](function(_____$$) {
                if (______$ == ____________________________(new ____________________________(_____$$))) {
                    ______$_ = {};
                    ______$_[_______$] = $________;
                    ______$ = new _$______([_____$], ______$_);
                    __$_____ = ___$____[_____$__](______$);
                    __[_____](___$)[__$___] = _____$___ + __$_____ + ______$__;
                } else {
                    __[_____](___$)[__$___] = $;
                }
            });
        }).catch(function() {
            __[_____](___$)[__$___] = $;
        });
    }).catch(function() {
        __[_____](___$)[__$___] = $;
    });
}
$$[$_________](___________________________________, $_$);
```

Avant l'encodage JJEncode, une première passe d'obfuscation avait été appliquée. Celle-ci consiste en la déclaration des objets utiles au script dans plusieurs variables aux noms composés uniquement des caractères `$` et `_`. Après remplacement des variables globales par leur valeur, et la réécriture de certaines expressions (`e['name']` remplacés par `e.name` par exemple), nous renommons également les fonctions et leurs variables locales selon le rôle qu'ils semblent jouer. Le listing 4.1 présente la version finalement réécrite du script Javascript, replongée dans son contexte HTML initial.

Listing 4.1– st4-desobf.html : code source HTML désobfusqué

```
<html>
<head>
<style>
        * { font-family: Lucida Grande,Lucida Sans Unicode,Lucida Sans,Geneva,Verdana,sans-serif;
      text-align:center; }
        #status { font-size: 16px; margin: 20px; }
        #status a { color: green; }
        #status b { color: red; }
</style>
</head>
<body>
```

```
<script>
var data = "2b1f25cf8db5d243f59b065da6b56753b72e28f16eb271b9ad1[...]";
var hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c";
document.write('<h1>Download manager</h1>');
document.write('<div id="status"><i>loading...</i></div>');
document.write('<div style="display:none"><a target="blank" href="chrome://browser/content/prefere
nces/preferences.xul">Back to preferences</a></div>');

function strToByteArray(s) {
    a = [];
    for (i = 0 ; i < s.length ; ++i)
      a.push(s.charCodeAt(i));
    return new Uint8Array(a);
}

function unhexify(s) {
    a = [];
    for (i = 0 ; i < s.length / 2 ; ++i)
      a.push(parseInt(s.substr(i * 2, 2), 16));
    return new Uint8Array(a);
}

function hexify(a) {
    s = '';
    for (i = 0 ; i < a.byteLength ; ++i) {
        u3 = a[i].toString(16);
        if (u3.length < 2) s += 0;
        s += u3;
    }
    return s;
}

function decryptStage5() {
    iv = strToByteArray(window.navigator.userAgent.substr(
              window.navigator.userAgent.indexOf('(') + 1, 16));
    key = strToByteArray(window.navigator.userAgent.substr(
              window.navigator.userAgent.indexOf(')') - 16, 16));
    algo = { name:   'AES-CBC',
             iv:     iv,
             length: key.length * 8};
    window.crypto.subtle.importKey('raw', key, algo, false, ['decrypt']).then(function(key) {
        window.crypto.subtle.decrypt(algo, key, unhexify(data))['then'](function(plain) {
            plainBytes = new Uint8Array(plain);
            window.crypto.subtle.digest({ name: 'SHA-1' }, plainBytes).then(function(plainHash) {
                if (hash == hexify(new Uint8Array(plainHash))) {
                    hash = new Blob([plainBytes], {type: 'application/octet-stream'});
                    blob = URL.createObjectURL(hash);
                    document.getElementById('status').innerHTML =
                        '<a href="' + blob + '" download="stage5.zip">download stage5</a>';
                } else {
                    document.getElementById('status').innerHTML = '<b>Failed to load stage5</b>';
                }
            });
        }).catch(function() {
            document.getElementById('status')innerHTML = '<b>Failed to load stage5</b>';
        });
    }).catch(function() {
        document.getElementById('status')innerHTML = '<b>Failed to load stage5</b>';
    });
}

window.setTimeout(decryptStage5, 1000);
</script>
</body>
</html>
```

Nous pouvons désormais étudier le comportement du script de cette page. La variable `data` contient en fait ce qui correspondrait au contenu du fichier *encrypted* dans les autres stages, en représentation hexadécimale. La fonction principale `decryptStage5`, appelée toutes les secondes, tente de déchiffrer cette donnée par l'algorithme AES en mode bloc CBC, puis vérifie si le condensat SHA-1 du text clair obtenu est égal au contenu de la variable `hash`.

4.2 Recherche de la clé de déchiffrement et de l'IV

Comme pour les autres épreuves, il s'agit désormais de retrouver la clé de déchiffrement des données du stage suivant. Les deux lignes du script auxquelles on s'intéresse maintenant sont les suivantes :

```
[...]
iv = strToByteArray(window.navigator.userAgent.substr(
        window.navigator.userAgent.indexOf('(') + 1, 16));
key = strToByteArray(window.navigator.userAgent.substr(
        window.navigator.userAgent.indexOf(')') - 16, 16));
[...]
```

Elles indiquent que le vecteur d'initialisation ainsi que la clé pour le déchiffrement sont extraites depuis le *User-Agent* du visiteur de la page. Plus précisément :

— L'IV est formé des 16 caractères après la première parenthèse ouvrante.

— La clé est formée des 16 caractères avant la première parenthèse fermante.

Un premier indice concernant le *User-Agent* à trouver est situé dans le début du script. L'instruction :

```
document.write('<div style="display:none"><a target="blank" href="chrome://browser/content/preferen
ces/preferences.xul">Back to preferences</a></div>');
```

ajoute à la page un lien, caché par CSS, vers la page `chrome://browser/content/preferences/preferences.xul`. L'auteur de la page (et donc la personne ayant le bon *User-Agent* pour déchiffrer son contenu) utilise le navigateur Firefox : contrairement à ce que laisse penser le début de cette url, il s'agit d'une adresse supportée exclusivement par Firefox [6].

La documentation de Mozilla indique la forme des *User-Agent* du navigateur :

`Mozilla/5.0 (`**`platform`**`; rv:`**`geckoversion`**`) Gecko/`**`geckotrail`** `Firefox/`**`firefoxversion`**

Il nous faut désormais trouver les deux *tokens platform* et *geckoversion*.

Le second indice concerne la librarie cryptographique utilisée dans le script. Il s'agit de la librarie SubtleCrypto, dont la documentation nous informe qu'elle n'est supportée que depuis la version 34 de Firefox.

Enfin, un dernier indice réside dans la feuille de style de la page HTML. Celle-ci spécifie les fontes à utiliser de préférence pour le rendu, entre autres "Lucida Grande" et "Geneva", deux fontes initialement designées pour le système d'exploitation Mac OS.

Nous disposons d'indices pour réduire la recherche à quelques dizaines de possibilités seulement :

— Le *token platform* peut avoir pour valeur "Macintosh ; Intel Mac OS X x.y" ou "Macintosh ; PPC Mac OS X x.y", avec $x.y$ différents numéros de versions.

— La version *geckoversion* semble située entre 34.0 (la première à supporter SubtleCrypto) et 37.0 (la plus récente à ce jour).

Puisque le script Javascript nous a révélé que le fichier déchiffré serait un fichier ZIP, nous utilisons la même technique de vérification de clé que pour le stage 2 : le nombre magique "PK" est d'abord recherché en début de fichier, puis le fichier est haché plusieurs fois en enlevant jusqu'à 16 octets de potentiel *padding*.

6. http://kb.mozillazine.org/Chrome_URLs

Listing 4.2– bf-st4ua.py : recherche du User-Agent

```
from Crypto.Cipher import *
import hashlib

with open('encrypted', 'r') as f:
  data = f.read()

def check_ua(platform_tokens, feature_tokens, revision_tokens):
  for pt in platform_tokens:
    for ft in feature_tokens:
      for rt in revision_tokens:
        uapart = pt + "; " + ft + "; " + rt

        key = uapart[-16:]
        iv  = uapart[:16]
        assert(len(key) == 16)
        assert(len(iv) == 16)

        cipher = AES.new(key, AES.MODE_CBC, iv)
        plain  = cipher.decrypt(data)

        if plain[0:2] == 'PK':
          for i in range(16):
            h = hashlib.sha1(plain[0:-i]).hexdigest()
            if h == "08c3be636f7dffd91971f65be4cec3c6d162cb1c":
              print("found a matching user-agent: (" + uapart + ")")

revision_tokens = []
for x in range(34,38):
  for y in range(10):
      revision_tokens.append("rv:" + str(x) + "." + str(y))

platform_tokens = ["Macintosh"]
feature_tokens = []
for i in range(8, 11):
  for j in range(10):
    feature_tokens.append("Intel Mac OS X " + str(i) + "." + str(j))
    feature_tokens.append("PPC Mac OS X " + str(i) + "." + str(j))
check_ua(platform_tokens, feature_tokens, revision_tokens)
```

```
$ python bf-st4ua.py
found a matching user-agent: (Macintosh; Intel Mac OS X 10.6; rv:35.0)
$
```

Le fichier *stage5.zip* peut tout à fait être sauvegardé durant la recherche de la clé dans le script précédent. Toutefois, nous choisissons de recharger la page initiale *stage4.html* dans Firefox, en ayant préalablement modifié notre *User-Agent* à l'aide de l'extension User-Agent Switcher[7]. Le rendu est le suivant :

FIGURE 4.2 – Rendu de stage4.html dans le navigateur (avec le User-Agent modifié)

7. https://addons.mozilla.org/fr/firefox/addon/user-agent-switcher/

Stage 5 : rétro-ingénierie de code pour microcontrôleur ST20

```
$ unzip stage5.zip
Archive:  stage5.zip
  inflating: input.bin
  inflating: schematic.pdf
$
```

Le document *schematic.pdf* affiche le schéma de la figure 5.1 ainsi que les informations brutes suivantes :

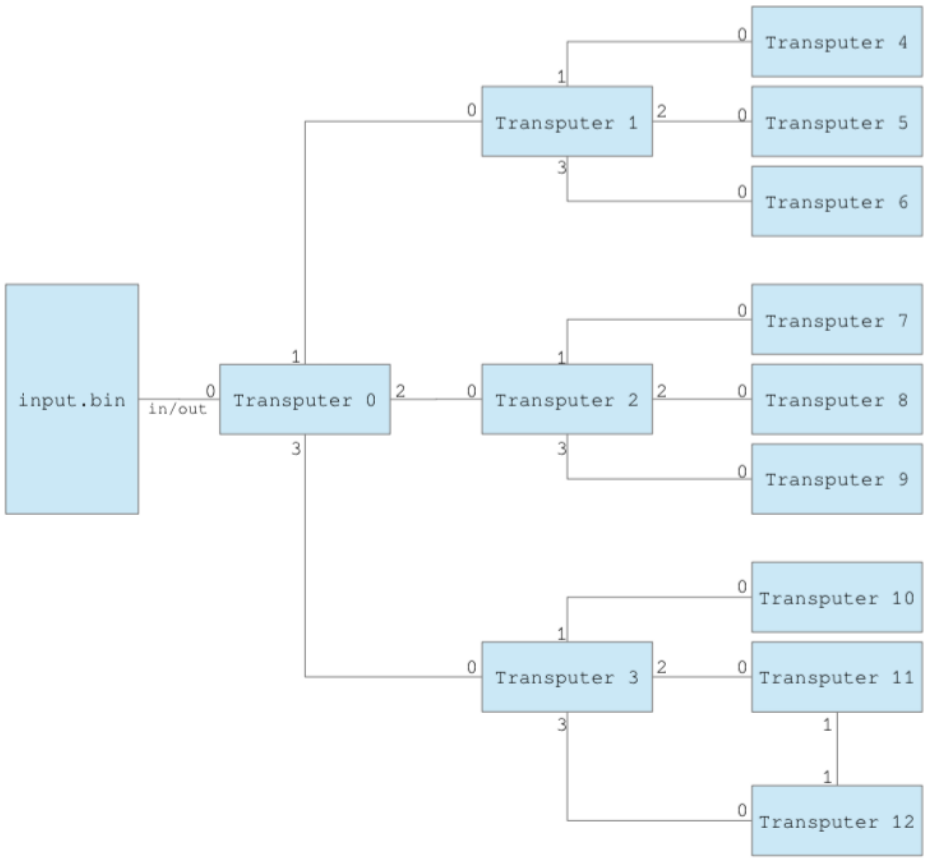
```
SHA256:
    a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81 - encrypted
    9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321 - decrypted

Test vector:
    key = "*SSTIC-2015*"
    data = "1d87c4c4e0ee40383c59447f23798d9fefe74fb82480766e".decode("hex")
    decrypt(key, data) == "I love ST20 architecture"
```

Le fichier *encrypted* mentionné ne figure pas dans l'archive. Une simple recherche d'une sous-partie de *input.bin* ayant le condensat SHA-256 indiqué permet de le localiser à la fin de ce fichier :

```
$ python
[...]
>>> import hashlib
>>> with open('input.bin', 'r') as f:
...   data = f.read()
...
>>> for i in range(0, len(data)):
...   h = hashlib.sha256(data[i:]).hexdigest()
...   if h == "a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81":
...     print("encrypted found at offset " + str(i))
...     break
...
encrypted found at offset 2477
>>>
```

Enfin, la phrase déchiffrée du vecteur de test ne peut être plus explicite : il s'agit là d'un programme pour architecture ST20, de laquelle nous présentons les spécificités dans la section suivante.

FIGURE 5.1 – Schéma des transputers (*schematic.pdf*)



5.1 Présentation de l'architecture ST20

Les microcontrôleurs ST20 implémentent des transputers. Ces unités de calcul ont été conçues pour réaliser des machines parallèles : plusieurs processus communiquent par échange de messages, et l'algorithme d'ordonnancement est directement implémenté en dur par le microcontrôleur. Chaque processus dispose de son propre espace mémoire, appelé *workspace*, et n'a besoin que d'un ensemble de registres limité lorsqu'il est exécuté. Ces registres sont listés dans le tableau suivant :

Areg Breg Creg	Registres de calcul formant une pile LIFO
Iptr	Pointeur d'instruction
Wptr	Adresse de l'espace de travail (*Workspace*) courant
Status	Registre de status

Les registres Areg, Breg et Creg ont la particularité de fonctionner comme une pile LIFO de taille 3 : lorsque un nouveau résultat issu d'un chargement ou d'un calcul est stocké dans Areg, l'ancien contenu de Breg est sauvé dans Creg et celui de Areg, dans Breg. Au démarrage, ou bien lorsque des données sont dépilées, ces registres ont pour valeur *undefined*.

Les registres stockent des données sur 4 octets. C'est la même taille que les données manipulées par les instructions rencontré dans les programmes fournis, à l'exception de quelques traitements octet par octet.

5.2 Désassemblage et découpage de *input.bin*

Nous trouvons des documentations pour deux architectures ST20 différentes : l'une concerne le ST20 C1 tandis que la seconde concerne les ST20 C2/C4. Bien que très proches, les deux architectures ne partagent pas exactement le même jeu d'instructions, et surtout, pas les mêmes *opcodes* pour certaines instructions en commun. Après une première tentative de désassemblage "à la main" en suivant les instructions de la variante C1, le programme assembleur obtenu ne semble pas constituer un programme correct. Nous décidons donc de nous concentrons sur la variante C2/C4.

Le code du désassembleur développé dans le cadre de ce challenge figure en annexe B. Voici les premières instructions désassemblées du fichier *input.bin* :

Listing 5.1– input.txt : désassemblage de *input.bin*

```
00000000  f8                  prod

00000001  64 b4               ajw 0x-4c
00000003  40                  ldc 0x0
00000004  d1                  stl 0x1      ; var_1
00000005  40                  ldc 0x0
00000006  d3                  stl 0x3      ; var_3
00000007  24 f2               mint
00000009  24 20 50            ldnlp 0x400
0000000c  23 fc               gajw

0000000e  64 b4               ajw 0x-4c
00000010  2c 49               ldc 0xc9
00000012  21 fb               ldpi         ; lbl_dd
00000014  24 f2               mint
00000016  48                  ldc 0x8
00000017  fb                  out

00000018  24 19       lbl_18: ldlp 0x49    ; var_73 ptr
0000001a  24 f2               mint
0000001c  54                  ldnlp 0x4
0000001d  4c                  ldc 0xc
0000001e  f7                  in
0000001f  24 79               ldl 0x49     ; var_73
00000021  21 a5               cj 0x15      ; lbl_38
00000023  2c 4d               ldc 0xcd
00000025  21 fb               ldpi         ; lbl_f4
00000027  24 f2               mint
00000029  54                  ldnlp 0x4
```

```
0000002a  24 79                ldl 0x49     ; var_73
0000002c  f7                   in
0000002d  2c 43                ldc 0xc3
0000002f  21 fb                ldpi         ; lbl_f4
00000031  24 7a                ldl 0x4a     ; var_74
00000033  24 79                ldl 0x49     ; var_73
00000035  fb                   out
00000036  61 00                j 0x-20      ; lbl_18

00000038  24 19       lbl_38:  ldlp 0x49    ; var_73 ptr
0000003a  24 f2                mint
0000003c  51                   ldnlp 0x1
0000003d  4c                   ldc 0xc
0000003e  fb                   out
0000003f  24 19                ldlp 0x49    ; var_73 ptr
00000041  24 f2                mint
00000043  52                   ldnlp 0x2
00000044  4c                   ldc 0xc
00000045  fb                   out
00000046  24 19                ldlp 0x49    ; var_73 ptr
00000048  24 f2                mint
0000004a  53                   ldnlp 0x3
0000004b  4c                   ldc 0xc
0000004c  fb                   out
0000004d  29 44                ldc 0x94
0000004f  21 fb                ldpi         ; lbl_e5
00000051  24 f2                mint
00000053  48                   ldc 0x8
00000054  fb                   out
00000055  12                   ldlp 0x2     ; var_2 ptr
00000056  24 f2                mint
00000058  54                   ldnlp 0x4
00000059  44                   ldc 0x4
0000005a  f7                   in
0000005b  15                   ldlp 0x5     ; var_5 ptr
0000005c  24 f2                mint
0000005e  54                   ldnlp 0x4
0000005f  4c                   ldc 0xc
00000060  f7                   in
00000061  28 48                ldc 0x88
00000063  21 fb                ldpi         ; lbl_ed
00000065  24 f2                mint
00000067  48                   ldc 0x8
00000068  fb                   out
00000069  13                   ldlp 0x3     ; var_3 ptr
0000006a  24 f2                mint
0000006c  54                   ldnlp 0x4
0000006d  41                   ldc 0x1
0000006e  f7                   in
0000006f  19                   ldlp 0x9     ; var_9 ptr
00000070  24 f2                mint
00000072  54                   ldnlp 0x4
00000073  13                   ldlp 0x3     ; var_3 ptr
00000074  f1                   lb
00000075  f7                   in
00000076  40                   ldc 0x0
00000077  d4                   stl 0x4      ; var_4
00000078  11          lbl_78:  ldlp 0x1     ; var_1 ptr

[...]

000000db  66 0b                j 0x-65      ; lbl_78

000000dd  42 6f 6f 74 20 6f 6b 00     db "Boot ok"
000000e5  43 6f 64 65 20 4f 6b 00     db "Code ok"
000000ed  44 65 63 72 79 70 74 00     db "Decrypt"

000000f5  24 bc       lbl_f5:  ajw 0x4c
000000f7  22 f0                ret
000000f9  71                   ldl 0x1      ; var_1
000000fa  0                    nop
000000fb  0                    nop
000000fc  0                    nop
```

```
000000fd  4                   j 0x4        ; lbl_102
000000fe  0                   nop
000000ff  0                   nop
00000100  80                  adc 0x0
00000101  0                   nop
00000102  0           lbl_102:  nop
00000103  0                   nop
00000104  0                   nop
00000105  70                  ldl 0x0      ; var_0
00000106  60 b8               ajw 0x-8
00000108  24 f2               mint
0000010a  24 20 50            ldnlp 0x400
0000010d  23 fc               gajw
0000010f  60 b8               ajw 0x-8
00000111  15                  ldlp 0x5     ; var_5 ptr
00000112  24 f2               mint
00000114  54                  ldnlp 0x4
00000115  4c                  ldc 0xc
00000116  f7                  in
00000117  75                  ldl 0x5      ; var_5

[...]
```

La première instruction réalise le produit de deux valeurs non définies. Nous expliquons sa présence dans un prochain paragraphe.

Le bloc suivant (de 0x01 à 0x0c) correspond à l'initialisation du transputer.

Ensuite, à l'adresse 0x17, la chaîne "Boot ok" est envoyée sur le canal situé à l'adresse 0x80000000. On peut penser que cette chaîne est destinée à l'affichage. Ainsi, 0x80000000 stockerait le canal 0 correspondant à l'*input* (*input.bin*) / *output* (affichage) du schéma 5.1.

Une première boucle est ensuite exécutée (de 0x18 à 0x36). A chacun de ses tours, le transputer :

— reçoit 12 octets dont les 4 premiers forment une taille sz et les 4 suivants, une adresse $addr$,
— reçoit sz octets et les écrits à l'adresse `lbl_f4`,
— envoie ces octets vers l'adresse $addr$.

La boucle est interrompue lorsque la taille sz reçue est égale à 0. Le dernier bloc lu est propagé (de 0x38 à 0x4c) vers 3 adresses différentes : celles aux adresses 0x80000004, 0x80000008 et 0x8000000c.

Suit un nouvel enchaînement de lectures / écritures depuis *in* / *out* :

— écriture de la chaîne "Code ok",
— lecture de 4 caractères,
— écriture de la chaîne "Decrypt",
— lecture d'une taille sz,
— lecture de sz octets.

La suite du code semble correspondre à l'algorithme de déchiffrement, il s'agit d'une boucle dans laquelle un caractère est lu sur l'*input*, puis envoyé vers 3 adresses (les 3 transputers suivants dans l'arbre). Les résultats sont lus avant de modifier la clé de déchiffrement et de retourner à l'adresse 0x78.

On infère de ces multiples lectures / écritures la correspondance entre les adresses utilisées lors des instructions `in` et `out`, et les numéros de canaux figurant sur le schéma du document PDF. Elle est présentée en figure 5.2.

Ces correspondances seront également valides pour les autres transputers, car ces valeurs sont chargées avec l'instruction `ldnlp` de chargement de valeur **non locale** (en dehors du *workspace* et commune à tous les processus).

Enfin, on constate que bien que l'exécution ne puisse dépasser l'instruction à 0xdb (à cause d'un saut non conditionnel), les données présentes après l'adresse 0x106 ressemblent à un autre début de programme. Il s'agit certainement du code d'un autre transputer.

FIGURE 5.2 – Correspondance entre adresses globales et canaux de communication

Adresse	Numéro de canal	R/W
0x80000000	0	W
0x80000004	1	W
0x80000008	2	W
0x8000000C	3	W
0x80000010	0	R
0x80000014	1	R
0x80000018	2	R
0x8000001C	3	R

C'est ici qu'intervient la présence de la première instruction `prod`. Son *opcode* est 0xF8, et on a remarqué qu'un deuxième code semblait commencer à l'adresse 0x106 = 0xF8 + 14. Ces 14 octets de décalage s'expliquent par le groupe de 12 octets que lit le premier transputer plus les tailles sur 1 octets des transputers 1 et 2. Aux blocs de 12 octets doivent correspondrent, d'après le code étudié, une taille suivie d'une adresse (puis 4 autres octets). En lisant ce bloc à partir de 0xF9, on trouve une taille de 0x71 octets pour l'adresse 0x80000004. Il s'agit donc bien du code du deuxième transputer, qui commence lui aussi par un octet de taille 0x70 (0x71 moins cet octet de taille). Le script Python ci-dessous effectue ce découpage en simulant une partie de la propagation de parties du fichier *input.bin*, afin d'isoler les parties à désassembler pour chacun des transputers du schéma. Des parties des transputers de la troisième rangée, non pas 1 mais 2 blocs de 12 octets sont à supprimer, puisque comme nous le verrons dans la section suivante, les transputers 1 à 3 consomment eux aussi un bloc (12 octets) pour envoyer le *chunk* de code à la bonne adresse. Les détails concernant les transputers numérotés de 4 à 12 ainsi que leurs *payloads* sont expliqués dans la sous-section dédiée au code qu'ils exécutent.

Listing 5.2– split.py : découpage du code ST20

```
import struct

fin = open('input.bin')

# transputer 0
sz = ord(fin.read(1))
with open("transputers/trans0.bin", "w") as fout:
    fout.write(fin.read(sz))
print("extracted␣transputers/trans0.bin␣(" + str(sz) + "␣bytes)")

# transputers 1 to 3
for i in range(3):
    (sz1, ad1, _) = struct.unpack("<III", fin.read(12))
    with open("transputers/trans" + str(i+1) + ".bin", "w") as fout:
        sz2 = ord(fin.read(1))
        assert(sz1 == sz2 + 1)
        fout.write(fin.read(sz2))
    s = "(" + str(sz2) + "␣bytes,␣to␣" + hex(ad1) + ")"
    print("extracted␣transputers/trans" + str(i+1) + ".bin␣" + s)

# transputers 4 to 9
for i in range(9):
    (sz1, ad1, _)   = struct.unpack("<III", fin.read(12))
    (sz2, ad2, _) = struct.unpack("<III", fin.read(12))
    with open("transputers/trans" + str(i+4) + ".bin", "w") as fout:
        assert(sz1 == sz2 + 12)
```

```
        fout.write(fin.read(sz2))
    s = "(" + str(sz2) + "␣bytes,␣via␣" + hex(ad1) + ",␣to␣" + hex(ad2) + ")"
    print("extracted␣transputers/trans" + str(i+4) + ".bin␣" + s)

for i in range(9):
    (sz1, _, _) = struct.unpack("<III", fin.read(12))
    (sz2, _, _) = struct.unpack("<III", fin.read(12))
    (sz3, _, _) = struct.unpack("<III", fin.read(12))
    with open("transputers/trans" + str(i+4) + "_payload.bin", "w") as fout:
        assert(sz1 == sz2 + 12)
        assert(sz2 == sz3 + 12)
        fout.write(fin.read(sz3))
    s = "(" + str(sz3) + "␣bytes)"
    print("extracted␣transputers/trans" + str(i+4) + "_payload.bin␣" + s)


(sz, _, _) = struct.unpack("<III", fin.read(12))
assert(sz == 0)

print(fin.read(4))
fin.read(12)
sz = ord(fin.read(1))
print(fin.read(sz))

print(fin.tell())
```

```
$ python split.py
extracted transputers/trans0.bin (248 bytes)
extracted transputers/trans1.bin (112 bytes, to 0x80000004)
extracted transputers/trans2.bin (112 bytes, to 0x80000008)
extracted transputers/trans3.bin (112 bytes, to 0x8000000c)
extracted transputers/trans4.bin (37 bytes, via 0x80000004, to 0x80000004)
extracted transputers/trans5.bin (37 bytes, via 0x80000004, to 0x80000008)
extracted transputers/trans6.bin (37 bytes, via 0x80000004, to 0x8000000c)
extracted transputers/trans7.bin (37 bytes, via 0x80000008, to 0x80000004)
extracted transputers/trans8.bin (37 bytes, via 0x80000008, to 0x80000008)
extracted transputers/trans9.bin (37 bytes, via 0x80000008, to 0x8000000c)
extracted transputers/trans10.bin (37 bytes, via 0x8000000c, to 0x80000004)
extracted transputers/trans11.bin (37 bytes, via 0x8000000c, to 0x80000008)
extracted transputers/trans12.bin (37 bytes, via 0x8000000c, to 0x8000000c)
extracted transputers/trans4_payload.bin (68 bytes)
extracted transputers/trans5_payload.bin (68 bytes)
extracted transputers/trans6_payload.bin (128 bytes)
extracted transputers/trans7_payload.bin (88 bytes)
extracted transputers/trans8_payload.bin (144 bytes)
extracted transputers/trans9_payload.bin (72 bytes)
extracted transputers/trans10_payload.bin (140 bytes)
extracted transputers/trans11_payload.bin (100 bytes)
extracted transputers/trans12_payload.bin (120 bytes)
KEY:
congratulations.tar.bz2
2477
$
```

On constate finalement que le dernier enchaînement de lectures / écritures du transputer 0 présentées précédemment sont en fait les lectures sur *input.bin* de la clé de déchiffrement et du nom de fichier à générer. La dernière instruction du script affiche le décalage dans le fichier, après la lecture de ces données : le 2477$^{\text{ème}}$ octet est bien l'*offset* que nous avions trouvé en recherchant le contenu chiffré à partir du condensat SHA-256 présent dans le document.

Nous avons à présent isolé le code exécuté par chaque transputer. Dans la section suivante, nous les étudions chacun séparément dans le but de simuler ensuite le comportement général du programme du microcontrôleur.

5.3 Analyse des transputers

Dans cette section, nous réécrivons les actions de chaque transputer en pseudo-code. Les notations utilisées sont intuitives et proches du langage C (sans types, puisque les registres du ST20 ont tous une taille de 4 octets), et nous nous autorisons l'appel aux deux fonctions `IN(size, addr, dst_ptr)` et `OUT(size, addr, src_ptr)`.

5.3.1 Transputer 0

Nous nous concentrons cette fois sur les instructions 0x77 à 0xda, c'est-à-dire la boucle de chiffrement :

Listing 5.3– trans0.txt : désassemblage du transputer 0

```
[...]

0000005b  15                  ldlp 0x5     ; var_5 ptr
0000005c  24 f2               mint
0000005e  54                  ldnlp 0x4
0000005f  4c                  ldc 0xc
00000060  f7                  in

[...]

00000075  40                  ldc 0x0
00000076  d4                  stl 0x4     ; var_4
00000077  11          lbl_77: ldlp 0x1    ; var_1 ptr
00000078  24 f2               mint
0000007a  54                  ldnlp 0x4
0000007b  41                  ldc 0x1
0000007c  f7                  in
0000007d  15                  ldlp 0x5    ; var_5 ptr
0000007e  24 f2               mint
00000080  51                  ldnlp 0x1
00000081  4c                  ldc 0xc
00000082  fb                  out
00000083  15                  ldlp 0x5    ; var_5 ptr
00000084  24 f2               mint
00000086  52                  ldnlp 0x2
00000087  4c                  ldc 0xc
00000088  fb                  out
00000089  15                  ldlp 0x5    ; var_5 ptr
0000008a  24 f2               mint
0000008c  53                  ldnlp 0x3
0000008d  4c                  ldc 0xc
0000008e  fb                  out
0000008f  10                  ldlp 0x0    ; var_0 ptr
00000090  81                  adc 0x1
00000091  24 f2               mint
00000093  55                  ldnlp 0x5
00000094  41                  ldc 0x1
00000095  f7                  in
00000096  10                  ldlp 0x0    ; var_0 ptr
00000097  82                  adc 0x2
00000098  24 f2               mint
0000009a  56                  ldnlp 0x6
0000009b  41                  ldc 0x1
0000009c  f7                  in
0000009d  10                  ldlp 0x0    ; var_0 ptr
0000009e  83                  adc 0x3
0000009f  24 f2               mint
000000a1  57                  ldnlp 0x7
000000a2  41                  ldc 0x1
000000a3  f7                  in
000000a4  10                  ldlp 0x0    ; var_0 ptr
000000a5  81                  adc 0x1
000000a6  f1                  lb
000000a7  10                  ldlp 0x0    ; var_0 ptr
```

```
000000a8  82                  adc 0x2
000000a9  f1                  lb
000000aa  23 f3               xor
000000ac  10                  ldlp 0x0    ; var_0 ptr
000000ad  83                  adc 0x3
000000ae  f1                  lb
000000af  23 f3               xor
000000b1  10                  ldlp 0x0    ; var_0 ptr
000000b2  81                  adc 0x1
000000b3  23 fb               sb
000000b5  11                  ldlp 0x1    ; var_1 ptr
000000b6  f1                  lb
000000b7  74                  ldl 0x4     ; var_4
000000b8  15                  ldlp 0x5    ; var_5 ptr
000000b9  f2                  bsub
000000ba  f1                  lb
000000bb  74                  ldl 0x4     ; var_4
000000bc  2c f1               ssub
000000be  23 f3               xor
000000c0  10                  ldlp 0x0    ; var_0 ptr
000000c1  23 fb               sb
000000c3  10                  ldlp 0x0    ; var_0 ptr
000000c4  81                  adc 0x1
000000c5  f1                  lb
000000c6  74                  ldl 0x4     ; var_4
000000c7  15                  ldlp 0x5    ; var_5 ptr
000000c8  f2                  bsub
000000c9  23 fb               sb
000000cb  74                  ldl 0x4     ; var_4
000000cc  81                  adc 0x1
000000cd  25 fa               dup
000000cf  d4                  stl 0x4     ; var_4
000000d0  cc                  eqc 0xc
000000d1  a3                  cj 0x3      ; lbl_d5
000000d2  80                  adc 0x0
000000d3  40                  ldc 0x0
000000d4  d4                  stl 0x4     ; var_4
000000d5  10          lbl_d5: ldlp 0x0    ; var_0 ptr
000000d6  24 f2               mint
000000d8  41                  ldc 0x1
000000d9  fb                  out
000000da  66 0b               j 0x-65     ; lbl_77

[...]
```

Voici sa réécriture en pseudo-code :

```
IN(12, 0x80000010, &var_5);
var_4 = 0;
[...]
while (1) {
  IN(1, 0x80000010, &var_1);

  OUT(12, 0x80000004, &var_5);
  OUT(12, 0x80000008, &var_5);
  OUT(12, 0x8000000C, &var_5);

  IN(1, 0x80000014, &x);
  IN(1, 0x80000018, &y);
  IN(1, 0x8000001C, &z);

  x = x ^ y ^ z;
  var_0 = var_1 ^ (var_4 + 2 * var_5[var_4]);
  var_5[var_4] = x;
  var_4 = (var_4 + 1) % 12;
}
```

5.3.2 Transputers 1 à 3

Les codes des trois transputers de la deuxième rangée sont identiques :

```
$ cmp transputers/trans{1,2}.bin
$ cmp transputers/trans{1,3}.bin
$
```

Listing 5.4– trans1-3.txt : désassemblage des transputers 1 à 3

```
00000000  60 b8               ajw 0x-8
00000002  24 f2               mint
00000004  24 20 50            ldnlp 0x400
00000007  23 fc               gajw
00000009  60 b8               ajw 0x-8
0000000b  15          lbl_b:  ldlp 0x5    ; var_5 ptr
0000000c  24 f2               mint
0000000e  54                  ldnlp 0x4
0000000f  4c                  ldc 0xc
00000010  f7                  in
00000011  75                  ldl 0x5     ; var_5
00000012  21 a2               cj 0x12     ; lbl_26
00000014  25 44               ldc 0x54
00000016  21 fb               ldpi        ; lbl_6c
00000018  24 f2               mint
0000001a  54                  ldnlp 0x4
0000001b  75                  ldl 0x5     ; var_5
0000001c  f7                  in
0000001d  24 4b               ldc 0x4b
0000001f  21 fb               ldpi        ; lbl_6c
00000021  76                  ldl 0x6     ; var_6
00000022  75                  ldl 0x5     ; var_5
00000023  fb                  out
00000024  61 05               j 0x-1b     ; lbl_b
00000026  11          lbl_26: ldlp 0x1    ; var_1 ptr
00000027  24 f2               mint
00000029  54                  ldnlp 0x4
0000002a  4c                  ldc 0xc
0000002b  f7                  in
0000002c  11                  ldlp 0x1    ; var_1 ptr
0000002d  24 f2               mint
0000002f  51                  ldnlp 0x1
00000030  4c                  ldc 0xc
00000031  fb                  out
00000032  11                  ldlp 0x1    ; var_1 ptr
00000033  24 f2               mint
00000035  52                  ldnlp 0x2
00000036  4c                  ldc 0xc
00000037  fb                  out
00000038  11                  ldlp 0x1    ; var_1 ptr
00000039  24 f2               mint
0000003b  53                  ldnlp 0x3
0000003c  4c                  ldc 0xc
0000003d  fb                  out
0000003e  10                  ldlp 0x0    ; var_0 ptr
0000003f  81                  adc 0x1
00000040  24 f2               mint
00000042  55                  ldnlp 0x5
00000043  41                  ldc 0x1
00000044  f7                  in
00000045  10                  ldlp 0x0    ; var_0 ptr
00000046  82                  adc 0x2
00000047  24 f2               mint
00000049  56                  ldnlp 0x6
0000004a  41                  ldc 0x1
0000004b  f7                  in
0000004c  10                  ldlp 0x0    ; var_0 ptr
0000004d  83                  adc 0x3
0000004e  24 f2               mint
00000050  57                  ldnlp 0x7
00000051  41                  ldc 0x1
00000052  f7                  in
```

```
00000053  10                ldlp 0x0    ; var_0 ptr
00000054  81                adc 0x1
00000055  f1                lb
00000056  10                ldlp 0x0    ; var_0 ptr
00000057  82                adc 0x2
00000058  f1                lb
00000059  23 f3             xor
0000005b  10                ldlp 0x0    ; var_0 ptr
0000005c  83                adc 0x3
0000005d  f1                lb
0000005e  23 f3             xor
00000060  25 fa             dup
00000062  10                ldlp 0x0    ; var_0 ptr
00000063  23 fb             sb
00000065  10                ldlp 0x0    ; var_0 ptr
00000066  24 f2             mint
00000068  41                ldc 0x1
00000069  fb                out
0000006a  64 0a             j 0x-46     ; lbl_26
0000006c  0          lbl_6c:  j 0x0       ; lbl_6d
0000006d  b8         lbl_6d:  ajw 0x8
0000006e  22 f0             ret
```

Comme évoqué lors du découpage du fichier *input.bin*, la première boucle (instructions 0x0b à 0x24) effectue la même opération de propagation de binaire exécutable que le transputer 0, vers les trois transputers connectés en dernière rangée.

Concernant la boucle de chiffrement (de 0x26 à 0x6a), la voici réécrite en pseudo-code :

```
while (1) {
  IN(12, 0x80000010, &var_1);
  OUT(12, 0x80000004, &var_1);
  OUT(12, 0x80000008, &var_1);
  OUT(12, 0x8000000C, &var_1);

  IN(1, 0x80000014, &x);
  IN(1, 0x80000018, &y);
  IN(1, 0x8000001C, &z);

  var_0 = x ^ y ^ z;
  OUT(1, 0x80000000, &var_0);
}
```

5.3.3 Transputers 4 à 12

Voici le code extrait d'*input.bin* exécuté par les transputers 4 à 12 (il est identique pour ces 9 transputers).

Listing 5.5– trans4-12.txt : désassemblage des transputers 4 à 12

```
00000000  24 60 bd             ajw 0x-403
00000003  24 f2                mint
00000005  24 20 50             ldnlp 0x400
00000008  23 fc                gajw
0000000a  60 bd                ajw 0x-3
0000000c  10                   ldlp 0x0    ; var_0 ptr
0000000d  24 f2                mint
0000000f  54                   ldnlp 0x4
00000010  4c                   ldc 0xc
00000011  f7                   in
00000012  4b                   ldc 0xb
00000013  21 fb                ldpi        ; lbl_20
00000015  24 f2                mint
00000017  54                   ldnlp 0x4
00000018  70                   ldl 0x0     ; var_0
00000019  f7                   in
0000001a  43                   ldc 0x3
0000001b  21 fb                ldpi        ; lbl_20
0000001d  72                   ldl 0x2     ; var_2
```

```
0000001e  f2                  bsub
0000001f  f6                  gcall
00000020  0           lbl_20: j 0x0       ; lbl_21
00000021  b3          lbl_21: ajw 0x3
00000022  22 f0               ret
```

Ce code réalise deux actions :

— D'abord, il reçoit un bloc de 12 octets dont les 4 premiers octets indique une taille, et reçoit ensuite des données de cette taille, qu'il stocke à l'adresse `lbl_20`.

— Ensuite, il effectue un appel de fonction (via `gcall`) à cette adresse décalée de `var_2`.

Il ne s'agit donc pas du code définitif des transputers 4 à 12 pour l'algorithme de déchiffrement, mais plutôt de leur chargeur. Dans le script 5.2, nous avons extrait 9 *payloads* : ce sont les parties effectives à étudier pour les transputers de la dernière rangée.

De plus, ces codes utilisent deux fonctions afin d'effectuer leurs envois et réception de données. Elles sont déclarées toutes deux au début de chaque fichier de code désassemblé, c'est le décalage `var_2` (inialisé par d'autres transputers à l'aide d'un jeu de décalage de pointeur de *workspace*) lors du saut qui fait démarrer chaque transputer à partir de l'adresse 0x0c du code qu'il a reçu.

Nous étudions dans la suite ces 9 *payloads* pour reconstruire l'algorithme de déchiffrement.

Transputer 4

Listing 5.6– trans4_payload.txt : désassemblage du transputer 4

```
00000000  73                  ldl 0x3     ; var_3
00000001  72                  ldl 0x2     ; var_2
00000002  74                  ldl 0x4     ; var_4
00000003  f7                  in
00000004  22 f0               ret
00000006  73                  ldl 0x3     ; var_3
00000007  72                  ldl 0x2     ; var_2
00000008  74                  ldl 0x4     ; var_4
00000009  fb                  out
0000000a  22 f0               ret

0000000c  60 bb               ajw 0x-5
0000000e  40                  ldc 0x0
0000000f  d1                  stl 0x1     ; var_1
00000010  40                  ldc 0x0
00000011  11                  ldlp 0x1    ; var_1 ptr
00000012  23 fb               sb
00000014  4c          lbl_14: ldc 0xc
00000015  d0                  stl 0x0     ; var_0
00000016  12                  ldlp 0x2    ; var_2 ptr
00000017  24 f2               mint
00000019  54                  ldnlp 0x4
0000001a  76                  ldl 0x6     ; var_6
0000001b  61 93               call 0x-1d
0000001d  40                  ldc 0x0
0000001e  d0                  stl 0x0     ; var_0
0000001f  70          lbl_1f: ldl 0x0     ; var_0
00000020  12                  ldlp 0x2    ; var_2 ptr
00000021  f2                  bsub
00000022  f1                  lb
00000023  11                  ldlp 0x1    ; var_1 ptr
00000024  f1                  lb
00000025  f2                  bsub
00000026  2f 4f               ldc 0xff
00000028  24 f6               and
0000002a  11                  ldlp 0x1    ; var_1 ptr
0000002b  23 fb               sb
0000002d  70                  ldl 0x0     ; var_0
0000002e  81                  adc 0x1
0000002f  d0                  stl 0x0     ; var_0
```

```
00000030  4c                  ldc 0xc
00000031  70                  ldl 0x0     ; var_0
00000032  f9                  gt
00000033  a2                  cj 0x2      ; lbl_36
00000034  61 09               j 0x-17     ; lbl_1f
00000036  41          lbl_36: ldc 0x1
00000037  d0                  stl 0x0     ; var_0
00000038  11                  ldlp 0x1    ; var_1 ptr
00000039  24 f2               mint
0000003b  76                  ldl 0x6     ; var_6
0000003c  63 98               call 0x-38
0000003e  20 62 03            j 0x-2d     ; lbl_14
```

Voici sa réécriture en pseudo-code :

```
var_1 = 0;

while (1) {
  IN(12, 0x80000010, &var_2);

  for (var_0 = 0 ; var_0 < 12 ; var_0++) {
    var_1 = (var_2[var_0] + var_1) & 0xFF;
  }

  OUT(1, 0x80000000, &var_1);
}
```

Transputer 5

Listing 5.7– trans5_payload.txt : désassemblage du transputer 5

```
00000000  73                  ldl 0x3     ; var_3
00000001  72                  ldl 0x2     ; var_2
00000002  74                  ldl 0x4     ; var_4
00000003  f7                  in
00000004  22 f0               ret
00000006  73                  ldl 0x3     ; var_3
00000007  72                  ldl 0x2     ; var_2
00000008  74                  ldl 0x4     ; var_4
00000009  fb                  out
0000000a  22 f0               ret

0000000c  60 bb               ajw 0x-5
0000000e  40                  ldc 0x0
0000000f  d1                  stl 0x1     ; var_1
00000010  40                  ldc 0x0
00000011  11                  ldlp 0x1    ; var_1 ptr
00000012  23 fb               sb
00000014  4c          lbl_14: ldc 0xc
00000015  d0                  stl 0x0     ; var_0
00000016  12                  ldlp 0x2    ; var_2 ptr
00000017  24 f2               mint
00000019  54                  ldnlp 0x4
0000001a  76                  ldl 0x6     ; var_6
0000001b  61 93               call 0x-1d
0000001d  40                  ldc 0x0
0000001e  d0                  stl 0x0     ; var_0
0000001f  70          lbl_1f: ldl 0x0     ; var_0
00000020  12                  ldlp 0x2    ; var_2 ptr
00000021  f2                  bsub
00000022  f1                  lb
00000023  71                  ldl 0x1     ; var_1
00000024  23 f3               xor
00000026  2f 4f               ldc 0xff
00000028  24 f6               and
0000002a  11                  ldlp 0x1    ; var_1 ptr
0000002b  23 fb               sb
0000002d  70                  ldl 0x0     ; var_0
0000002e  81                  adc 0x1
0000002f  d0                  stl 0x0     ; var_0
```

```
00000030  4c                  ldc 0xc
00000031  70                  ldl 0x0     ; var_0
00000032  f9                  gt
00000033  a2                  cj 0x2      ; lbl_36
00000034  61 09               j 0x-17     ; lbl_1f
00000036  41          lbl_36: ldc 0x1
00000037  d0                  stl 0x0     ; var_0
00000038  11                  ldlp 0x1    ; var_1 ptr
00000039  24 f2               mint
0000003b  76                  ldl 0x6     ; var_6
0000003c  63 98               call 0x-38
0000003e  20 62 03            j 0x-2d     ; lbl_14
```

Voici sa réécriture en pseudo-code :

```
var_1 = 0;

while (1) {
  IN(12, 0x80000010, &var_2);

  for (var_0 = 0 ; var_0 < 12 ; var_0++) {
    var_1 = (var_2[var_0] ^ var_1) & 0xFF;
  }

  OUT(1, 0x80000000, &var_1);
}
```

Transputer 6

Listing 5.8– trans6_payload.txt : désassemblage du transputer 6

```
00000000  73                  ldl 0x3     ; var_3
00000001  72                  ldl 0x2     ; var_2
00000002  74                  ldl 0x4     ; var_4
00000003  f7                  in
00000004  22 f0               ret
00000006  73                  ldl 0x3     ; var_3
00000007  72                  ldl 0x2     ; var_2
00000008  74                  ldl 0x4     ; var_4
00000009  fb                  out
0000000a  22 f0               ret

0000000c  60 b9               ajw 0x-7
0000000e  40                  ldc 0x0
0000000f  d2                  stl 0x2     ; var_2
00000010  40                  ldc 0x0
00000011  d1                  stl 0x1     ; var_1
00000012  40                  ldc 0x0
00000013  d3                  stl 0x3     ; var_3
00000014  4c          lbl_14: ldc 0xc
00000015  d0                  stl 0x0     ; var_0
00000016  14                  ldlp 0x4    ; var_4 ptr
00000017  24 f2               mint
00000019  54                  ldnlp 0x4
0000001a  78                  ldl 0x8     ; var_8
0000001b  61 93               call 0x-1d
0000001d  73                  ldl 0x3     ; var_3
0000001e  c0                  eqc 0x0
0000001f  21 ab               cj 0x1b     ; lbl_3c
00000021  40                  ldc 0x0
00000022  d0                  stl 0x0     ; var_0
00000023  70          lbl_23: ldl 0x0     ; var_0
00000024  14                  ldlp 0x4    ; var_4 ptr
00000025  f2                  bsub
00000026  f1                  lb
00000027  71                  ldl 0x1     ; var_1
00000028  f2                  bsub
00000029  2f 2f 2f 4f         ldc 0xffff
0000002d  24 f6               and
0000002f  d1                  stl 0x1     ; var_1
```

```
00000030  70                  ldl 0x0     ; var_0
00000031  81                  adc 0x1
00000032  d0                  stl 0x0     ; var_0
00000033  4c                  ldc 0xc
00000034  70                  ldl 0x0     ; var_0
00000035  f9                  gt
00000036  a3                  cj 0x3      ; lbl_3a
00000037  80                  adc 0x0
00000038  61 09               j 0x-17     ; lbl_23
0000003a  41          lbl_3a: ldc 0x1
0000003b  d3                  stl 0x3     ; var_3
0000003c  71          lbl_3c: ldl 0x1     ; var_1
0000003d  28 20 20 40         ldc 0x8000
00000041  24 f6               and
00000043  4f                  ldc 0xf
00000044  24 f0               shr
00000046  71                  ldl 0x1     ; var_1
00000047  24 20 20 40         ldc 0x4000
0000004b  24 f6               and
0000004d  4e                  ldc 0xe
0000004e  24 f0               shr
00000050  23 f3               xor
00000052  2f 2f 2f 4f         ldc 0xffff
00000056  24 f6               and
00000058  71                  ldl 0x1     ; var_1
00000059  41                  ldc 0x1
0000005a  24 f1               shl
0000005c  2f 2f 2f 4f         ldc 0xffff
00000060  24 f6               and
00000062  23 f3               xor
00000064  2f 2f 2f 4f         ldc 0xffff
00000068  24 f6               and
0000006a  25 fa               dup
0000006c  d1                  stl 0x1     ; var_1
0000006d  2f 4f               ldc 0xff
0000006f  24 f6               and
00000071  12                  ldlp 0x2    ; var_2 ptr
00000072  23 fb               sb
00000074  41                  ldc 0x1
00000075  d0                  stl 0x0     ; var_0
00000076  12                  ldlp 0x2    ; var_2 ptr
00000077  24 f2               mint
00000079  78                  ldl 0x8     ; var_8
0000007a  67 9a               call 0x-76
0000007c  20 66 05            j 0x-6b     ; lbl_14
```

Voici sa réécriture en pseudo-code :

```
var_1 = 0;
var_3 = 0;

while (1) {
  IN(12, 0x80000010, &var_4);

  if (var_3 == 0) {
    for (var_0 = 0 ; var_0 < 12 ; var_0++) {
      var_1 = (var_4[var_0] + var_1) & 0xFFFF;
    }
    var_3 = 1;
  }

  tmp = ((var_1 & 0x8000) >> 0xF) ^ ((var_1 & 0x4000) >> 0xE);
  tmp = ((tmp & 0xFFFF) ^ (var_1 << 1)) & 0xFFFF;
  var_1 = tmp;

  OUT(1, 0x80000000, &tmp);
}
```

Transputer 7

Listing 5.9– trans7_payload.txt : désassemblage du transputer 7

```
00000000  73                ldl 0x3     ; var_3
00000001  72                ldl 0x2     ; var_2
00000002  74                ldl 0x4     ; var_4
00000003  f7                in
00000004  22 f0             ret
00000006  73                ldl 0x3     ; var_3
00000007  72                ldl 0x2     ; var_2
00000008  74                ldl 0x4     ; var_4
00000009  fb                out
0000000a  22 f0             ret

0000000c  60 b9             ajw 0x-7
0000000e  40                ldc 0x0
0000000f  d3                stl 0x3     ; var_3
00000010  4c          lbl_10: ldc 0xc
00000011  d0                stl 0x0     ; var_0
00000012  14                ldlp 0x4    ; var_4 ptr
00000013  24 f2             mint
00000015  54                ldnlp 0x4
00000016  78                ldl 0x8     ; var_8
00000017  61 97             call 0x-19
00000019  40                ldc 0x0
0000001a  d1                stl 0x1     ; var_1
0000001b  40                ldc 0x0
0000001c  d2                stl 0x2     ; var_2
0000001d  40                ldc 0x0
0000001e  d0                stl 0x0     ; var_0
0000001f  70          lbl_1f: ldl 0x0     ; var_0
00000020  14                ldlp 0x4    ; var_4 ptr
00000021  f2                bsub
00000022  f1                lb
00000023  71                ldl 0x1     ; var_1
00000024  f2                bsub
00000025  2f 4f             ldc 0xff
00000027  24 f6             and
00000029  d1                stl 0x1     ; var_1
0000002a  14                ldlp 0x4    ; var_4 ptr
0000002b  70                ldl 0x0     ; var_0
0000002c  86                adc 0x6
0000002d  f2                bsub
0000002e  f1                lb
0000002f  72                ldl 0x2     ; var_2
00000030  f2                bsub
00000031  2f 4f             ldc 0xff
00000033  24 f6             and
00000035  d2                stl 0x2     ; var_2
00000036  70                ldl 0x0     ; var_0
00000037  81                adc 0x1
00000038  d0                stl 0x0     ; var_0
00000039  46                ldc 0x6
0000003a  70                ldl 0x0     ; var_0
0000003b  f9                gt
0000003c  a2                cj 0x2      ; lbl_3f
0000003d  61 00             j 0x-20     ; lbl_1f
0000003f  72          lbl_3f: ldl 0x2     ; var_2
00000040  71                ldl 0x1     ; var_1
00000041  23 f3             xor
00000043  2f 4f             ldc 0xff
00000045  24 f6             and
00000047  13                ldlp 0x3    ; var_3 ptr
00000048  23 fb             sb
0000004a  41                ldc 0x1
0000004b  d0                stl 0x0     ; var_0
0000004c  13                ldlp 0x3    ; var_3 ptr
0000004d  24 f2             mint
0000004f  78                ldl 0x8     ; var_8
00000050  64 94             call 0x-4c
00000052  20 64 0b          j 0x-45     ; lbl_10
```

Voici sa réécriture en pseudo-code :

```
var_3 = 0;

while (1) {
  IN(12, 0x80000010, &var_4);
  var_1 = 0;
  var_2 = 0;

  for (var_0 = 0 ; var_0 < 6 ; var_0++) {
    var_1 = (var_4[var_0]     + var_1) & 0xFF;
    var_2 = (var_4[var_0 + 6] + var_2) & 0xFF;
  }

  var_3 = (var_1 ^ var_2) & 0xFF;
  OUT(1, 0x80000000, &var_3);
}
```

Transputer 8

Listing 5.10– trans8_payload.txt : désassemblage du transputer 8

```
00000000  73                ldl 0x3     ; var_3
00000001  72                ldl 0x2     ; var_2
00000002  74                ldl 0x4     ; var_4
00000003  f7                in
00000004  22 f0             ret
00000006  73                ldl 0x3     ; var_3
00000007  72                ldl 0x2     ; var_2
00000008  74                ldl 0x4     ; var_4
00000009  fb                out
0000000a  22 f0             ret

0000000c  61 bf             ajw 0x-11
0000000e  40                ldc 0x0
0000000f  d3                stl 0x3     ; var_3
00000010  40                ldc 0x0
00000011  d4                stl 0x4     ; var_4
00000012  40                ldc 0x0
00000013  d2                stl 0x2     ; var_2
00000014  40          lbl_14: ldc 0x0
00000015  d0                stl 0x0     ; var_0
00000016  40          lbl_16: ldc 0x0
00000017  72                ldl 0x2     ; var_2
00000018  43                ldc 0x3
00000019  f8                prod
0000001a  15                ldlp 0x5    ; var_5 ptr
0000001b  fa                wsub
0000001c  70                ldl 0x0     ; var_0
0000001d  f2                bsub
0000001e  23 fb             sb
00000020  70                ldl 0x0     ; var_0
00000021  81                adc 0x1
00000022  d0                stl 0x0     ; var_0
00000023  4c                ldc 0xc
00000024  70                ldl 0x0     ; var_0
00000025  f9                gt
00000026  a2                cj 0x2      ; lbl_29
00000027  61 0d             j 0x-13     ; lbl_16
00000029  72          lbl_29: ldl 0x2     ; var_2
0000002a  81                adc 0x1
0000002b  d2                stl 0x2     ; var_2
0000002c  44                ldc 0x4
0000002d  72                ldl 0x2     ; var_2
0000002e  f9                gt
0000002f  a2                cj 0x2      ; lbl_32
00000030  61 02             j 0x-1e     ; lbl_14
00000032  4c          lbl_32: ldc 0xc
00000033  d0                stl 0x0     ; var_0
00000034  74                ldl 0x4     ; var_4
```

```
00000035  43                ldc 0x3
00000036  f8                prod
00000037  15                ldlp 0x5    ; var_5 ptr
00000038  fa                wsub
00000039  24 f2             mint
0000003b  54                ldnlp 0x4
0000003c  21 72             ldl 0x12    ; var_18
0000003e  63 90             call 0x-40
00000040  74                ldl 0x4     ; var_4
00000041  81                adc 0x1
00000042  25 fa             dup
00000044  d4                stl 0x4     ; var_4
00000045  c4                eqc 0x4
00000046  a3                cj 0x3      ; lbl_4a
00000047  80                adc 0x0
00000048  40                ldc 0x0
00000049  d4                stl 0x4     ; var_4
0000004a  40          lbl_4a: ldc 0x0
0000004b  13                ldlp 0x3    ; var_3 ptr
0000004c  23 fb             sb
0000004e  40                ldc 0x0
0000004f  d2                stl 0x2     ; var_2
00000050  40          lbl_50: ldc 0x0
00000051  d1                stl 0x1     ; var_1
00000052  40                ldc 0x0
00000053  d0                stl 0x0     ; var_0
00000054  72          lbl_54: ldl 0x2     ; var_2
00000055  43                ldc 0x3
00000056  f8                prod
00000057  15                ldlp 0x5    ; var_5 ptr
00000058  fa                wsub
00000059  70                ldl 0x0     ; var_0
0000005a  f2                bsub
0000005b  f1                lb
0000005c  71                ldl 0x1     ; var_1
0000005d  f2                bsub
0000005e  2f 4f             ldc 0xff
00000060  24 f6             and
00000062  d1                stl 0x1     ; var_1
00000063  70                ldl 0x0     ; var_0
00000064  81                adc 0x1
00000065  d0                stl 0x0     ; var_0
00000066  4c                ldc 0xc
00000067  70                ldl 0x0     ; var_0
00000068  f9                gt
00000069  a3                cj 0x3      ; lbl_6d
0000006a  80                adc 0x0
0000006b  61 07             j 0x-19     ; lbl_54
0000006d  73          lbl_6d: ldl 0x3     ; var_3
0000006e  71                ldl 0x1     ; var_1
0000006f  23 f3             xor
00000071  2f 4f             ldc 0xff
00000073  24 f6             and
00000075  13                ldlp 0x3    ; var_3 ptr
00000076  23 fb             sb
00000078  72                ldl 0x2     ; var_2
00000079  81                adc 0x1
0000007a  d2                stl 0x2     ; var_2
0000007b  44                ldc 0x4
0000007c  72                ldl 0x2     ; var_2
0000007d  f9                gt
0000007e  a3                cj 0x3      ; lbl_82
0000007f  80                adc 0x0
00000080  63 0e             j 0x-32     ; lbl_50
00000082  41          lbl_82: ldc 0x1
00000083  d0                stl 0x0     ; var_0
00000084  13                ldlp 0x3    ; var_3 ptr
00000085  24 f2             mint
00000087  21 72             ldl 0x12    ; var_18
00000089  68 9b             call 0x-85
0000008b  20 65 04          j 0x-5c     ; lbl_32
```

Voici sa réécriture en pseudo-code :

```
var_4 = 0;

for (var_2 = 0 ; var_2 < 4 ; var_2++) {
  for (var_0 = 0 ; var_0 < 12 ; var_0++) {
    var_5[var_2 * 12 + var_0] = 0;
  }
}

while (1) {
  IN(12, 0x80000010, &(var_5[var_4 * 12]));

  var_3 = 0;

  for (var_2 = 0 ; var_2 < 4 ; var_2++) {
    var_1 = 0;

    for (var_0 = 0 ; var_0 < 12 ; var_0++) {
      var_1 = (var_1 + var_5[var_2 * 12 + var_0]) & 0xFF;
    }

    var_3 = (var_1 ^ var_3) & 0xFF;
  }

  OUT(1, 0x80000000, &var_3);
}
```

Transputer 9

Listing 5.11– trans9_payload.txt : désassemblage du transputer 9

```
00000000  73                  ldl 0x3     ; var_3
00000001  72                  ldl 0x2     ; var_2
00000002  74                  ldl 0x4     ; var_4
00000003  f7                  in
00000004  22 f0               ret
00000006  73                  ldl 0x3     ; var_3
00000007  72                  ldl 0x2     ; var_2
00000008  74                  ldl 0x4     ; var_4
00000009  fb                  out
0000000a  22 f0               ret

0000000c  60 bb               ajw 0x-5
0000000e  40                  ldc 0x0
0000000f  d1                  stl 0x1     ; var_1
00000010  4c          lbl_10: ldc 0xc
00000011  d0                  stl 0x0     ; var_0
00000012  12                  ldlp 0x2    ; var_2 ptr
00000013  24 f2               mint
00000015  54                  ldnlp 0x4
00000016  76                  ldl 0x6     ; var_6
00000017  61 97               call 0x-19
00000019  40                  ldc 0x0
0000001a  11                  ldlp 0x1    ; var_1 ptr
0000001b  23 fb               sb
0000001d  40                  ldc 0x0
0000001e  d0                  stl 0x0     ; var_0
0000001f  70          lbl_1f: ldl 0x0     ; var_0
00000020  12                  ldlp 0x2    ; var_2 ptr
00000021  f2                  bsub
00000022  f1                  lb
00000023  70                  ldl 0x0     ; var_0
00000024  47                  ldc 0x7
00000025  24 f6               and
00000027  24 f1               shl
00000029  71                  ldl 0x1     ; var_1
0000002a  23 f3               xor
```

```
0000002c  2f 4f               ldc 0xff
0000002e  24 f6               and
00000030  11                  ldlp 0x1    ; var_1 ptr
00000031  23 fb               sb
00000033  70                  ldl 0x0     ; var_0
00000034  81                  adc 0x1
00000035  d0                  stl 0x0     ; var_0
00000036  4c                  ldc 0xc
00000037  70                  ldl 0x0     ; var_0
00000038  f9                  gt
00000039  a2                  cj 0x2      ; lbl_3c
0000003a  61 03               j 0x-1d     ; lbl_1f
0000003c  41          lbl_3c: ldc 0x1
0000003d  d0                  stl 0x0     ; var_0
0000003e  11                  ldlp 0x1    ; var_1 ptr
0000003f  24 f2               mint
00000041  76                  ldl 0x6     ; var_6
00000042  63 92               call 0x-3e
00000044  20 63 09            j 0x-37     ; lbl_10
```

Voici sa réécriture en pseudo-code :

```
while (1) {
  IN(12, 0x80000010, &var_2);
  var_1 = 0;
  for (var_0 = 0 ; var_0 < 12 ; var_0++) {
    var_1 = ((var_2[var_0] << (var_0 & 0x7)) ^ var_1) & 0xFF;
  }
  OUT(1, 0x80000000, &var_1);
}
```

Transputer 10

Listing 5.12– trans10_payload.txt : désassemblage du transputer 10

```
00000000  73                  ldl 0x3     ; var_3
00000001  72                  ldl 0x2     ; var_2
00000002  74                  ldl 0x4     ; var_4
00000003  f7                  in
00000004  22 f0               ret
00000006  73                  ldl 0x3     ; var_3
00000007  72                  ldl 0x2     ; var_2
00000008  74                  ldl 0x4     ; var_4
00000009  fb                  out
0000000a  22 f0               ret

0000000c  60 b0               ajw 0x-10
0000000e  40                  ldc 0x0
0000000f  d3                  stl 0x3     ; var_3
00000010  40                  ldc 0x0
00000011  d2                  stl 0x2     ; var_2
00000012  40                  ldc 0x0
00000013  d0                  stl 0x0     ; var_0
00000014  40          lbl_14: ldc 0x0
00000015  d1                  stl 0x1     ; var_1
00000016  40          lbl_16: ldc 0x0
00000017  70                  ldl 0x0     ; var_0
00000018  43                  ldc 0x3
00000019  f8                  prod
0000001a  14                  ldlp 0x4    ; var_4 ptr
0000001b  fa                  wsub
0000001c  71                  ldl 0x1     ; var_1
0000001d  f2                  bsub
0000001e  23 fb               sb
00000020  71                  ldl 0x1     ; var_1
00000021  81                  adc 0x1
00000022  d1                  stl 0x1     ; var_1
00000023  4c                  ldc 0xc
00000024  71                  ldl 0x1     ; var_1
00000025  f9                  gt
```

```
00000026  a2                  cj 0x2      ; lbl_29
00000027  61 0d               j 0x-13     ; lbl_16
00000029  70          lbl_29: ldl 0x0     ; var_0
0000002a  81                  adc 0x1
0000002b  d0                  stl 0x0     ; var_0
0000002c  44                  ldc 0x4
0000002d  70                  ldl 0x0     ; var_0
0000002e  f9                  gt
0000002f  a2                  cj 0x2      ; lbl_32
00000030  61 02               j 0x-1e     ; lbl_14
00000032  4c          lbl_32: ldc 0xc
00000033  d0                  stl 0x0     ; var_0
00000034  72                  ldl 0x2     ; var_2
00000035  43                  ldc 0x3
00000036  f8                  prod
00000037  14                  ldlp 0x4    ; var_4 ptr
00000038  fa                  wsub
00000039  24 f2               mint
0000003b  54                  ldnlp 0x4
0000003c  21 71               ldl 0x11    ; var_17
0000003e  63 90               call 0x-40
00000040  72                  ldl 0x2     ; var_2
00000041  81                  adc 0x1
00000042  25 fa               dup
00000044  d2                  stl 0x2     ; var_2
00000045  c4                  eqc 0x4
00000046  a3                  cj 0x3      ; lbl_4a
00000047  80                  adc 0x0
00000048  40                  ldc 0x0
00000049  d2                  stl 0x2     ; var_2
0000004a  40          lbl_4a: ldc 0x0
0000004b  d1                  stl 0x1     ; var_1
0000004c  40                  ldc 0x0
0000004d  d0                  stl 0x0     ; var_0
0000004e  70          lbl_4e: ldl 0x0     ; var_0
0000004f  43                  ldc 0x3
00000050  f8                  prod
00000051  14                  ldlp 0x4    ; var_4 ptr
00000052  fa                  wsub
00000053  f1                  lb
00000054  71                  ldl 0x1     ; var_1
00000055  f2                  bsub
00000056  2f 4f               ldc 0xff
00000058  24 f6               and
0000005a  d1                  stl 0x1     ; var_1
0000005b  70                  ldl 0x0     ; var_0
0000005c  81                  adc 0x1
0000005d  d0                  stl 0x0     ; var_0
0000005e  44                  ldc 0x4
0000005f  70                  ldl 0x0     ; var_0
00000060  f9                  gt
00000061  a3                  cj 0x3      ; lbl_65
00000062  80                  adc 0x0
00000063  61 09               j 0x-17     ; lbl_4e
00000065  71          lbl_65: ldl 0x1     ; var_1
00000066  43                  ldc 0x3
00000067  24 f6               and
00000069  43                  ldc 0x3
0000006a  f8                  prod
0000006b  14                  ldlp 0x4    ; var_4 ptr
0000006c  fa                  wsub
0000006d  71                  ldl 0x1     ; var_1
0000006e  44                  ldc 0x4
0000006f  24 f0               shr
00000071  4c                  ldc 0xc
00000072  21 ff               rem
00000074  2f 4f               ldc 0xff
00000076  24 f6               and
00000078  f2                  bsub
00000079  f1                  lb
0000007a  13                  ldlp 0x3    ; var_3 ptr
0000007b  23 fb               sb
0000007d  41                  ldc 0x1
```

```
0000007e  d0                  stl 0x0     ; var_0
0000007f  13                  ldlp 0x3    ; var_3 ptr
00000080  24 f2               mint
00000082  21 71               ldl 0x11    ; var_17
00000084  67 90               call 0x-80
00000086  20 65 09            j 0x-57     ; lbl_32
```

Voici sa réécriture en pseudo-code :

```
var_2 = 0;
for (var_0 = 0 ; var_0 < 4 ; var_0++) {
  for (var_1 = 0 ; var_1 < 12 ; var_1++) {
    var_4[var_0 * 12 + var_1] = 0;
  }
}

while (1) {
  IN(12, 0x80000010, &(var_4[var_2 * 12]));
  var_2 = (var_2 + 1) % 4;

  var_1 = 0;
  for (var_0 = 0 ; var_0 < 4 ; var_0++) {
    var_1 = (var_4[var_0 * 12] + var_1) & 0xFF;
  }

  var_3 = var_4[(var_1 & 0x3) * 12] + (((var_1 >> 4) % 12) & 0xFF);
  OUT(1, 0x80000000, &var_3);
}
```

Transputer 11

Listing 5.13– trans11_payload.txt : désassemblage du transputer 11

```
00000000  73                  ldl 0x3     ; var_3
00000001  72                  ldl 0x2     ; var_2
00000002  74                  ldl 0x4     ; var_4
00000003  f7                  in
00000004  22 f0               ret
00000006  73                  ldl 0x3     ; var_3
00000007  72                  ldl 0x2     ; var_2
00000008  74                  ldl 0x4     ; var_4
00000009  fb                  out
0000000a  22 f0               ret

0000000c  60 ba               ajw 0x-6
0000000e  40                  ldc 0x0
0000000f  d1                  stl 0x1     ; var_1
00000010  40                  ldc 0x0
00000011  d2                  stl 0x2     ; var_2
00000012  4c          lbl_12: ldc 0xc
00000013  d0                  stl 0x0     ; var_0
00000014  13                  ldlp 0x3    ; var_3 ptr
00000015  24 f2               mint
00000017  54                  ldnlp 0x4
00000018  77                  ldl 0x7     ; var_7
00000019  61 95               call 0x-1b
0000001b  40                  ldc 0x0
0000001c  11                  ldlp 0x1    ; var_1 ptr
0000001d  23 fb               sb
0000001f  13                  ldlp 0x3    ; var_3 ptr
00000020  f1                  lb
00000021  13                  ldlp 0x3    ; var_3 ptr
00000022  83                  adc 0x3
00000023  f1                  lb
00000024  23 f3               xor
00000026  13                  ldlp 0x3    ; var_3 ptr
00000027  87                  adc 0x7
00000028  f1                  lb
00000029  23 f3               xor
0000002b  2f 4f               ldc 0xff
```

```
0000002d  24 f6               and
0000002f  11                  ldlp 0x1    ; var_1 ptr
00000030  23 fb               sb
00000032  41                  ldc 0x1
00000033  d0                  stl 0x0     ; var_0
00000034  11                  ldlp 0x1    ; var_1 ptr
00000035  24 f2               mint
00000037  51                  ldnlp 0x1
00000038  77                  ldl 0x7     ; var_7
00000039  63 9b               call 0x-35
0000003b  41                  ldc 0x1
0000003c  d0                  stl 0x0     ; var_0
0000003d  11                  ldlp 0x1    ; var_1 ptr
0000003e  24 f2               mint
00000040  55                  ldnlp 0x5
00000041  77                  ldl 0x7     ; var_7
00000042  64 9c               call 0x-44
00000044  11                  ldlp 0x1    ; var_1 ptr
00000045  f1                  lb
00000046  4c                  ldc 0xc
00000047  21 ff               rem
00000049  2f 4f               ldc 0xff
0000004b  24 f6               and
0000004d  11                  ldlp 0x1    ; var_1 ptr
0000004e  23 fb               sb
00000050  11                  ldlp 0x1    ; var_1 ptr
00000051  f1                  lb
00000052  13                  ldlp 0x3    ; var_3 ptr
00000053  f2                  bsub
00000054  f1                  lb
00000055  12                  ldlp 0x2    ; var_2 ptr
00000056  23 fb               sb
00000058  41                  ldc 0x1
00000059  d0                  stl 0x0     ; var_0
0000005a  12                  ldlp 0x2    ; var_2 ptr
0000005b  24 f2               mint
0000005d  77                  ldl 0x7     ; var_7
0000005e  65 96               call 0x-5a
00000060  20 65 0f            j 0x-51     ; lbl_12
```

Voici sa réécriture en pseudo-code :

```
while (1) {
  IN(12, 0x80000010, &var_3);
  var_1 = (var_3[0] ^ var_3[3] ^ var_3[7]) & 0xFF;
  OUT(1, 0x80000004, &var_1);
  IN(1, 0x80000014, &var_1);
  var_1 = (var_1 % 12) & 0xFF;
  var_2 = var_3[var_1];
  OUT(1, 0x80000000, &var_2);
}
```

Transputer 12

Listing 5.14– trans12_payload.txt : désassemblage du transputer 12

```
00000000  73                  ldl 0x3     ; var_3
00000001  72                  ldl 0x2     ; var_2
00000002  74                  ldl 0x4     ; var_4
00000003  f7                  in
00000004  22 f0               ret
00000006  73                  ldl 0x3     ; var_3
00000007  72                  ldl 0x2     ; var_2
00000008  74                  ldl 0x4     ; var_4
00000009  fb                  out
0000000a  22 f0               ret

0000000c  60 ba               ajw 0x-6
0000000e  40                  ldc 0x0
0000000f  d2                  stl 0x2     ; var_2
```

```
00000010  40                  ldc 0x0
00000011  d1                  stl 0x1     ; var_1
00000012  40                  ldc 0x0
00000013  d0                  stl 0x0     ; var_0
00000014  40          lbl_14: ldc 0x0
00000015  70                  ldl 0x0     ; var_0
00000016  13                  ldlp 0x3    ; var_3 ptr
00000017  f2                  bsub
00000018  23 fb               sb
0000001a  70                  ldl 0x0     ; var_0
0000001b  81                  adc 0x1
0000001c  d0                  stl 0x0     ; var_0
0000001d  4c                  ldc 0xc
0000001e  70                  ldl 0x0     ; var_0
0000001f  f9                  gt
00000020  a2                  cj 0x2      ; lbl_23
00000021  60 01               j 0x-f      ; lbl_14
00000023  40          lbl_23: ldc 0x0
00000024  11                  ldlp 0x1    ; var_1 ptr
00000025  23 fb               sb
00000027  13                  ldlp 0x3    ; var_3 ptr
00000028  81                  adc 0x1
00000029  f1                  lb
0000002a  13                  ldlp 0x3    ; var_3 ptr
0000002b  85                  adc 0x5
0000002c  f1                  lb
0000002d  23 f3               xor
0000002f  13                  ldlp 0x3    ; var_3 ptr
00000030  89                  adc 0x9
00000031  f1                  lb
00000032  23 f3               xor
00000034  2f 4f               ldc 0xff
00000036  24 f6               and
00000038  11                  ldlp 0x1    ; var_1 ptr
00000039  23 fb               sb
0000003b  4c                  ldc 0xc
0000003c  d0                  stl 0x0     ; var_0
0000003d  13                  ldlp 0x3    ; var_3 ptr
0000003e  24 f2               mint
00000040  54                  ldnlp 0x4
00000041  77                  ldl 0x7     ; var_7
00000042  64 9c               call 0x-44
00000044  41                  ldc 0x1
00000045  d0                  stl 0x0     ; var_0
00000046  12                  ldlp 0x2    ; var_2 ptr
00000047  24 f2               mint
00000049  55                  ldnlp 0x5
0000004a  77                  ldl 0x7     ; var_7
0000004b  64 93               call 0x-4d
0000004d  41                  ldc 0x1
0000004e  d0                  stl 0x0     ; var_0
0000004f  11                  ldlp 0x1    ; var_1 ptr
00000050  24 f2               mint
00000052  51                  ldnlp 0x1
00000053  77                  ldl 0x7     ; var_7
00000054  64 90               call 0x-50
00000056  12                  ldlp 0x2    ; var_2 ptr
00000057  f1                  lb
00000058  4c                  ldc 0xc
00000059  21 ff               rem
0000005b  2f 4f               ldc 0xff
0000005d  24 f6               and
0000005f  12                  ldlp 0x2    ; var_2 ptr
00000060  23 fb               sb
00000062  12                  ldlp 0x2    ; var_2 ptr
00000063  f1                  lb
00000064  13                  ldlp 0x3    ; var_3 ptr
00000065  f2                  bsub
00000066  f1                  lb
00000067  11                  ldlp 0x1    ; var_1 ptr
00000068  23 fb               sb
0000006a  41                  ldc 0x1
0000006b  d0                  stl 0x0     ; var_0
```

```
0000006c  11                  ldlp 0x1    ; var_1 ptr
0000006d  24 f2               mint
0000006f  77                  ldl 0x7     ; var_7
00000070  66 94               call 0x-6c
00000072  20 65 0e            j 0x-52     ; lbl_23
```

Voici sa réécriture en pseudo-code :

```
for (var_0 = 0 ; var_0 < 12 ; var_0++) {
  var_3[var_0] = 0;
}

while (1) {
  var_1 = (var_3[1] ^ var_3[5] ^ var_3[9]) & 0xFF;
  IN(12, 0x80000010, &var_3);
  IN(1, 0x80000014, &var_2);
  OUT(1, 0x80000004, &var_1);
  var_2 = (var_2 % 12) & 0xFF;
  var_1 = var_3[var_2];
  OUT(1, 0x80000000, &var_1);
}
```

Les transputers 11 et 12 diffèrent des autres transputers par des envois/réceptions de données supplémentaires en milieu de boucle de calcul. En effet, ils s'échangent une variable de calcul au milieu de leur procédure ; les adresses correspondent bien à ces deux transputers dans le tableau 5.2 et le schéma 5.1.

5.4 Simulation de l'algorithme de déchiffrement

A ce stade, nous disposons de toutes les informations nécessaires pour reconstruire une simulation de l'algorithme de déchiffrement du ST20. Nous l'implémentons en C afin d'opérer un contrôle suffisant sur les tailles des données ainsi qu'afin d'obtenir de bonnes performances pour la recherche de la clé par force brute (cf. section suivante).

Les communications effectuées par le microcontrôleur sont déterministes : le transputer 0 envoie la clé de déchiffrement aux transputers 1, 2 et 3, qui à leur tour la propagent aux transputers 4 à 12, puis l'information calculée dans ces noeuds est remontée vers le premier transputer pour modifier la clé courante. Il n'est donc pas nécessaire d'avoir recours à la programmation concurrente pour simuler cet algorithme. Chaque transputer est représenté par une fonction ; les données qu'il reçoit sont accessibles dans une variable globale, et il envoie le résultat de son calcul par valeur de retour de la fonction. Enfin, dans cette variable globale, on maintient également les états des divers transputers utilisés par des calculs d'un tour de boucle à un autre.

Listing 5.15– st20-sim.c : simulation de l'algorithme de déchiffrement

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

struct {
  unsigned char key[12];
  int t0var_4;
  unsigned char t4var_1;
  unsigned char t5var_1;
  unsigned short t6var_1;
  unsigned char t6var_3;
  unsigned char t8var_4;
  unsigned char t8var_5[48];
  unsigned char t10var_2;
  unsigned char t10var_4[48];
  unsigned char t11_12var_3[12];
} st;

unsigned char trans0(char c);
unsigned char trans1();
```

```
unsigned char trans2();
unsigned char trans3();
unsigned char trans4();
unsigned char trans5();
unsigned char trans6();
unsigned char trans7();
unsigned char trans8();
unsigned char trans9();
unsigned char trans10();
unsigned short trans11_12();


unsigned char trans0(unsigned char)
{
  unsigned char x = trans1();
  unsigned char y = trans2();
  unsigned char z = trans3();

  x = x ^ y ^ z;
  unsigned char var_0 = var_1 ^ (st.t0var_4 + 2 * st.key[st.t0var_4]);
  st.key[st.t0var_4] = x;
  st.t0var_4 = (st.t0var_4 + 1) % 12;

  return var_0;
}

unsigned char trans1()
{
  return trans4() ^ trans5() ^ trans6();
}

unsigned char trans2()
{
  return trans7() ^ trans8() ^ trans9();
}

unsigned char trans3()
{
  unsigned short xy = trans11_12();
  unsigned char x = xy >> 8;
  unsigned char y = xy & 0xff;
  return trans10() ^ x ^ y;
}

unsigned char trans4()
{
  for (int var_0 = 0 ; var_0 < 12 ; var_0++)
    st.t4var_1 = st.key[var_0] + st.t4var_1;
  return st.t4var_1;
}

unsigned char trans5()
{
  for (int var_0 = 0 ; var_0 < 12 ; var_0++)
    st.t5var_1 = st.key[var_0] ^ st.t5var_1;
  return st.t5var_1;
}

unsigned char trans6()
{
  if (st.t6var_3 == 0) {
    for (int var_0 = 0 ; var_0 < 12 ; var_0++) {
      st.t6var_1 = (st.key[var_0] + st.t6var_1) & 0xFFFF;
    }
    st.t6var_3 = 1;
  }

  unsigned short tmp;
  tmp = ((st.t6var_1 & 0x8000) >> 0xF) ^
        ((st.t6var_1 & 0x4000) >> 0xE);
  tmp = ((tmp & 0xFFFF) ^ (st.t6var_1 << 1)) & 0xFFFF;
  st.t6var_1 = tmp;
```

```
  return (tmp & 0xFF);
}

unsigned char trans7()
{
  unsigned char var_1 = 0;
  unsigned char var_2 = 0;

  for (int var_0 = 0 ; var_0 < 6 ; var_0++) {
    var_1 = (st.key[var_0]     + var_1) & 0xFF;
    var_2 = (st.key[var_0 + 6] + var_2) & 0xFF;
  }

  unsigned char var_3 = (var_1 ^ var_2) & 0xFF;
  return var_3;
}

unsigned char trans8()
{
  for (int i = 0 ; i < 12 ; i++) {
    st.t8var_5[st.t8var_4 * 12 + i] = st.key[i];
  }
  st.t8var_4 = (st.t8var_4 + 1) % 4;

  unsigned char var_3 = 0;

  for (int var_2 = 0 ; var_2 < 4 ; var_2++) {
    unsigned char var_1 = 0;

    for (int var_0 = 0 ; var_0 < 12 ; var_0++) {
      var_1 = (var_1 + st.t8var_5[var_2 * 12 + var_0]) & 0xFF;
    }

    var_3 = (var_1 ^ var_3) & 0xFF;
  }

  return var_3;
}

unsigned char trans9()
{
  unsigned char var_1 = 0;
  for (int var_0 = 0 ; var_0 < 12 ; var_0++) {
    var_1 = ((st.key[var_0] << (var_0 & 0x7)) ^ var_1) & 0xFF;
  }
  return var_1;
}

unsigned char trans10()
{
  for (int i = 0 ; i < 12 ; i++) {
    st.t10var_4[st.t10var_2 * 12 + i] = st.key[i];
  }
  st.t10var_2 = (st.t10var_2 + 1) % 4;

  unsigned char var_1 = 0;
  for (int var_0 = 0 ; var_0 < 4 ; var_0++) {
    var_1 = (st.t10var_4[var_0 * 12] + var_1) & 0xFF;
  }

  unsigned char var_3;
  var_3 = st.t10var_4[(var_1 & 0x3) * 12 + (((var_1 >> 4) % 12) & 0xFF)];
  return var_3;
}

unsigned short trans11_12()
{
  unsigned char t11var_1;
  t11var_1 = st.key[0] ^ st.key[3] ^ st.key[7];

  unsigned char t12var_1;
  t12var_1 = (st.t11_12var_3[1] ^ st.t11_12var_3[5] ^ st.t11_12var_3[9]);
```

```
  for (int i = 0 ; i < 12 ; i++) {
    st.t11_12var_3[i] = st.key[i];
  }

  unsigned char t12var_2;
  unsigned t11var_2;
  t12var_2 = st.t11_12var_3[t11var_1 % 12];
  t11var_2 = st.key[t12var_1 % 12];
  return (t11var_2 << 8) | t12var_2;
}

int main(int argc, char **argv)
{
  st.key[0]  = '*';
  st.key[1]  = 'S';
  st.key[2]  = 'S';
  st.key[3]  = 'T';
  st.key[4]  = 'I';
  st.key[5]  = 'C';
  st.key[6]  = '-';
  st.key[7]  = '2';
  st.key[8]  = '0';
  st.key[9]  = '1';
  st.key[10] = '5';
  st.key[11] = '*';

  for (int i = 0 ; i < 48 ; i++) {
    st.t8var_5[i] = 0;
    st.t10var_4[i] = 0;
  }
  for (int i = 0 ; i < 12 ; i++) {
    st.t11_12var_3[i] = 0;
  }

  unsigned char test_vector[25] = "\x1d\x87\xc4\xc4\xe0\xee\x40\x38\x3c\x59\x44\x7f\x23\x79\x8d\x9f
\xef\xe7\x4f\xb8\x24\x80\x76\x6e";

  for (int i = 0 ; i < 24 ; i++) {
    test_vector[i] = trans0(test_vector[i]);
  }

  printf("%s\n", test_vector);
  return 0;
}
```

```
$ gcc st20-sim.c
$ ./a.out
I love ST20 architecture
$
```

5.5 Recherche de la clé de déchiffrement

Notre prochain objectif est de retrouver la clé qui nous permettra de déchiffrer le fichier *congratulations.tar.bz2*. Cette fois, nous n'avons pas affaire à une chiffrement par bloc (comme l'était AES dans les autres stages), mais à un chiffrement sensible à une attaque de type "clair connu". La clé étant modifiée tout au long du déroulement de l'algorithme, nous ne disposons que du premier bloc de 12 octets pour découvrir la clé.

Les 12 premiers octets d'un fichier de type BZ2 [1] sont les suivants :

— 2 octets pour le nombre magique "BZ",
— 1 octet pour la version ('h' pour bzip2),
— 1 octet pour la taille des blocs à compresser (de '1' à '9'),
— 6 octets pour le second nombre magique "\x31\x41\x59\x26\x53\x59",

1. http://en.wikipedia.org/wiki/Bzip2#File_format

— 2 octets pour une partie du CRC32 du premier bloc.

Sur ces 12 octets, nous en connaissons donc déjà 9 ; une attaque par force brute nous permettra de retrouver la taille des blocs ainsi que la partie du CRC qui nous aiderons à compléter notre connaissance de la clé.

On recherche donc d'abord un à un les caractères de la clé qui déchiffrent le premier bloc du fichier *encrypted* extrait de *input.bin* vers un en-tête BZ2 correct.

Listing 5.16– atk-st5key.c : découverte de 9 des 12 octets de la clé

```
[...]

void reset(int n, char k)
{
  for (int i = 0 ; i < 12 ; i++) {
    st.key[i] = 0;
  }
  st.key[n] = k;

  st.t0var_4 = 0;
  st.t4var_1 = 0;
  st.t5var_1 = 0;
  st.t6var_1 = 0;
  st.t6var_3 = 0;
  st.t8var_4 = 0;
  st.t10var_2 = 0;

  for (int i = 0 ; i < 48 ; i++) {
    st.t8var_5[i] = 0;
    st.t10var_4[i] = 0;
  }
  for (int i = 0 ; i < 12 ; i++) {
    st.t11_12var_3[i] = 0;
  }
}

int main(int argc, char **argv)
{
  unsigned char plaintext[13] = "BZh.\x31\x41\x59\x26\x53\x59..";
  unsigned char encrypted[13] = "\xfe\xf3\x50\xdc\x81\xbc\x97\x27\x89\xac\x72\x28";
  unsigned char decrypted[13];

  for (int n = 0 ; n < 12 ; n++) {
    if (n == 3 || n == 10 || n == 11) {
      printf("0x..␣");
      continue;
    }

    for (int k = 0 ; k < 128 ; k++) {
      reset(n, k);
      for (int i = 0 ; i < 12 ; i++) {
        decrypted[i] = trans0(encrypted[i]);
      }
      if (decrypted[n] == plaintext[n])
        printf("0x%02x␣", k);
    }
  }

  printf("\n");
  return 0;
}
```

```
$ gcc atk-st5key.c
$ ./a.out
0x5e 0x54 0x1b 0x.. 0x56 0x7c 0x64 0x7d 0x69 0x76 0x.. 0x..
$
```

Il ne reste maintenant plus que 3 caractères de clé inconnus. Nous les trouverons par force brute. Afin d'accélérer cette recherche, nous utilisons une heuristique différente de la comparaison du condensat SHA-256 pour vérifier la validité du fichier déchiffré. Les octets 16 à 48 d'un fichier

BZ2 contiennent une table des symboles utilisés par l'algorithme de compression/décompression de Huffman. Cette table est très souvent assez dense. Nous recherchons un fichier déchiffré dans lequel au moins 50% des symboles de cette table sont marqués à 0xFF. De cette façon, il n'est pas nécessaire de déchiffrer immédiatement l'intégralité du fichier, mais uniquement ses 48 premiers octets.

Listing 5.17– bf-st5key.c : recherche de la clé

```
[...]

void reset(unsigned char k1, unsigned char k2, unsigned char k3)
{
  st.key[0]  = 0x5e;
  st.key[1]  = 0x54;
  st.key[2]  = 0x1b;
  st.key[3]  = k1;
  st.key[4]  = 0x56;
  st.key[5]  = 0x7c;
  st.key[6]  = 0x64;
  st.key[7]  = 0x7d;
  st.key[8]  = 0x69;
  st.key[9]  = 0x76;
  st.key[10] = k2;
  st.key[11] = k3;

  st.t0var_4 = 0;
  st.t4var_1 = 0;
  st.t5var_1 = 0;
  st.t6var_1 = 0;
  st.t6var_3 = 0;
  st.t8var_4 = 0;
  st.t10var_2 = 0;

  for (int i = 0 ; i < 48 ; i++) {
    st.t8var_5[i] = 0;
    st.t10var_4[i] = 0;
  }
  for (int i = 0 ; i < 12 ; i++) {
    st.t11_12var_3[i] = 0;
  }
}

int main(int argc, char **argv)
{
  unsigned char encrypted[250606];
  FILE *fin = fopen("encrypted", "rb");
  fread(encrypted, 1, 250606, fin);
  fclose(fin);

  unsigned char decrypted[250606];

  for (int k1 = 0 ; k1 < 128 ; k1++) {
    for (int k2 = 0 ; k2 < 256 ; k2++) {
      for (int k3 = 0 ; k3 < 256 ; k3++) {
        reset((unsigned char)k1,
              (unsigned char)k2,
              (unsigned char)k3);
        for (int i = 0 ; i < 48 ; i++) {
          decrypted[i] = trans0(encrypted[i]);
        }

        int nb = 0;
        for (int i = 0 ; i < 48 ; i++) {
          if (decrypted[i] == 0xFF) nb++;
        }
        if (nb >= 16) {
          printf("%d␣%d␣%d\n", k1, k2, k3);
          for (int i = 0 ; i < 250606 ; i++)
            decrypted[i] = trans0(encrypted[i]);
          FILE *fout = fopen("congratulations.tar.bz2", "wb");
          fwrite(decrypted, 1, 250606, fout);
```

```
          fclose(fout);
        }
      }
    }
  }

  return 0;
}
```

```
$ gcc bf-st5key.c -O3
$ ./a.out
161 218 197
$ shasum -a 256 congratulations.tar.bz2
9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321  congratulations.tar.bz2
```

Stage 6 : stéganographie et fichiers images

```
$ bunzip2 congratulations.tar.bz2
$ tar xvf congratulations.tar
x congratulations.jpg
$
```

Cette nouvelle archive ne contient qu'un fichier dont l'image est présentée dans la figure ci-dessous :

FIGURE 6.1 – L'image *congratulations.jpg*

La suite du challenge est donc cachée dans ce fichier.

Il existe une multitude de méthodes pour dissimuler des données dans un fichier [1]. Certaines techniques consistent à insérer directement un fichier dans un autre, en ajustant éventuellement les offsets des structures du fichier hôte afin de conserver son intégrité. D'autres techniques plus avancées utilisent la structure d'un fichier et ses spécificités pour y éparpiller des données par morceaux, là où elles seront le moins visibles. C'est un petit panorama de ces techniques sur support de fichiers images que propose cette dernière étape du challenge SSTIC 2015.

6.1 Données cachées dans un fichier *JPG*

Hachoir [2] est une librarie Python permettant de découper un flux de données morceau par morceau, selon le type de flux considéré. Plus spécifiquement, l'outil hachoir-subfile inspecte un fichier à la recherche de nombres magiques afin d'extraire d'autres fichiers de celui-ci.

```
$ hachoir-subfile congratulations.jpg
[+] Start search on 252569 bytes (246.6 KB)

[+] File at 0 size=55248 (54.0 KB): JPEG picture
[+] File at 55248: bzip2 archive

[+] End of search -- offset=252569 (246.6 KB)
```

Hachoir-subfile détecte ici l'empreinte du début d'un fichier BZ2 à la suite du marqueur de fin de fichier JPG "\xFF\xD9". Nous extrayons cette archive et découvrons une nouvelle image à l'intérieur.

```
$ dd if=congratulations.jpg of=congratz2.bz2 bs=1 skip=55248
197321+0 records in
197321+0 records out
197321 bytes transferred in 0.488730 secs (403742 bytes/sec)
$ bunzip congratz2.bz2
$ file congratz2
congratz2: POSIX tar archive (GNU)
$ tar xvf congratz2
x congratulations.png
$
```

Ce n'était visiblement pas le dernier effort. Recherchons des données dans cette nouvelle image.

6.2 Données cachées dans un fichier *PNG*

```
$ hachoir-subfile congratulations.png
[+] Start search on 197557 bytes (192.9 KB)

[+] File at 0 size=197557 (192.9 KB): PNG picture: 636x474x32 (alpha layer)

[+] End of search -- offset=197557 (192.9 KB)
$
```

Cette fois, il faut inspecter d'un peu plus près la structure du fichier. Le site de la libPNG propose une documentation détaillée sur ce format PNG. Les fichiers sont découpés en *chunks*, dont les premiers caractères en définissent le type :

— Chaque fichier doit commencer par un *chunk* de type IHDR,
— et terminer par un *chunk* de type IEND.
— Entre les deux figure au minimum un *chunk* de type IDAT contenant les données de l'image.

1. http://fr.wikipedia.org/wiki/St%C3%A9ganographie
2. https://bitbucket.org/haypo/hachoir/wiki/Home

FIGURE 6.2 – L'image *congratulations.png*

— Plusieurs autres types de *chunks* sont optionnels, par exemple : bKGD, pHYs, sBIT, sPLT, hIST, ...

La librarie png du language Python nous permet d'inspecter simplement le fichier *congratulations.png* en listant les *chunks* qui le composent :

```
$ python
[...]
>>> import png
>>> r = png.Reader("congratulations.png")
>>> [chunk[0] for chunk in r.chunks()]
['IHDR', 'bKGD', 'pHYs', 'tIME', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 's
Tic', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic
', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 'sTic', 'IDAT', 'IDAT', 'IDAT', ' IDAT', 'IDAT',
'IDAT', 'IDAT', 'IDAT', 'IEND']
>>> ^D
$
```

Parmi les *chunks* composant l'image PNG se trouvent plusieurs *chunks* de type 'sTic'. Ce sont ces morceaux de fichiers qu'il nous faut extraire afin de reconstruire l'information suivante.

```
$ python
[...]
>>> import png
>>> r = png.Reader("congratulations.png")
>>> f = open("congratz3", "w")
>>> for chunk in r.chunks():
...     if chunk[0] == "sTic":
...         f.write(chunk[1])
...
>>> f.close()
>>> ^D
$ file congratz3
congratz3: zlib compressed data
$ openssl zlib -d < congratz3 > congratz3.bz2
$ file congratz3.bz2
```

```
congratz3.bz2: bzip2 compressed data, block size = 900k
$ rm congratz3 ; bunzip2 congratz3.bz2
$ file congratz3
congratz3: POSIX tar archive (GNU)
$ tar xvf congratz3
x congratulations.tiff
```

Une nouvelle image, au format TIFF cette fois.

FIGURE 6.3 – L'image *congratulations.tiff*

6.3 Données cachées dans un fichier *TIFF*

Le format d'image TIFF [3] est un format de conteneur, enrobant les informations directes de couleurs de pixels d'une image. La librarie Pillow [4] nous permet d'inspecter les quelques premiers pixels de l'image :

```
$ python
[...]
>>> from PIL import *
>>> im = Image.open("congratulations.tiff")
>>> [im.getpixel((x,0)) for x in range(10)]
[(0, 1, 0), (0, 0, 0), (0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 1, 0), (1, 0, 0), (1, 0, 0), (0, 1, 0),
(1, 0, 0)]
>>>
```

Dans les premières lignes de l'image, tous les pixels semblent être noirs (composantes R, G et B à 0). Or on constate que quelques composantes semblent être égale à 1 dans la première ligne au moins. C'est un des signes de l'utilisation d'une technique similaire à LSB, dans laquelle les bits de poids faibles des composantes de couleurs sont utilisées pour stocker des données. Cette

3. http://fr.wikipedia.org/wiki/Tagged_Image_File_Format
4. http://python-pillow.github.io/

altération des couleurs n'a alors qu'un impact très faible sur le rendu général de l'image. Ici, seuls les deux premières composantes (R et G) semblent être exploitées, la troisième (B) reste à 0 sur les pixels noirs. Le script suivant extrait ces bits de poids faible et construit un nouveau fichier.

Listing 6.1– lsb.py : extraction des données dissimulées dans *congratulations.tiff*

```
from PIL import Image

im = Image.open("congratulations.tiff")
s = ""
(w,h) = im.size

for y in range(h):
    for x in range(w):
        (r,g,_) = im.getpixel((x,y))
        s += str(r%2) + str(g%2)

with open("congratz4.bz2", "wb") as f:
    for i in range(0, len(s), 8):
        f.write(chr(int(s[i:i+8], 2)))
```

```
$ python lsb.py
$ bunzip2 congratz4.bz2

bunzip2: congratz4.bz2: trailing garbage after EOF ignored
$ tar xvf congratz4
x congratulations.gif
$
```

Et voilà la dernière image, au format GIF.

FIGURE 6.4 – L'image *congratulations.gif*

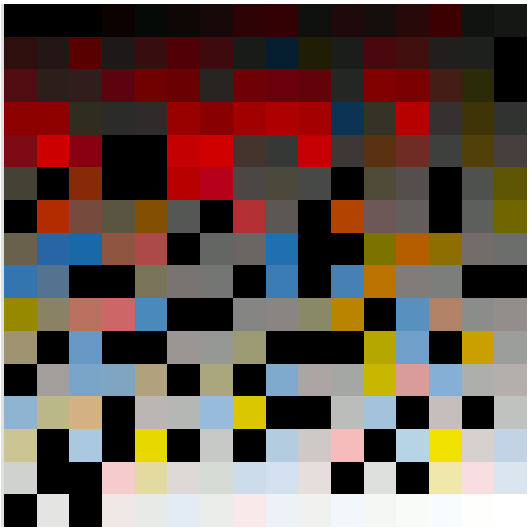
6.4 Données cachées dans un fichier *GIF*

De même que pour l'image précédente, on inspecte quelques pixels :

```
$ python
[...]
>>> from PIL import Image
>>> im = Image.open("congratulations.gif")
>>> [im.getpixel((x,0)) for x in range(10)]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
>>> len(im.getcolors())
254
>>>
```

Ici, pas de composantes RGB, mais un seul entier pour représenter une couleur. L'image embarque une palette de couleurs, dont les couleurs sont référencées par les valeurs des pixels. Un appel à `im.getcolors()` nous renvoie une liste de couples formés des couleurs (indexées donc) et du nombre de pixels utilisant cette couleur. 254 couleurs aux indexes différents sont utilisé par cette image. Toutefois, on ne semble pas distinguer autant de couleurs différentes dans l'image... La figure ci-dessous présente la palette de couleur de l'image, obtenue en l'ouvrant avec le logiciel Gimp.

FIGURE 6.5 – Palette de couleurs de *congratulations.gif*



On constate que plusieurs couleurs de palette affichent la couleur noire. Des pixels noirs peuvent se différencier par leur index, mais être tous visualisés en noir. Toujours sous Gimp, nous modifions tour à tour plusieurs couleurs de la palette, et obtenons finalement l'image suivante :

Nous avons finalement trouvé l'adresse e-mail du challenge SSTIC :

`1713e7c1d0b750ccd4e002bb957aa799@challenge.sstic.org`

FIGURE 6.6 – L'image *congratulations.gif* modifiée

Conclusion

Ce challenge a été l'occasion de découvrir ou redécouvrir plusieurs formats de fichiers hétéroclites, ainsi que diverses techniques pour y dissimuler de l'information. Le stage 5, pour lequel il a fallu passer autant par le désassemblage et par la rétro-ingénierie d'un programme ST20 que par la recherche d'une clé de chiffrement par force brute a été particulièrement instructif et stimulant. Merci donc aux concepteurs du challenge pour ce grand cru 2015 !

ANNEXE A

Outils et librairies utilisées

- python2.7, python3.2
- librairies python :
  - turtle
  - pillow
  - libpng
  - pycrypto
- gcc
- librairies c :
  - libkripto
- g++
- librairies c++ :
  - crypto++
- ducky decode
- hachoir
- wireshark
- firefox
- vim
- latex

ANNEXE B

Code du désassembleur ST20 développé pour le stage 5

Le désassembleur présenté ci-dessous n'est pas exhaustif, mais supporte tous les *opcodes* rencontrés dans le stage 5.

Listing B.1– st20-disas.py : désassembleur de ST20

```
import struct
import sys

if len(sys.argv) < 2:
    print("USAGE: python st20-disas.py <FILE> (<SIZE>)")
    sys.exit(1)

fname = sys.argv[1]

with open(fname, 'r') as f:
    data = f.read()

if len(sys.argv) == 3:
    fsize = int(sys.argv[2])
else:
    fsize = len(data)

seconds= { 0x00: "rev",
           0x01: "lb",
           0x02: "bsub",
           0x03: "endp",
           0x04: "diff",
           0x05: "add",
           0x06: "gcall",
           0x07: "in",
           0x08: "prod",
           0x09: "gt",
           0x0a: "wsub",
           0x0b: "out",
           0x0c: "sub",
           0x0d: "startp",
           0x0e: "outbyte",
           0x0f: "outword",
           0x1b: "ldpi",
           0x1f: "rem",
           0x20: "ret",
           0x33: "xor",
```

```
          0x3b: "sb",
          0x3c: "gajw",
          0x40: "shr",
          0x41: "shl",
          0x42: "mint",
          0x46: "and",
          0x5a: "dup",
          0xc1: "ssub"
        }

class Instr:
    def __init__(self, addr, hex_repr, op, arg = None):
        self.__addr     = addr
        self.__hex_repr = hex_repr
        self.__op       = op
        self.__arg      = arg
        self.__com      = None
        self.__label    = False

    def get_addr(self):
        return self.__addr

    def get_op(self):
        return self.__op

    def get_arg(self):
        return self.__arg

    def set_com(self, com):
        self.__com = com

    def labelize(self):
        self.__label = True

    def __str__(self):
        tmp = hex(self.__hex_repr)[2:]
        hex_repr = "␣".join([ tmp[i:i+2] for i in range(0, len(tmp), 2) ])
        str = "%08x␣␣%s␣" % (self.__addr, hex_repr.ljust(10))
        if self.__label:
            str += ("lbl_" + hex(self.__addr)[2:] + ":").rjust(8)
        else:
            str += "␣" * 8
        str += "␣␣" + self.__op
        if self.__arg != None:
            str += "␣0x%x" % (self.__arg,)
        if self.__com:
            str += "\t;␣" + self.__com
        return str

listing   = []
lbl_table = []
addr = -1
size = 0
hexa = 0
reg  = 0

# step 1: decode opcodes
while addr < fsize - 1:
    addr += 1
    size += 1
    d = ord(data[addr])

    if hexa != 0: hexa = hexa << 8
    hexa += d

    opc = d & 0xF0
    dat = d & 0x0F
    reg = reg | dat

    start = addr - size + 1

    if opc == 0x00:
        instr = Instr(start, hexa, "j", reg)
```

```
        instr.set_com("lbl_" + hex(addr + reg + 1)[2:])
        lbl_table.append((instr, addr + reg + 1))
    if opc == 0x10:
        instr = Instr(start, hexa, "ldlp", reg)
        instr.set_com("var_" + str(reg) + "␣ptr")
    elif opc == 0x20:
        reg = reg << 4
    elif opc == 0x30:
        instr = Instr(start, hexa, "ldln", reg)
    elif opc == 0x40:
        instr = Instr(start, hexa, "ldc", reg)
    elif opc == 0x50:
        instr = Instr(start, hexa, "ldnlp", reg)
    elif opc == 0x60:
        reg = (~reg) << 4
    elif opc == 0x70:
        instr = Instr(start, hexa, "ldl", reg)
        instr.set_com("var_" + str(reg))
    elif opc == 0x80:
        instr = Instr(start, hexa, "adc", reg)
    elif opc == 0x90:
        instr = Instr(start, hexa, "call", reg)
    elif opc == 0xa0:
        instr = Instr(start, hexa, "cj", reg)
        instr.set_com("lbl_" + hex(addr + reg + 1)[2:])
        lbl_table.append((instr, addr + reg + 1))
    elif opc == 0xb0:
        instr = Instr(start, hexa, "ajw", reg)
    elif opc == 0xc0:
        instr = Instr(start, hexa, "eqc", reg)
    elif opc == 0xd0:
        instr = Instr(start, hexa, "stl", reg)
        instr.set_com("var_" + str(reg))
    elif opc == 0xe0:
        instr = Instr(start, hexa, "stnl", reg)
    elif opc == 0xf0:
        if reg == 0x1b:
            assert(listing[-1].get_op() == "ldc")
            dst = addr + listing[-1].get_arg() + 1
            instr = Instr(start, hexa, "ldpi")
            instr.set_com("lbl_" + hex(dst)[2:])
            lbl_table.append((instr, dst))
        else:
            try:
                instr = Instr(start, hexa, seconds[reg])
            except:
                instr = Instr(start, hexa, "unknown", reg)

    if opc != 0x60 and opc != 0x20:
        listing.append(instr)
        reg  = 0
        size = 0
        hexa = 0

# step 2: annotate label locations
def find_instr_at(dst, listing):
    for instr in listing:
        if instr.get_addr() == dst:
            return instr
        if instr.get_addr() > dst:
            return None
    return None

for (_, dst) in lbl_table:
    dst_instr = find_instr_at(dst, listing)
    if dst_instr:
        dst_instr.labelize()

# output listing
for instr in listing:
    print(instr)
```