Solution du challenge SSTIC 2015

Frédéric Perriot
fperriot@gmail.com

0 Introduction

Le challenge du SSTIC 2015 consiste à retrouver une adresse courriel du domaine @challenge.sstic.org cachée dans un fichier "challenge.zip".

Pour extraire l'adresse en question, on doit passer par six étapes.

La première étape consiste à extraire une série de scripts PowerShell depuis un programme écrit en DuckyScript.

La seconde nécessite d'assembler une clé de déchiffrement en utilisant les indices disséminés dans un jeu de tir à la première personne.

La troisième étape consiste à reconstituer un dessin d'après une capture USB de souris.

La quatrième nécessite de comprendre un programme Javascript obfusqué, et de retrouver une clé de déchiffrement en faisant une attaque par dictionnaire hybride.

La cinquième étape met en jeu l'étude d'un système parallèle composé de transputeurs ST20. On doit reconstruire un algorithme de chiffrement, et exploiter ses faiblesses pour retrouver une clé par le biais d'une attaque à clair connu.

La sixième et dernière étape nécessite d'extraire une série d'images emboîtées les unes dans les autres par des procédés variés. L'image finale contient l'adresse courriel recherchée.

Ce document expose une voie possible pour la résolution du challenge, qui n'est probablement ni la plus courte, ni la mieux fréquentée.

Pour résoudre ce challenge, j'ai travaillé sur un système d'exploitation Linux Debian, et utilisé les langages de programmation Perl, Python, OCaml, C, et bash.

Aucun canard n'a été blessé durant la rédaction de cette solution.

1 Ducky Script et Powershell

Le fichier initial se présente sous la forme d'une image disque nommée "sdcard.img". L'examen rapide d'un dump hexadécimal indique qu'il s'agit d'un disque au format FAT16 dont les fichiers ont été effacés (indiqué par l'octet 0xe5 en début d'entrée):

```
$ xxd sdcard.img | less
0000000: eb3c 906d 6b66 732e 6661 7400 0204 0100  .<.mkfs.fat.....
0000010: 0200 0200 00f8 f400 2000 4000 0000 0000  ........ .@.....
0000020: 90d0 0300 8000 293b 880d e54e 4f20 4e41  ......);...NO NA
0000030: 4d45 2020 2020 4641 5431 3620 2020 0e1f  ME    FAT16   ..
...
003d200: e562 0075 0069 006c 0064 000f 00bd 2e00  .b.u.i.l.d......
003d210: 7300 6800 0000 ffff ffff 0000 ffff ffff  s.h.............
003d220: e555 494c 4420 2020 5348 2020 0000 2d1e  .UILD   SH  ..-.
003d230: 7a46 7a46 0000 2d1e 7a46 0300 2d00 0000  zFzF..-.zF..-...
003d240: 4169 006e 006a 0065 0063 000f 00e4 7400  Ai.n.j.e.c....t.
003d250: 2e00 6200 6900 6e00 0000 0000 ffff ffff  ..b.i.n.........
003d260: 494e 4a45 4354 2020 4249 4e20 0000 3a1e  INJECT  BIN ..:.
003d270: 7a46 7a46 0000 3a1e 7a46 0400 a2ab 0a02  zFzF..:.zF......
```

L'outil forensique "The Sleuth Kit" permet de récupérer les fichiers effacés:

```
$ fls sdcard.img
r/r * 4:        build.sh
r/r 6:  inject.bin
v/v 3992179:    $MBR
v/v 3992180:    $FAT1
v/v 3992181:    $FAT2
d/d 3992182:    $OrphanFiles
$ fcat sdcard.img build.sh > build.sh
$ fcat inject.bin sdcard.img > inject.bin
```

Voyons le contenu de "build.sh":

```
$ cat build.sh
java -jar encoder.jar -i /tmp/duckyscript.txt
```

Une recherche de "duckyscript" sur Internet nous indique que ces fichiers sont liés à l'outil d'audit "USB Rubber Ducky". [1] Il s'agit d'une clé USB qui, sous couvert d'être un média de stockage passif, injecte en réalité des frappes clavier dans l'hôte où elle est connectée.

[1] http://www.usbrubberducky.com/

Figure 1: clé USB Rubber Ducky

Les frappes à injecter sont programmées dans un langage de script nommé "Ducky Script", compilable dans un format binaire par l'outil "encoder.jar". Le fichier binaire de sortie est généralement nommé "inject.bin". Il est placé sur une mémoire flash au format *micro SD card* et fournit la charge utile au micro-contrôleur de la clé. Voilà donc la nature du second fichier récupéré, qu'il s'agit alors de décompiler vers le script source.

Par chance un décodeur existe déjà. Il s'agit d'un script Perl "ducky-decode.pl" écrit par "Midnitesnake". [2] Après quelques modifications mineures pour lui permettre de lire le gros (34Mo) fichier "inject.bin", on récupère le script d'origine:

```
$ perl ducky-decode.pl inject.bin > duckyscript.txt
$ head -30 duckyscript.txt
DELAY 3000
GUI R

DELAY 500

ENTER

DELAY 1000
cmd
ENTER

DELAY 50
powershell
SPACE
-enc
SPACE
ZgB1AG4AYwB0AGkAbwBuACAAdwByAGkAdAB1AF8AZgBpAGwAZQBfAGIAeQB0AGU...
ENTER
powershell
SPACE
```

[2] https://code.google.com/p/ducky-decode/source/browse/trunk/ducky-decode.pl

```
-enc
SPACE
ZgB1AG4AYwB0AGkAbwBuACAAdwByAGkAdABlAF8AZgBpAGwAZQBfAGIAeQB0AGU...
ENTER
powershell
SPACE
-enc
SPACE
ZgB1AG4AYwB0AGkAbwBuACAAdwByAGkAdABlAF8AZgBpAGwAZQBfAGIAeQB0AGU...
ENTER
```

Le script exécute donc un shell Windows (`cmd`) puis une série de commandes PowerShell, en utilisant l'option `-enc` de PowerShell qui permet de passer un argument encodé en base64.

On commence par extraire les lignes de commandes PowerShell individuelles à l'aide d'un petit script Perl:

Listing 1: extract_powershell_cmdlines.pl

```
$c = 1;
while (<>) {
    next unless /.{100}/;
    s/00a0$//;
    $file = sprintf "%06d", $c;
    $c++;
    open(F, ">$file") or die;
    print F $_;
    close F
}
```

que l'on invoque ainsi:

```
$ perl extract_powershell_cmdlines.pl duckyscript.txt
$ for f in 00*; do base64 -d $f > $f.dec; done
```

On obtient 3390 fichiers de script PowerShell encodés en UTF-16. Ils sont presque tous semblables, mis à part le dernier. Voici le premier, formatté pour la clarté de lecture:

Listing 2: 000000.dec

```
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(
        'UEsDBAoDAAAAADa...
        ...yKYfPb+nXHdCvsY4S+s8AJFW2UwdXtOh6gUsBzWnXw=='));
```

```
}
else {
    write_file_bytes([Convert]::FromBase64String(
        'VAByAHkASABhAHIAZABlAHIA'));
}
```

Et voici le dernier script:

Listing 3: 003390.dec

```
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";
}
```

Si le nom de l'utilisateur Windows est approprié ("challenge2015sstic"), la série de scripts PowerShell réassemble donc un fichier "stage2.zip", et le script final vérifie son condensat SHA-1.

Plutôt que de configurer un poste Windows, il est facile de répliquer l'algorithme de construction du fichier "stage2.zip" à l'aide d'un petit script:

Listing 4: extract_stage2_chunk.pl

```
die unless /^function write_file.*?FromBase64String\('(.*?)'/;
print $1;
```

que l'on invoque ainsi:

Listing 5: rebuild_stage2_zip.sh

```
for f in 00*.dec; do
    cat $f | iconv -f utf-16le -t utf-8      \
           | perl -n extract_stage2_chunk.pl \
           | base64 -d >> stage2.zip;
done
```

À l'issue de cette opération, on vérifie que tout s'est bien passé:

```
$ sha1sum stage2.zip
ea9b8a6f5b527e72652019313c25b56ad27c7ec6  stage2.zip
```

2 FPS et AES

La deuxième étape du challenge est fournie sous forme d'une archive zip dont le contenu est le suivant:

```
$ unzip -l stage2.zip
Archive:  stage2.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
   501008  2015-03-25 17:17   encrypted
      320  2015-03-25 17:17   memo.txt
  2998438  2015-03-18 10:22   sstic.pk3
---------                     -------
  3499766                     3 files
```

Après extraction le fichier "memo.txt" nous indique la marche à suivre.

Listing 6: memo.txt

```
Cipher: AES-OFB
IV: 0x53535449433230313352d537461676532
Key: Damn... I ALWAYS forget it. Fortunately I found a way to hide it into my favorite game !

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

Il s'agit de trouver une clé AES qui nous permettra de déchiffrer le fichier "encrypted".

Le fichier "sstic.pk3" s'avère être un niveau du jeu Quake III. Le format .pk3 est en réalité une archive zip, que l'on peut inspecter. On y découvre alors un répertoire "textures/sstic" contenant une série de pièces au format graphique Targa (.tga).

Il y a en particulier 80 pièces carrées présentant chacune trois chaînes hexadécimales en couleur et un symbole, et 25 pièces rectangulaires dont 24 présentent toutes les combinaisons possibles de trois couleurs et huit symboles. Les trois couleurs sont vert, blanc, et orange. Les huit symboles sont:

- la disquette
- le drapeau
- la goutte
- l'écran
- le signal
- le wifi
- la chaîne
- le soleil

Figure 2: exemple de pièce carrée

Figure 3: exemple de pièce rectangulaire

Il est très probable que les chaînes hexadécimales des pièces carrées sont des morceaux de la clé AES recherchée. Chaque morceau fait quatre octets de long. Selon la taille de clé (128, 192, ou 256 bits) on doit sans doute mettre bout-à-bout 4, 6, ou 8 morceaux pour réassembler la clé.

Étant donné le nombre de morceaux (80*3 = 240), la combinatoire énorme exclut toute recherche exhaustive, il faut donc trouver des indices pour réduire le nombre de choix possibles.

On remarque au passage quelques particularités qui pourraient être significatives:

- seule la goutte apparaît en bas à gauche dans les pièces carrées, les autres symboles apparaissent en bas à droite
- deux pièces carrées sont répétées deux fois: un signal (110183801.tga et 522248554.tga) et un drapeau (457615433.tga et 1328144738.tga)

Ces maigres indices n'avancent pas à grand chose, il faut donc sans doute explorer le niveau du jeu. Sur Linux on peut installer OpenArena et suivre les instructions du fichier "game/README" pour lancer le jeu.

Au bout d'un certain temps passé à jouer, on recueille huit pièces carrées apparaîssant à différents endroits du niveau:

- au mur de la salle de départ
- derrière une boîte dans la grande salle
- derrière le panneau coulissant qu'on doit actionner depuis la salle de l'ordinateur
- au sol d'un ascenseur (répliqué quatre fois)
- sur les caisses dans la salle départ
- sur l'ordinateur
- sur le cube flottant dans la grande salle (il faut sauter pour l'apercevoir!)
- sous l'escalier devant le panneau coulissant

Figure 4: première pièce au mur de la salle de départ

Ceci est cohérent avec une clé de 256 bits, dont chaque morceau proviendrait d'une des pièces carrées observées dans le jeu. Chacune de ces pièces porte un symbole différent, ce qui permettrait donc d'en sélectionner une en connaissant son symbole. Reste le problème du choix de la couleur pour chaque morceau de clé.

Dans le jeu, il est fait mention d'une pièce secrète dont l'accès est limité dans le temps à quelques dizaines de secondes. Après une phase de recherche infructueuse, on se résoud à employer une technique à l'efficacité éprouvée: la triche.

On charge le niveau en "god mode" en tapant la commande `/devmap sstic` dans la console d'OpenArena et on bascule en mode passe-muraille avec `/noclip`. (Au passage, on tape aussi `/music off`.) Après quelques minutes d'exploration, on tombe sur la salle secrète, où apparaît une frise murale composées de huit pièces rectangulaires associant un symbole à une couleur.

Figure 5: frise de la salle secrète

Chaque élément de la frise indique donc un morceau de clé unique. Il ne reste plus qu'à extraire les huit morceaux de clé, les concaténer, et déchiffrer le fichier 'encrypted' en AES-OFB. Un petit script Python fait l'affaire:

Listing 7: decrypt.py

```
from Crypto.Cipher import AES

IV = b'SSTIC2015-Stage2'
key = ('9e2f31f7' +
       '8153296b' +
       '3d9b0ba6' +
       '7695dc7c' +
       'b0daf152' +
       'b54cdc34' +
       'ffe0d355' +
       '26609fac').decode('hex')

encrypted = open('encrypted', 'rb').read()

cipher = AES.new(key, AES.MODE_OFB, IV)
decrypted = cipher.decrypt(encrypted)
padding = ord(decrypted[-1])
decrypted = decrypted[:-padding]
```

```
open('decrypted', 'wb').write(decrypted)
```

On vérifie ensuite que la sortie correspond au condensat attendu:

```
$ sha256sum decrypted
845f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187  decrypted
```

3 La souris et le serpent

Un appel à la commande `file` révèle que le fichier déchiffré à l'étape 2 est une archive zip. On en liste le contenu.

```
$ file decrypted
decrypted: Zip archive data, at least v1.0 to extract

$ unzip -l decrypted
Archive:  decrypted
  Length      Date    Time    Name
---------  ---------- -----   ----
   296798  2015-03-25 17:06   encrypted
      330  2015-03-25 17:14   memo.txt
  2347070  2015-03-03 10:14   paint.cap
---------                     -------
  2644198                     3 files
```

Comme à l'étape précédente, le fichier "memo.txt" indique la marche à suivre:

Listing 8: memo.txt

```
Cipher: Serpent-1-CBC-With-CTS
IV: 0x53535449433230313552d537461676533
Key: Well, definitely can't remember it... So this time I securely stored it with Paint.

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

On ouvre le fichier "paint.cap" avec Wireshark et on voit qu'il s'agit d'une capture de communication USB avec une souris.

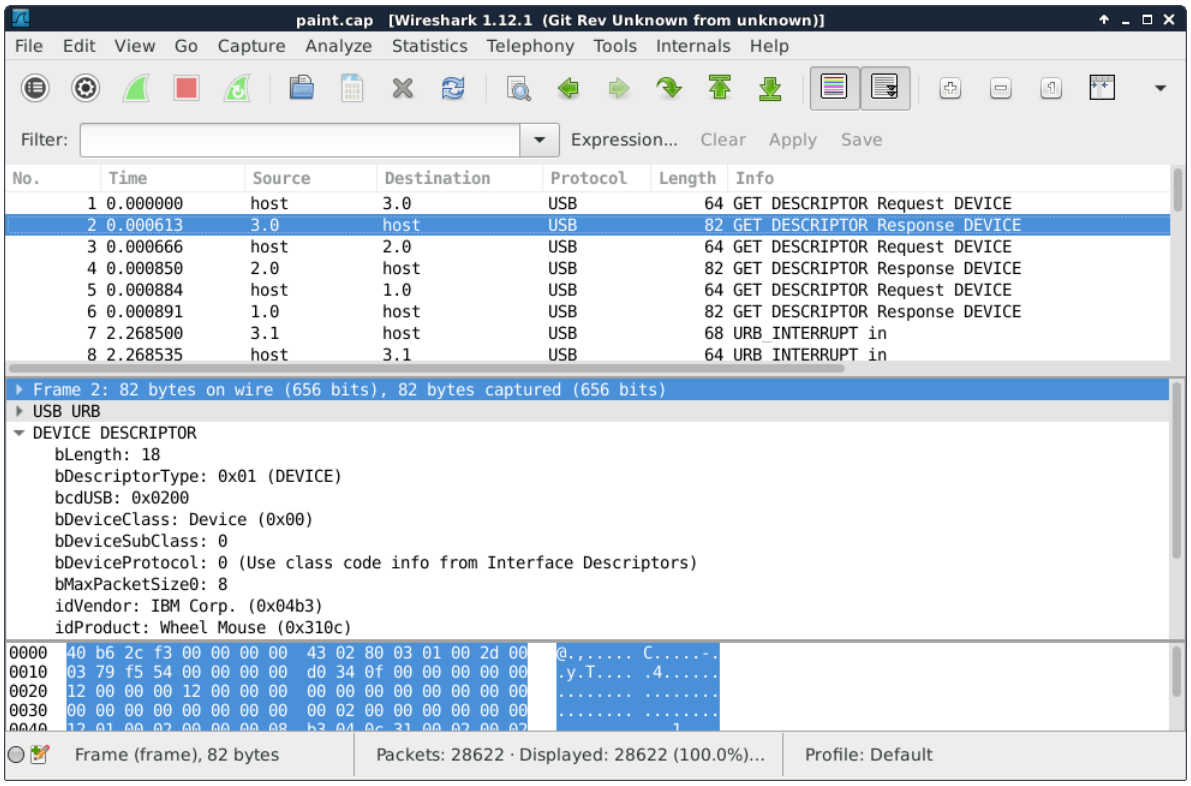


Figure 6: paint.cap dans Wireshark

La référence à Paint et la capture de souris indiquent qu'il faut reconstruire un dessin à partir de la trace USB de la souris. À l'examen des paquets, on voit que la charge utile se situe dans les quatre derniers octets des paquets à destination de l'hôte.

En cherchant un peu d'information sur le fonctionnement des souris USB, on tombe sur le code source du pilote Linux "usbmouse.c", dont voici un extrait:

Listing 9: usb_mouse_irq() dans le pilote usbmouse.c de Linux

```
static void usb_mouse_irq(struct urb *urb)
{
        struct usb_mouse *mouse = urb->context;
        signed char *data = mouse->data;
        struct input_dev *dev = &mouse->dev;

        if (urb->status) return;

        input_report_key(dev, BTN_LEFT,   data[0] & 0x01);
        input_report_key(dev, BTN_RIGHT,  data[0] & 0x02);
        input_report_key(dev, BTN_MIDDLE, data[0] & 0x04);
        input_report_key(dev, BTN_SIDE,   data[0] & 0x08);
        input_report_key(dev, BTN_EXTRA,  data[0] & 0x10);

        input_report_rel(dev, REL_X,     data[1]);
        input_report_rel(dev, REL_Y,     data[2]);
```

```
        input_report_rel(dev, REL_WHEEL, data[3]);
}
```

Il apparaît que le premier octet de données indique l'état des boutons de la souris, tandis le deuxième et le troisième indiquent le mouvement relatif selon les axes X et Y respectivement. (Le dernier octet concernant la roue est inutilisé dans notre capture.)

À la lumière de ces informations, nous pouvons reconstituer le dessin. On commence par extraire les données utiles de la capture en utilisant `tshark`, la version en ligne de commande de Wireshark. En sélectionnant uniquement les paquets de 68 octets, et en se limitant au seul champ "Leftover Capture Data", on obtient les données voulues.

```
$ tshark -Y 'frame.cap_len == 68' -r ./paint.cap -V |
  grep Leftover |
  perl -ne 'BEGIN { $by = qr/[\da-f]{2}/i }
            /($by)($by)($by)($by)$/i;
            die unless $4 eq "00";
            print "$1 $2 $3\n"' > mouse_events
$ head -5 mouse_events
00 fe 00
00 ff 00
00 fe 00
00 ff 00
00 fe 00
```

Après quoi on peut écrire un petit programme qui interprète les mouvements de la souris et l'état du bouton gauche, et reproduit le dessin final. (voir le code OCaml à l'annexe A). On l'invoque comme ceci:

```
$ cat mouse_events - | ./drawmouse
```

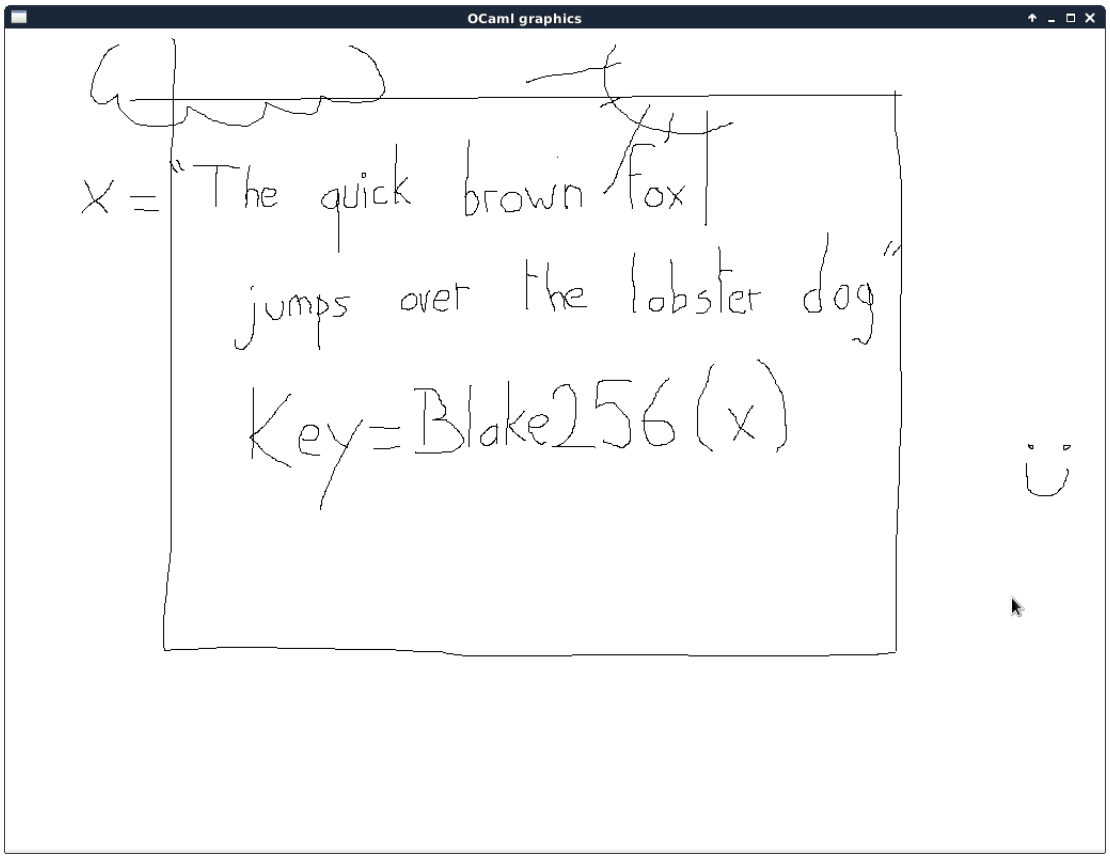


Figure 7: le dessin reconstitué d'après paint.cap

BLAKE est une fonction de hashage conçue par Jean-Philippe Aumasson, Luca Henzen, Willi Meier, et Raphael C.-W. Phan, fondée sur l'algorithme de chiffrement ChaCha [3]. BLAKE avait fait partie des cinq derniers candidats retenus dans la compétition du NIST pour désigner le standard SHA-3 (finalement gagnée par Keccak en 2012).

Il existe une implantation de la fonction de hashage BLAKE en Python, par Larry Bugbee [4], que nous utilisons.

L'algorithme de chiffrement Serpent, inventé par Ross Anderson, Eli Biham, et Lars Knudsen, était un des finalistes de la compétition AES. La bibliothèque Python CryptoPlus [5] de Philippe Teuwen offre une implantation de Serpent, mais uniquement en mode CBC standard, sans CTS.

Le mode CTS (*CipherText Stealing*) consiste à compléter un dernier bloc clair incomplet en y ajoutant des octets de bourrage issus du bloc chiffré précédent. Puis on chiffre le bloc complété, on intervertit les deux derniers blocs chiffrés, et on ampute le dernier bloc chiffré de sorte que le texte chiffré fasse la même taille que le texte clair initial.

[3] `http://en.wikipedia.org/wiki/BLAKE_%28hash_function%29`
[4] `http://www.seanet.com/~bugbee/crypto/blake/`
[5] `https://github.com/doegox/python-cryptoplus`

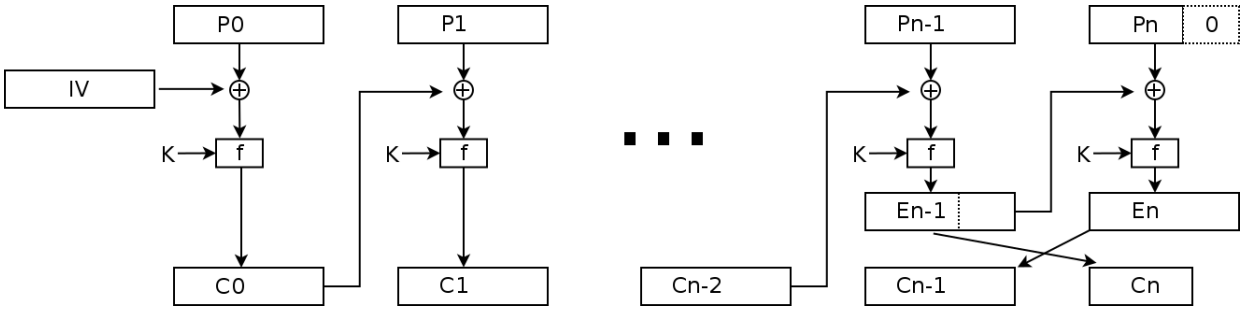


Figure 8: chiffrement en mode CBC-CTS

Lors du déchiffrement, on récupère le suffixe amputé du dernier bloc en déchiffrant l'avant-dernier bloc en mode ECB.

L'article de Wikipedia sur le mode CTS explique comment employer une API CBC pour faire du CTS; il s'agit simplement de déchiffrer un bloc supplémentaire avant de poursuivre le déchiffrement en CBC des deux derniers blocs [6].

Le script présenté en annexe B réalise le déchiffrement. On l'invoque et on vérifie le succès de l'opération:

```
$ python decrypt_serpent.py
$ sha256sum decrypted
7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2  decrypted
```

[6] http://en.wikipedia.org/wiki/Ciphertext_stealing#CBC_ciphertext_stealing_decryption_using_a_standard_CBC_interface

4 Javascript obfusqué

Le fichier déchiffré à l'étape précédente est une archive zip contenant un unique fichier nommé "stage4.html". Il s'agit d'une page Web contenant du Javascript obfusqué.

Listing 10: stage4.html

```
<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 = "2b1f25cf [... gros dump hexa ...] c1646ffe2";
        var hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c";
        $=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_:(![]+"")[$],_$_:++$,
            $_$$:({}+"")[$],$$_$:($[$]+"")[$],_$$:++$,$$$_:(!""+"")[$],$__
            :++$,$_$:++$,$$__:({}+"")[$],$$_:++$,$$$:++$,$___:++$,$__$:++$
            };$.$_=($.$_=$+"")[$.$_$]+($._$=$.$_[$.__$])+($.$$=($.$+"")[$.
            __$])+((!$)+"")[$._$$]+($.__=$.$_[$.$$_])+($.$=(!""+"")[$.__$])
            +($._=(!""+"")[$._$_])+$.$_[$.$_$]+$.__+$._$+$.$;$.$$=$.$+(!""+
            "")[$._$$]+$.__+$._+$.$+$.$$;$.$=($.___)[$.$_][$.$_];$.$($.$($.
            $$+"\""+"__="+$.$$_$+$._$+$.$$__+$._+"\\"+$.__$+$.$_$+$.$_$+$.
            $$$_+"\\"+$.__$+$.$_$+$.$$_+$.__+";$$$='\\"+$.__$+$.$$_+$._$$+$
            .__+$.$_$_+"\\"+$.__$+$.$__+$.$$$+$.$$$_+$.$_$+"';$$_$='"+(![]+
            "")[$._$_]+$._$+$.$_$_+$.$$_$+"';$_$$='\\"+$.$__+$.___+"';
            _$$$$$='"+$._+"\\"+$.__$+$.$$_+$._$$+$.$$$_+"\\"+$.__$+$.$$_+$.
            _$_+"';_$$$='"+$.$$_$+"\\"+$.__$+$.$_$+$.__$+"\\"+"
        [... quatre pages de script horribles ...]
    </script>
</body>
</html>
```

Lorsqu'on ouvre "stage4.html" dans un navigateur, il prétend effectuer un téléchargement... Allons-nous le croire? Que nenni!

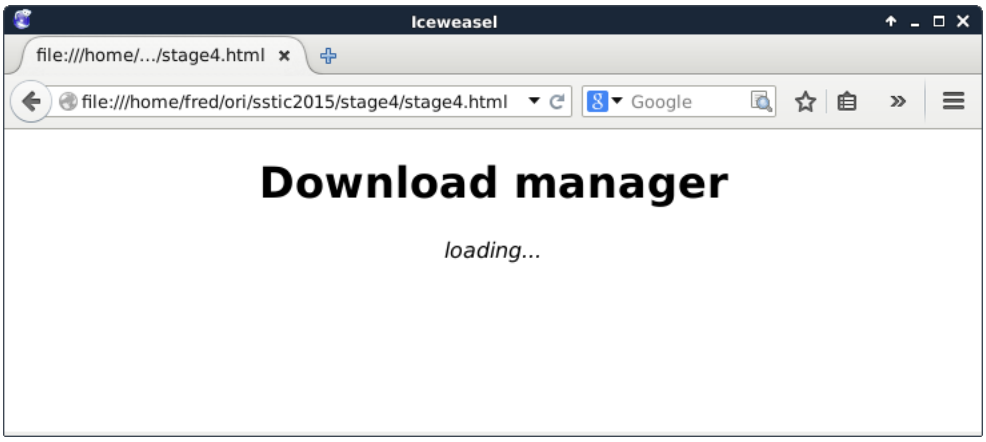


Figure 9: stage4.html dans un navigateur

La variable `data` du script contient probablement des données chiffrées, la variable `hash` un condensat de validation (similaire à ceux des étapes précédentes) et le script obsfuqué se charge de déchiffrer les données en utilisant un clé inconnue.

Il nous faut donc désobfusquer le code Javascript pour comprendre quel est l'algorithme de chiffrement employé et quelle est la clé.

désobfuscation du Javascript

Une première simplification possible est de tenter de formatter le code plus lisiblement. Pour ce faire, un certain nombre d'outils en ligne existent. J'ai employé "Javascript Beautifier" [7], qui fournit le résultat suivant:

Listing 11: script reformatté par jsbeautifier

```
var data = "[...]";
var hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c";
$ = ~[];
$ = {
    ___: ++$,
    $$$$: (![] + "")[$],
    __$: ++$,
    $_$_: (![] + "")[$],
    _$_: ++$,
    $_$$: ({} + "")[$],
    $$_$: ($[$] + "")[$],
    _$$: ++$,
    $$$_: (!"" + "")[$],
    $__: ++$,
    $_$: ++$,
    $$__: ({} + "")[$],
    $$_: ++$,
    $$$: ++$,
    $___: ++$,
    $__$: ++$
```

[7] http://jsbeautifier.org/

```
21 };
22 $.$_ = ($.$_ = $ + "")[$.$_$] + ($._$ = $.$_[$.__$]) + ($.$$ = ($.$ + "")[$
       .__$]) +
23     ((!$) + "")[$._$$] + ($.__ = $.$_[$.$$_]) + ($.$ = (!"" + "")[$.__$]) +
           (
24         $._ = (!"" + "")[$._$_]) + $.$_[$.$_$] + $.__ + $._$ + $.$;
25 $.$$ = $.$ + (!"" + "")[$._$$] + $.__ + $._ + $.$ + $.$$;
26 $.$ = ($.___)[$.$_][$.$_];
27 [...]
```

Le sens de ce début de script (lignes 3 à 21) est un peu plus clair: il construit un objet nommé `$` qui sert de dictionnaire dans le reste du code. Les champs de l'objet sont initialisés par des expressions baroques, abusant des conversions implicites fournies par le language. Le choix d'identifiants composés exclusivement des caractères `$` et `_` obscurcit encore l'intention du programmeur.

Cette technique d'obfuscation rappelle celle du Javascript non-alphanumérique de Patricio Palladino [8].

► Aparté: on divise traditionnellement les langages de programmation en trois catégories: (1) typage fort, (2) typage faible, et (3) typage pathétique. Javascript entre sans doute dans la dernière catégorie. À l'instar d'une cartomancienne, il s'efforce de donner un sens à des motifs chaotiques qui n'en ont à l'évidence aucun.

Ainsi, l'expression `~[]`, le non bit-à-bit du tableau vide, prend la valeur -1, car la valeur numérique du tableau vide est considérée comme 0.

L'expression `![]`, le non logique du tableau vide, prend la valeur `false` car la valeur logique du tableau vide est considérée comme vraie.

En revanche, l'expression `!""`, le non logique de la chaîne vide, prend la valeur `true`, car la valeur logique de la chaîne vide est considérée comme fausse.

Le reste est à l'avenant. Autant dire que l'arbitraire règne dans ce domaine, et qu'il est urgent d'oublier ces ignominies sitôt qu'on les a apprises. ◄

Les valeurs des champs de l'objet dictionnaire pourraient être reconstruites ligne par ligne en appliquant les règles de conversions de Javascript. Cela va plus vite de charger le début du script dans un débogueur Javascript, comme celui de Firefox, et d'observer le résultat produit.

[8] `http://patriciopalladino.com/blog/2012/08/09/non-alphanumeric-javascript.html`

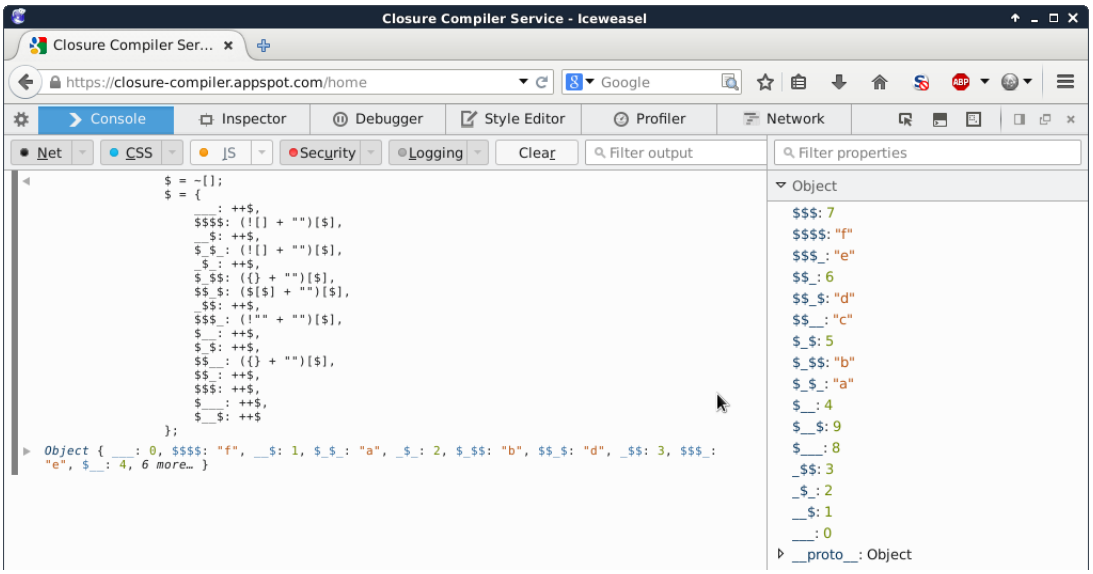


Figure 10: valeurs des champs de l'objet dictionnaire

Les valeurs du dictionnaire ne sont autres que les chiffres hexadécimaux de 0 à 0xF, sous forme numérique jusqu'à 9, puis sous forme de chaîne de caractères. (On remarque au passage que les champs sont nommés selon un code binaire où `_` représente 0 et `$` représente 1. Le nom d'un champ correspond à l'écriture binaire de la valeur qu'il contient.)

Les lignes suivantes (22 à 27) enrichissent le dictionnaire créé précédemment de quelques symboles supplémentaires dont les valeurs sont: `"return"`, `"constructor"`, les lettres `u`, `o`, et `t`, ainsi que l'objet `Function()`, défini comme le constructeur du constructeur de `0`.

L'objet `Function()` joue un rôle majeur dans la suite du script puisqu'il permet d'engendrer une fonction à partir du texte de son code. Une telle fonction est synthétisée et appelée par le script; c'est le corps de cette fonction qu'il nous faut comprendre.

Pour déterminer le corps de la fonction construite, on fait une série de passes à travers trois outils complémentaires: la console Javascript du navigateur, le compilateur optimiseur Google Closure [9], et jsbeautifier.

On évalue tout d'abord le corps de la fonction dans la console et on l'affiche avec `alert()`. On obtient un corps de fonction qui retourne une chaîne où apparaissent de nombreux caractères échappés sous forme octale, par exemple `\155` pour représenter la lettre `m`.

Listing 12: après évaluation dans la console JS du navigateur

```
return"__=docu\155e\156t;$$$='\163ta\147e5';$$_$='load';$_$$='\40';_$$$$$='
    u\163e\162';_$$$='d\151\166';$$_$$$='\156a\166\151\147ato\162';$$_$$
    ='\160\162efe\162e\156ce\163';$_$$$='to';$$$_$='\150\162ef';$$$$_='=';
    $$$$$='c\150\162o\155e'; [...]"
```

[9] `http://closure-compiler.appspot.com/home`

Puis on simplifie ce code en utilisant le compilateur Google Closure. Il faut sélectionner le mode "Advanced" et prendre soin de placer le `return` dans une fonction (pour se conformer à la syntaxe Javascript) et d'utiliser le résultat de l'appel à la fonction, par exemple avec un `alert()` (sans quoi, le code est mort et Google Closure l'élimine complètement).

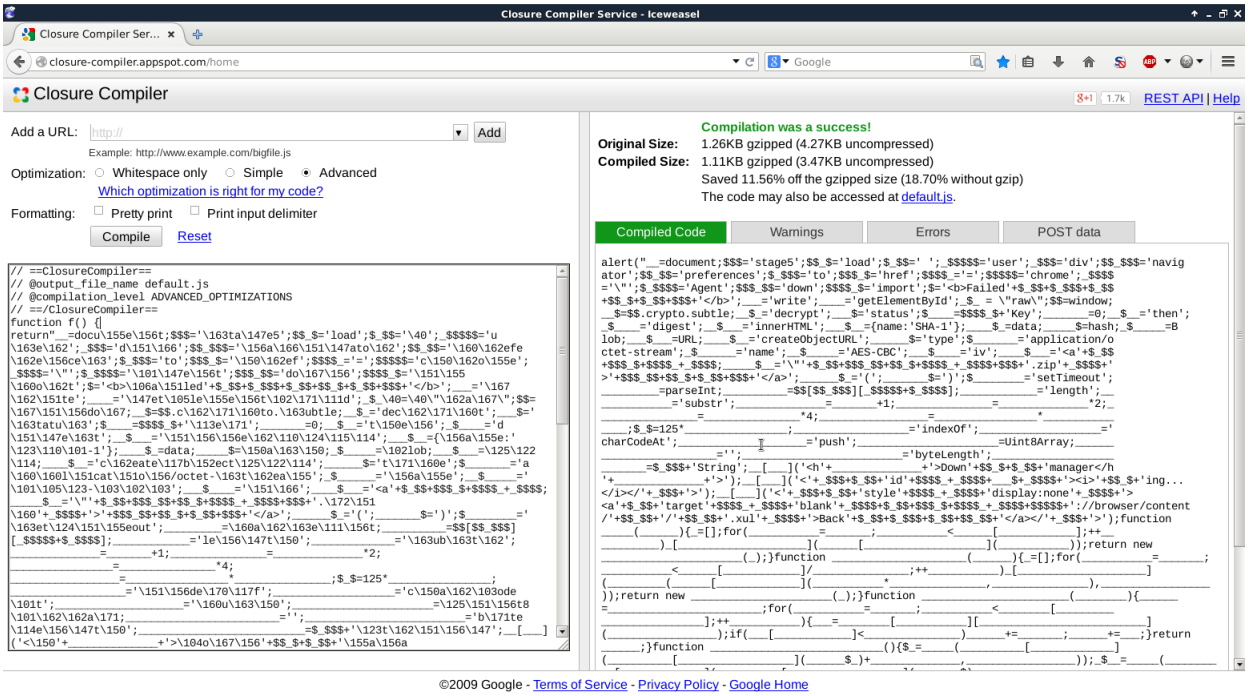


Figure 11: code simplifié par Google Closure

Une passe dans jsbeautifier aère la sortie très compacte de Google Closure, et on obtient le code suivant:

Listing 13: corps simplifié de la fonction principale

```
__ = 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></' + _$$$ + '>');
__[___]('<' + _$$$ + $_$$ + 'style' + $$$$_ + _$$$$ + 'display:none' +
    _$$$$ +
    '><a' + $_$$ + 'target' + $$$$_ + _$$$$ + 'blank' + _$$$$ + $_$$ +
    $$$_$ + $$$$_ + _$$$$ + $$$$$ + '://browser/content/' + $$_$$ + '/' +
    $$_$$ + '.xul' + _$$$$ + '>Back' + $_$$ + $_$$$ + $_$$ + $$_$$ +
```

```
    '</a></' + _$$$ + '>');

function _____(______) {
    _ = [];
    for (__________ = _______; __________ < ______[___________]; ++
        __________) _[____________________](______[____________________](
        __________));
    return new ________________________(_);
}

function ______________________(______) {
    _ = [];
    for (__________ = _______; __________ < ______[___________] /
        ________________; ++__________) _[____________________](_________(
        ______[___________](__________ * _____________,
            _____________), ________________));
    return new ________________________(_);
}

function ________________________(________) {
    ______ = ________________________;
    for (__________ = _______; __________ < ________[
            ________________________]; ++__________) {
        ___ = ________[__________][________________________](
            _________________);
        if (___[___________] < _____________) ______ += _______;
        ______ += ___;
    }
    return ______;
}

function ____________________________() {
    $_ = _____(__________[___________](__________[__________________](
        ______$_) + _____________, _________________));
    _$__ = _____(__________[___________](__________[__________________](
        _______$) - _________________, _________________));
    _$ = {};
    _$[_$______] = __$_____;
    _$[___$____] = $_;
    _$[___________] = _$__[___________] * _____________;
    __$[$____](_$_, _$__, _$, false, [__$_])[__$__](function(_$___) {
        __$[__$_](_$, _$___, ______________________(____$_))[__$__](
            function(___$_) {
                ____$ = new ______________________(___$_);
                __$[_$____](___$__, ____$)[__$__](function(
                    ____$$) {
                    if (_____$ ==
                        ________________________(new ______________________(
                            ____$$))) {
                        _____$_ = {};
                        _____$_[______$] = $_______;
                        _____$ = new _$_____([____$],
                            _____$_);
                        __$____ = ___$___[____$__](
                            _____$);
                        __[____](___$)[__$___] =
                            ____$___ + __$____ +
                            _____$__;
                    } else {
```

```
                        __[____](___$)[__$___] = $;
                    }
                });
            }).catch(function() {
                __[____](___$)[__$___] = $;
        });
    }).catch(function() {
        __[____](___$)[__$___] = $;
    });
}
$$[$________](____________________________, $_$);
```

Cela commence à prendre forme humaine. Le Javascript débute maintenant par la déclaration de constantes globales, dont on peut remplacer chaque occurrence par sa valeur fixe. Le petit script Perl suivant s'acquitte de cette tâche:

Listing 14: simplify.pl

```
$id = qr/(?:\$|_)+/;
$var = qr/[a-z_\$][a-z_\$\d.]+/i;
$str = qr/"[^"]+"|'[^']+'/;
$const = qr/\d+/;
$numexpr = qr/\d+\s*(?:\+|\*)\s*\d+/;

while (<>) {
    s/(?>($id))(?!\s*=)/exists $dict{$1} ? $dict{$1} : $1/ge;
    print;
    if (/^($id) = ($var|$str|$const);/) {
        $dict{$1} = $2;
    } elsif (/^($id) = ($numexpr);/) {
        $result = eval $2;
        $dict{$1} = $result;
    }
}
```

Après la propagation des constantes et un nouveau passage dans Google Closure et jsbeautifier, on obtient un code déjà très propre.

Listing 15: deuxième passe de simplification

```
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/preferences/preferences.xul">Back to preferences</a></div>'
);

function b(a) {
    _ = [];
    for (__________ = 0; __________ < a.length; ++__________) _.push(a.
        charCodeAt(
        __________));
    return new Uint8Array(_)
}

function c() {
    var a = data;
    _ = [];
    for (__________ = 0; __________ < a.length / 2; ++__________) _.push
        (
```

```
        parseInt(a.substr(2 * __________, 2), 16));
    return new Uint8Array(_)
}
window.setTimeout(function() {
    $_ = b(__________.substr(__________.indexOf("(") + 1, 16));
    _$__ = b(__________.substr(__________.indexOf(")") - 16, 16));
    _$ = {
        name: "AES-CBC"
    };
    _$.iv = $_;
    _$.length = 8 * _$__.length;
    window.crypto.a[$____]("raw", _$__, _$, !1, ["decrypt"]).then(
        function(a) {
            window.crypto.a.decrypt(_$, a, c()).then(function(a) {
                ____$ = new Uint8Array(a);
                window.crypto.a.digest(___$__, ____$).then(
                    function(a) {
                        var d = _____$;
                        a = new Uint8Array(a);
                        ______ =
                            _________________________;
                        for (__________ = 0; __________ <
                            a.byteLength; ++__________
                        ) ___ = a[__________][
                            _________________________
                        ](16), ______ += "write";
                        d == ______ ? (_____$_ = {
                                type: "application/octet-stream"
                            }, _____$ = new Blob([
                                ____$
                            ], _____$_), __$____ =
                            URL.createObjectURL(
                                hash), document.getElementById(
                                "status").innerHTML =
                            ____$___ + __$____ +
                            _____$__) : document.getElementById(
                            "status").innerHTML = $
                    })
            })["catch"](function() {
                document.getElementById("status").innerHTML =
                    $
            })
        })["catch"](function() {
        document.getElementById("status").innerHTML = $
    })
}, 1E3);
```

Une dernière passe de renommage manuel pour choisir les noms de fonctions et les noms de variables de boucles achève de désobfusquer le code.

Listing 16: script final

```
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/preferences/preferences.xul">Back to preferences</a></div>'
);
```

```
function u8arr_of_str(a) {
    _ = [];
    for (i = 0; i < a.length; ++i) {
        _.push(a.charCodeAt(i));
    }
    return new Uint8Array(_);
}

function u8_data() {
    var a = data;
    _ = [];
    for (i = 0; i < a.length / 2; ++i) {
        _.push(parseInt(a.substr(2 * i, 2), 16));
    }
    return new Uint8Array(_);
}
window.setTimeout(function() {
    ua = window.navigator.userAgent;
    ua1 = ua.substr(ua.indexOf("(") + 1, 16);
    iv = u8arr_of_str(ua1);
    ua2 = ua.substr(ua.indexOf(")") - 16, 16);
    key = u8arr_of_str(ua2);
    algo = { name: "AES-CBC" };
    algo.iv = iv;
    algo.length = 8 * key.length;
    window.crypto.subtle.importKey("raw", key, algo, !1, ["decrypt"])
    .then(function(sessionKey) {
        window.crypto.subtle.decrypt(algo, sessionKey, u8_data())
        .then(function(plaintext) {
            plain = new Uint8Array(plaintext);
            window.crypto.subtle.digest({ name: "SHA-1" }, plain)
            .then(function(calc_hash) {
                    var d = hash;
                    a = new Uint8Array(calc_hash);
                    v1 = "";
                    for (i = 0; i < a.byteLength; ++i) {
                        v1 += a[i].toString(16);
                    }
                    (d == v1) ?
                        (s = {},
                         s.type = "application/octet-stream",
                         blob = new Blob([ plain ], s),
                         url = URL.createObjectURL(hash),
                         document.getElementById("status").innerHTML =
                          '<a href="' + url + '" download="stage5.zip">
                              download stage5</a>'
                        ) :
                        document.getElementById("status").innerHTML =
                                              "<b>Failed to load stage5</b>";
                });
        })
        .catch(function(ex) {
            document.getElementById("status").innerHTML =
                "<b>Failed to load stage5</b>";
        });
    })
    .catch(function(ex) {
        document.getElementById("status").innerHTML =
            "<b>Failed to load stage5</b>";
```

```
    });
}, 1000);
```

Le fonctionnement du script est désormais clair: il emploie l'API WebCrypto [10] pour déchiffrer le contenu de la variable `data` en AES-CBC. Deux portions du *User Agent* (l'identifiant du navigateur) sont utilisées comme clé et vecteur initial. Enfin un condensat SHA-1 du résultat est calculé et comparé au contenu de la variable `hash` pour s'assurer du succès de l'opération. Le fichier déchiffré est rendu disponible sous le nom de "stage5.zip".

déchiffrement

Le but est donc maintenant de trouver la clé de déchiffrement et le vecteur initial (IV) qui fourniront le bon fichier déchiffré. L'IV correspond aux 16 caractères suivant la première parenthèse ouvrante du User Agent. La clé correspond aux 16 caractères précédant la première parenthèse fermante.

Un User Agent typique ressemble à la chaîne suivante:

```
Mozilla/5.0 (X11; Linux x86_64; rv:31.0) Gecko/20100101 Firefox/31.0 Iceweasel/31.5.0
```

La partie entre parenthèses est la *plate-forme*, qui inclut des données sur le système d'exploitation sur lequel tourne le navigateur, éventuellement des informations sur la *locale* (langue et pays), et éventuellement un identifiant du moteur du navigateur.

Une hypothèse raisonnable est que le User Agent recherché est légitime, dans le sens où il existe réellement une combinaison d'un système et d'un navigateur susceptible de le produire. [11] Dans ce cas, on peut effectuer une attaque hybride entre l'attaque par dictionnaire et l'attaque par force brute: envisager toutes les combinaisons possibles entre une liste de systèmes connus et une liste de navigateurs connus.

On peut faire une première collecte d'éléments de ces listes sur Internet, quitte à en générer en plus, selon les schémas observés empiriquement, pour "boucher les trous" et tenter d'être exhaustif.

Un grand nombre de combinaisons semblent possibles a priori, mais un indice permet de réduire le périmètre de recherche: il s'agit de l'URL caché dans le Javascript obfusqué qui fait référence à la page "preferences.xul".

XUL (XML User interface Language) est une technologie de Mozilla, qui n'a donc cours que dans les navigateurs Firefox et dérivés. (Le protocole `chrome://` dans l'URL ne fait pas ici référence au navigateur Google Chrome, mais au nom d'une partie de l'interface graphique de Firefox [12] .)

Nous nous focalisons donc sur la génération d'agents Firefox, en suivant le format décrit par la documentation de Mozilla. [13] Les versions existantes de Firefox sont également documentées. [14].

Le script d'énumération des User Agent est présenté en annexe C. Pour accélérer le traitement, on déchiffre d'abord les premiers blocs, puis on cherche l'en-tête PK annonçant

[10] `http://www.w3.org/TR/WebCryptoAPI/`

[11] Notons cependant que ce n'est pas la seule hypothèse raisonnable. Notamment si on a remarqué que l'IV de l'étape 2 est "SSTIC2015-Stage2", que l'IV de l'étape 3 est "SSTIC2015-Stage3", il semble légitime de postuler que l'étape 4 emploie l'IV "SSTIC2015-Stage4". Cela s'avère une fausse piste...

[12] `https://developer.mozilla.org/en-US/docs/Glossary/Chrome`

[13] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Gecko_user_agent_string_reference`

[14] `https://www.mozilla.org/en-US/firefox/releases/`

une archive zip, et s'il est bien présent au début du texte clair on continue le déchiffrement et on vérifie le SHA-1.

```
$ python findua.py
[...]
Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:35.0)
504b0304140300000800ed897846829c764236d703009bdc030009000000696e
===
got it!
```

Ceci conclue l'étape 4.

5 Transputeurs

Voyons le contenu de l'archive "stage5.zip":

```
$ unzip -l stage5.zip
Archive:  stage5.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
   253083  2015-03-24 17:15   input.bin
    13089  2015-03-25 13:19   schematic.pdf
---------                     -------
   266172                     2 files
```

Le fichier "schematic.pdf" contient un plan d'architecture matérielle évoquant des "transputers" (qu'on francisera en *transputeurs*...) disposés dans une topologie particulière, ainsi qu'un vecteur de test pour ce qui doit être une opération de déchiffrement.

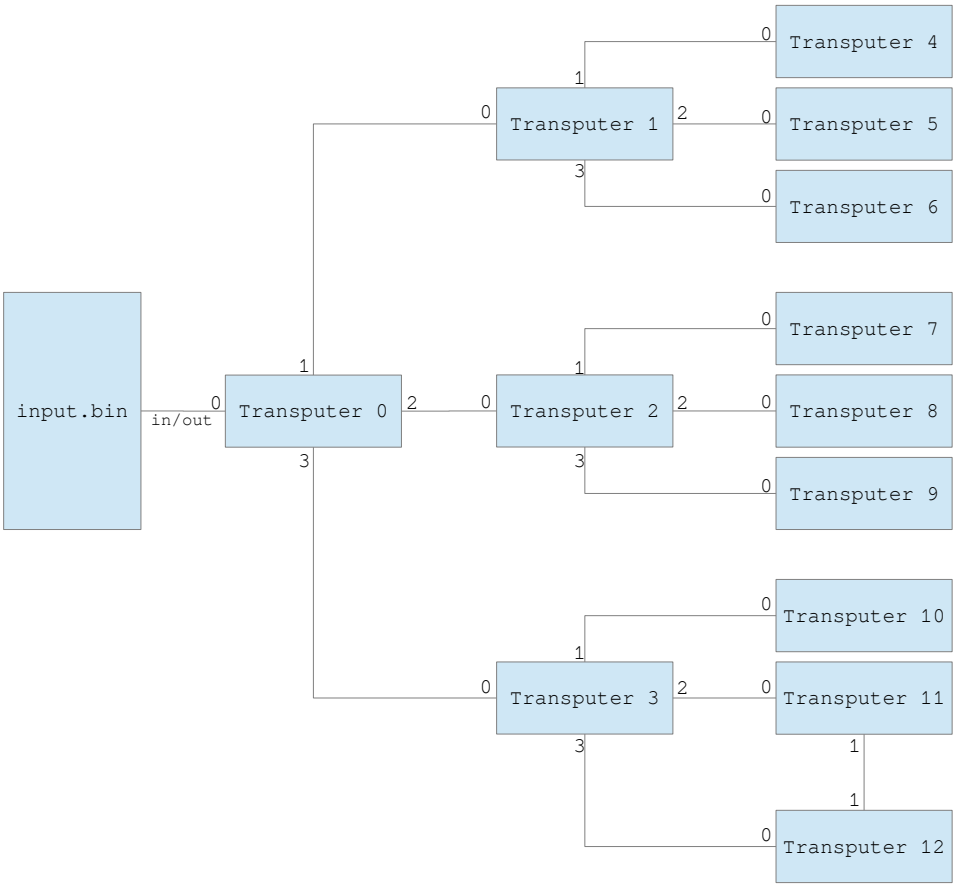
La mention de "ST20" dans le vecteur de test précise la nature des transputeurs, et nous permet de nous documenter sur le processeur en question [15]. Il s'agit d'un processeur de SGS-THOMSON datant des années 90. On trouve le manuel du processeur [16] et la référence du jeu d'instructions [17].

Hormis les fonctionnalités usuelles d'un processeur normal, la particularité d'un transputeur est d'être connectable à des pairs par des canaux de communication dédiés. Correctement programmés, les transputeurs peuvent travailler en parallèle et se synchroniser en échangeant des messages avec leurs voisins. Le ST20, quant à lui, peut être connecté à jusqu'à quatre voisins.

En cherchant sur Internet, on trouve aussi un émulateur de ST20 par "tempbsp" sur sourceforge [18], dont la lecture est très intéressante. Malheureusement cet émulateur ne gère pas les instructions `in` et `out`, or le calcul parallèle semble être la clé de cette épreuve.

Le fichier schematic contient aussi les condensats SHA-256 d'un texte chiffré à découvrir ("encrypted"), et du texte clair correpondant ("decrypted").

[15] http://en.wikipedia.org/wiki/Transputer#ST20
[16] http://pdf.datasheetcatalog.com/datasheet/stmicroelectronics/4942.pdf
[17] http://pdf.datasheetcatalog.com/datasheet/SGSThomsonMicroelectronics/mXruvtu.pdf
[18] http://st20emu.sourceforge.net/



SHA256:
    a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81 - encrypted
    9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321 - decrypted

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

Figure 12: schematic.pdf

On survole rapidement le second fichier, "input.bin", à l'aide d'un éditeur hexadécimal.

```
$ xxd input.bin | less
```

Une première partie paraît contenir des données de densité moyenne, qui pourraient être du code. Y apparaissent aussi quelques chaînes de caractères: "Boot ok", "Code Ok", "Decrypt", "KEY:", et "congratulations.tar.bz2". Après la chaîne "KEY:" viennent douze octets fixés à 0xFF. Le reste du fichier semble contenir des données de forte entropie, qui pourraient correspondre à un texte chiffré.

On extrait la fin de "input.bin" et on vérifie qu'il s'agit du chiffré dont le condensat est fourni par "schematic.pdf".

```
$ dd if=input.bin of=encrypted bs=1 skip=$((0x9ad))
250606+0 records in
250606+0 records out
250606 bytes (251 kB) copied, 0.299247 s, 837 kB/s
$ sha256sum encrypted
a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81  encrypted
```

En désassemblant le début de "input.bin" dans IDA Pro (il faut sélectionner le processeur "st20 c2" pour que toutes les instructions soient reconnues) on vérifie qu'il s'agit bien de code ST20 et on commence à faire des hypothèses sur la logique du code: peut-être que les instructions `in` et `out` lisent des bouts de données chiffrées et rendent les données déchiffrées?

Pour résumer, il semble bien qu'il nous faille analyser le code ST20 pour déterminer l'algorithme de déchiffrement utilisé, puis trouver la bonne clé pour récupérer le texte clair de "congratulations.tar.bz2".

À ce stade, plusieurs approches sont envisageables: on peut analyser le code ST20 manuellement sous IDA ou un équivalent. On peut envisager d'émuler le code, peut-être est-ce suffisant pour résoudre le problème en boîte noire. On pourrait aussi tenter de traduire le code dans un langage intermédiaire de plus haut niveau pour en faciliter la lecture, ou même en faire une interprétation symbolique pour récupérer l'algorithme de déchiffrement de cette façon.

J'ai choisi l'approche de l'émulateur car elle permet de se familiariser avec le jeu d'instructions du processeur - étape nécessaire de toute façon - et elle fournit par la suite des indices sur le comportement dynamique du code qui peuvent être difficiles à déceler par l'approche purement statique.

Émulateur ST20

L'émulateur se divise en deux parties: désassembleur et interpréteur (syntaxe et sémantique). Le désassembleur se compose d'une partie en C chargée du décodage des instructions, très largement inspirée du code équivalent dans l'émulateur ST20 de "tempbsp", et d'une partie en OCaml traduisant les numéros d'opcodes en instructions symboliques.

Le sous-ensemble du jeu d'instructions connu de l'émulateur est le suivant:

Listing 17: jeu d'instructions émulé

```
type insn =
    Ajw of int32    (* creation d'un cadre de pile *)
  | Ldc of int32    (* charge une constante *)
```

```
  | Adc of int32     (* ajoute une constante *)
  | Stl of int32     (* stockage dans une variable locale *)
  | Mint             (* charge l'entier minimum 0x80000000 *)
  | Ldnlp of int32   (* charge un pointeur *)
  | Gajw             (* initialisation de wptr *)
  | Ldpi             (* charge une adresse relative a iptr *)
  | In               (* lit un canal *)
  | Out              (* ecrit sur un canal *)
  | Ldlp of int32    (* charge un pointeur de variable locale *)
  | Ldl of int32     (* charge une variable locale *)
  | Cj of int32      (* saut conditionnel *)
  | J of int32       (* saut *)
  | Lb               (* charge un octet *)
  | Sb               (* stocke un octet *)
  | Xor
  | Bsub             (* ptr dans un tableau d'octets *)
  | Ssub             (* ptr dans un tableau d'entiers 16 bits *)
  | Dup
  | Eqc of int32
  | Gcall            (* appel via registre A *)
  | Call of int32
  | Ret
  | And
  | Gt
  | Shr
  | Shl
  | Prod
  | Wsub             (* ptr dans un tableau d'entiers 32 bits *)
  | Rem              (* modulo *)
  | Unknown
```

L'interpréteur, écrit en OCaml, agit sur un état du système englobant tous les transputeurs et tous les canaux de communications. À chaque tic d'horloge, chaque transputeur exécute une instruction ou attend un message d'un transputeur voisin. Après chaque tic d'horloge, les communications inter-transputeurs sont résolues et les transputeurs en attente d'un message sur un canal sont débloqués si des données sont apparues sur ce canal.

Les états possibles d'un transputeur individuel sont:

- Boot: le transputeur est en attente du programme initial (*bootstrap*) sur son canal d'entrée 0 ;
- Run: le transputeur est en cours d'exécution du programme ;
- In: le transputeur attend un message sur un canal d'entrée ;
- Out: le transputeur attend la réception d'un message écrit sur un canal de sortie.

Le ST20 est capable de booter dans deux modes différents: "boot from ROM" et "boot from link". Dans le premier cas, classique, le programme initial est chargé depuis une ROM située à une adresse fixe en mémoire. Dans le second cas, celui qui nous intéresse, le programme initial provient du canal 0 du transputeur.

Les parties intéressantes de la documentation sont à la page 68, Section 11.2.2, "Booting from link":

When booting from a link, the ST20-GP1 will wait for the first bootstrap message to arrive on the link. The first byte received down the link is the control byte. If the control byte is greater than 1 (i.e. 2 to 255), it is taken as the length in bytes of the boot code to be loaded down the link. The bytes following the control byte are then placed in internal memory starting at location MemStart. Following reception of the last byte the ST20-GP1 will start executing code at MemStart. The memory space immediately above the loaded code is used as work space. A byte arriving on the bootstrapping link after the last bootstrap byte, is retained and no acknowledge is sent until a process inputs from the link.

Ceci explique la mystérieuse valeur 0xf8 au début de "input.bin", faussement désassemblée en l'instruction `prod`, qui est en réalité la longueur du programme initial du transputeur 0.

Si la configuration "boot from link" est uniforme, les programmes initiaux des différents coeurs doivent donc être disséminés dans la grille de transputeurs, de voisin en voisin à partir de la source "input.bin" connectée au transputeur 0.

Dans le ST20, le pile du processeur s'appelle le *workspace*. Elle contient les variables locales. Le register `wptr` est le *workspace pointer* - l'équivalent de `ebp` en x86.

Le pointeur d'instruction s'appelle `iptr`.

Les adresses mémoire sont signées, l'adresse la plus basse est donc 0x80000000, soit -2^{31}. Ceci explique l'utilisation de l'instruction `mint`, qui renvoie 0x80000000, pour construire des adresses mémoires de canaux.

Les registres de calcul sont au nombre de trois, nommés A, B, et C, et organisés en pile.

L'implantation du workspace et de la pile des registres dans l'émulateur ne présente pas de difficulté particulière.

Les accès aux canaux de communication sont déclenchés par les instructions `in` et `out`. En général le modèle du rendez-vous est retenu pour la synchronisation entre les transputeurs. Le transputeur expéditeur d'un message attend que le transputeur destinataire tente une lecture d'un message de même taille, et vice versa. Les seules exceptions sont la gestion de la phase de boot, et le cas spécial du lien d'entrée/sortie pour le transputeur 0.

D'après les descriptions de `in` et `out` dans le manuel de référence du jeu d'instructions, et leur usage dans le code du challenge, on infère assez rapidement la correspondance entre les canaux et les adresses mémoire qui leur sont associées. Les canaux d'écriture vont de 0x80000000 à 0x8000000c, et les canaux de lecture de 0x80000010 à 0x8000001c.

Listing 18: correspondance entre adresse et numéro de canal

```
let chan_id_of_addr = function
  | 0x8000_0000l
  | 0x8000_0010l -> 0
  | 0x8000_0004l
  | 0x8000_0014l -> 1
  | 0x8000_0008l
  | 0x8000_0018l -> 2
  | 0x8000_000cl
  | 0x8000_001cl -> 3
  | _ -> failwith "invalid channel address"
```

```
;;
```

Dans l'émulateur on peut modéliser un canal monodirectionnel par un objet liant un processeur source à un processeur destination, et contenant, éventuellement, un message en attente de transmission. Chaque lien entre deux transputeurs est donc représenté par deux canaux, un pour chaque direction.

L'entrée principale de l'émulateur est le fichier "input.bin" (ou son équivalent pour le vecteur de test).

Il est utile d'ajouter certaines options à l'émulateur, notamment les suivantes:

- `-go`: session non-interactive
- `-q`: session silencieuse
- `-sink`: pour sauvergarder la sortie du transputeur 0 dans un fichier
- `-key`: pour changer la clé de déchiffrement dans le fichier d'entrée
- `-save`: pour sauvegarder les programmes initiaux de tous les transputeurs
- `-msgs`: pour sauvegarder tous les messages échangés

Le code de l'émulateur est présenté en annexe D.

Voici un exemple de début de session interactive:

```
$ ./prog.opt input.bin
-1 -> 0: read byte f8
got 248 bytes of bootstrap code; starting
1: waiting for bootstrap code
2: waiting for bootstrap code
3: waiting for bootstrap code
4: waiting for bootstrap code
5: waiting for bootstrap code
6: waiting for bootstrap code
7: waiting for bootstrap code
8: waiting for bootstrap code
9: waiting for bootstrap code
10: waiting for bootstrap code
11: waiting for bootstrap code
12: waiting for bootstrap code

A=???????? B=???????? C=???????? Wptr=80001000
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
0: 80000140 ajw fffffffb4

A=???????? B=???????? C=???????? Wptr=80000ed0
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
0: 80000142 ldc 0

A=00000000 B=???????? C=???????? Wptr=80000ed0
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
0: 80000143 stl 1
```

```
A=???????? B=???????? C=???????? Wptr=80000ed0
|4d 4d 4d 4d 00 00 00 00 4d 4d 4d 4d 4d 4d 4d 4d|MMMM    MMMMMMMM
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
0: 80000144 ldc 0

A=00000000 B=???????? C=???????? Wptr=80000ed0
|4d 4d 4d 4d 00 00 00 00 4d 4d 4d 4d 4d 4d 4d 4d|MMMM    MMMMMMMM
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
0: 80000145 stl 3

A=???????? B=???????? C=???????? Wptr=80000ed0
|4d 4d 4d 4d 00 00 00 00 4d 4d 4d 4d 00 00 00 00|MMMM    MMMM
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
0: 80000146 mint

A=80000000 B=???????? C=???????? Wptr=80000ed0
|4d 4d 4d 4d 00 00 00 00 4d 4d 4d 4d 00 00 00 00|MMMM    MMMM
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
0: 80000148 ldnlp 400

A=80001000 B=???????? C=???????? Wptr=80000ed0
|4d 4d 4d 4d 00 00 00 00 4d 4d 4d 4d 00 00 00 00|MMMM    MMMM
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
0: 8000014b gajw

A=???????? B=???????? C=???????? Wptr=80001000
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
0: 8000014d ajw fffffb4

A=???????? B=???????? C=???????? Wptr=80000ed0
|4d 4d 4d 4d 00 00 00 00 4d 4d 4d 4d 00 00 00 00|MMMM    MMMM
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
0: 8000014f ldc c9

A=000000c9 B=???????? C=???????? Wptr=80000ed0
|4d 4d 4d 4d 00 00 00 00 4d 4d 4d 4d 00 00 00 00|MMMM    MMMM
|4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d|MMMMMMMMMMMMMMMM
0: 80000151 ldpi
```

Étude en boîte noire

Armé de notre émulateur, nous souhaitons l'essayer sur le vecteur de test. On commence par produire un fichier "testvec.bin" dérivé de "input.bin", où l'on a remplacé la clé et le texte chiffré par ceux fournis dans le vecteur de test, à l'aide d'un éditeur hexadécimal.

```
$ xxd testvec.bin | tail -5
0000980: 0000 0000 004b 4559 3a2a 5353 5449 432d  .....KEY:*SSTIC-
0000990: 3230 3135 2a17 636f 6e67 7261 7475 6c61  2015*.congratula
00009a0: 7469 6f6e 732e 7461 722e 627a 321d 87c4  tions.tar.bz2...
00009b0: c4e0 ee40 383c 5944 7f23 798d 9fef e74f  ...@8<YD.#y....O
00009c0: b824 8076 6e                             .$.vn
```

Puis on lance l'émulateur:

```
$ ./prog.opt -go testvec.bin | grep "Write 0..-1"
Write 0->-1: 42 6f 6f 74 20 6f 6b 00 [length = 8]
Write 0->-1: 43 6f 64 65 20 4f 6b 00 [length = 8]
```

```
Write 0->-1: 44 65 63 72 79 70 74 84 [length = 8]
Write 0->-1: 49 I
Write 0->-1: 20
Write 0->-1: 6c l
Write 0->-1: 6f o
Write 0->-1: 76 v
Write 0->-1: 65 e
Write 0->-1: 20
Write 0->-1: 53 S
Write 0->-1: 54 T
Write 0->-1: 32 2
Write 0->-1: 30 0
Write 0->-1: 20
Write 0->-1: 61 a
Write 0->-1: 72 r
Write 0->-1: 63 c
Write 0->-1: 68 h
Write 0->-1: 69 i
Write 0->-1: 74 t
Write 0->-1: 65 e
Write 0->-1: 63 c
Write 0->-1: 74 t
Write 0->-1: 75 u
Write 0->-1: 72 r
Write 0->-1: 65 e
```

Après les messages "Boot ok", "Code Ok", "Decrypt", on obtient la sortie attendue, émise octet par octet sur le lien de sortie du transputeur 0. Cela se présente bien. Voyons l'effet d'un changement du premier octet de la clé, changé de 0x2a à 0x2b, par exemple:

```
$ ./prog.opt -go testvec_mod1.bin | grep "Write 0..-1"
Write 0->-1: 42 6f 6f 74 20 6f 6b 00 [length = 8]
Write 0->-1: 43 6f 64 65 20 4f 6b 00 [length = 8]
Write 0->-1: 44 65 63 72 79 70 74 84 [length = 8]
Write 0->-1: 4b K
Write 0->-1: 20
Write 0->-1: 6c l
Write 0->-1: 6f o
Write 0->-1: 76 v
Write 0->-1: 65 e
Write 0->-1: 20
Write 0->-1: 53 S
Write 0->-1: 54 T
Write 0->-1: 32 2
Write 0->-1: 30 0
Write 0->-1: 20
Write 0->-1: 09 .
Write 0->-1: aa .
Write 0->-1: 91 .
Write 0->-1: e0 .
Write 0->-1: f5 .
Write 0->-1: 8a .
Write 0->-1: a1 .
Write 0->-1: f9 .
Write 0->-1: 66 f
Write 0->-1: 93 .
Write 0->-1: c8 .
Write 0->-1: 69 i
```

On observe un changement du premier octet dans le premier bloc de douze octets, mais les onze octets suivants sont inchangés. En revanche, le second bloc de douze octets est entièrement modifié.

En répétant le test avec tous les octets de la clé un par un, on remarque que l'influence d'un octet de clé semble se limiter à un seul octet du premier bloc déchiffré, celui au même index que l'octet de clé. En revanche, tous les octets de clé influent sur le second bloc.

Si nous avions un indice sur le contenu du texte clair, cette relation entre un octet de clé et un octet du premier bloc déchiffré pourrait être exploitée pour retrouver la clé par le biais d'une attaque à texte clair connu. Or le nom de fichier "congratulations.tar.bz2" nous fournit un tel indice: on peut supposer que le début du texte clair est un en-tête bzip2.

Wikipedia nous renseigne sur le format de fichier bzip2 [19]:

Listing 19: en-tête bzip2

```
.magic:16              = 'BZ' signature/magic number
.version:8             = 'h' for Bzip2 ('H'uffman coding),
                         '0' for Bzip1 (deprecated)
.hundred_k_blocksize:8 = '1'..'9' block-size 100 kB-900 kB
.compressed_magic:48   = 0x314159265359 (BCD (pi))
.crc:32                = checksum for this block
.randomised:1          = 0=>normal, 1=>randomised (deprecated)
.origPtr:24            = starting pointer into BWT
```

Huit des douze premiers octets sont donc constants, et deux autres, 'h' et '9', sont presque certainement connus.

On peut vérifier empiriquement, par exemple en compressant les fichiers de `/usr/bin` avec bzip2, que les dix premiers de l'en-tête semblent constants en pratique.

```
$ for f in *.bz2; do head -c 10 $f | xxd; done | sort -u
0000000: 425a 6839 3141 5926 5359                  BZh91AY&SY
```

Quant aux onzième et douzième octets, ils correspondent à un CRC32 et sont donc inconnus a priori. Cependant une attaque par force brute sur deux octets est tout à fait praticable. Si la relation entre les octets de clé et les octets du premier bloc déchiffré est avérée, l'énumération exhaustive de toutes les clés possibles semble faisable.

Pour découvrir les octets de clé produisant les bonnes valeurs de sortie, on peut essayer de déchiffrer avec une clé composée de douze octets à 0x00, puis avec une clé composée de douze octets à 0x01, puis à 0x02, etc. et à chaque étape regarder quels octets dans la sortie correspondent à un en-tête bzip2. (On travaille sous l'hypothèse que les octets de clé sont indépendants les uns des autres.)

```
$ perl -e 'for ($i = 0; $i < 256; $i++) { $by = sprintf
  qq(%02x), $i; print $by x 16, "\n" }' | xargs -I '{}'
  echo echo '{}'\; ./prog.opt -go -key '{}' -sink '>(dd
  bs=1 skip=24 count=16 | xxd)' -q input.bin > tmp.sh
$ head -3 tmp.sh
echo 00000000000000000000000000000000; ./prog.opt -go
-key 00000000000000000000000000000000
-sink >(dd bs=1 skip=24 count=16 | xxd) -q input.bin
echo 01010101010101010101010101010101; ./prog.opt -go
```

[19] http://en.wikipedia.org/wiki/Bzip2#File_format

```
-key 01010101010101010101010101010101
-sink >(dd bs=1 skip=24 count=16 | xxd) -q input.bin
echo 02020202020202020202020202020202; ./prog.opt -go
-key 02020202020202020202020202020202
-sink >(dd bs=1 skip=24 count=16 | xxd) -q input.bin
$ ./tmp.sh 2>/dev/null | head -6
00000000000000000000000000000000
0000000: fef2 52df 85b9 9120 81a5 7823 cb51 a60a  ..R.... ..x#.Q..
01010101010101010101010101010101
0000000: fcf0 54d9 87bb 9f2e 83a7 7e25 1b05 2a48  ..T.......~%..*H
02020202020202020202020202020202
0000000: faf6 56db 89b5 9d2c 85a1 7c27 6bfd 02e2  ..V....,..|'k...
```

Voyons les antécédents du 'B' commençant l'en-tête bzip2:

```
$ ./tmp.sh 2>/dev/null | grep -B1 " B"
5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e
0000000: 424e ee63 417d 55e4 4d69 b4ef 2b55 eaa2  BN.cA}U.Mi..+U..
--
dedededededededededededededede
0000000: 424e ee63 417d 55e4 4d69 b4ef 2b55 ea54  BN.cA}U.Mi..+U.T
```

Il y en a deux: le premier octet de clé peut valoir 0x5e ou 0xde.

On constate de même pour les autres positions que le bit de poids fort d'un octet de clé n'a pas d'influence sur le premier bloc déchiffré. En revanche, il influe sur les blocs suivants.

Cela est gênant dans le cadre d'une attaque par force brute, car il faut envisager deux cas pour chacun des dix premiers octets de la clé, en plus de l'énumération exhaustive des deux derniers octets. En tout cela fait $2^{10} * 256^2$, soit environ 65 millions de possibilités. Notre émulateur est beaucoup trop lent pour réaliser une attaque de cette ampleur.

Faisons une collecte des programmes initiaux et des messages pour glaner quelques informations supplémentaires:

```
$ ./prog.opt -save -msgs input.bin
$ md5sum bootstrap.*
66168daacab8f12aa94f665bbbc4f787  bootstrap.0
0cbeb2a7a4f269356aeec0fe6bfc9192  bootstrap.1
3baf20a3228b40f92da74bc37f1655b1  bootstrap.10
3baf20a3228b40f92da74bc37f1655b1  bootstrap.11
3baf20a3228b40f92da74bc37f1655b1  bootstrap.12
0cbeb2a7a4f269356aeec0fe6bfc9192  bootstrap.2
0cbeb2a7a4f269356aeec0fe6bfc9192  bootstrap.3
3baf20a3228b40f92da74bc37f1655b1  bootstrap.4
3baf20a3228b40f92da74bc37f1655b1  bootstrap.5
3baf20a3228b40f92da74bc37f1655b1  bootstrap.6
3baf20a3228b40f92da74bc37f1655b1  bootstrap.7
3baf20a3228b40f92da74bc37f1655b1  bootstrap.8
3baf20a3228b40f92da74bc37f1655b1  bootstrap.9
```

On constate que transputeurs "intermédiaires", les 1, 2, et 3 partagent le même programme initial. De même les transputeurs situés en bout de chaîne, les 4 à 12, démarrent avec le même programme.

En observant les messages échangés entre les transputeurs, on peut faire la distinction entre de petits messages de 1 ou 12 octets, correspondant à des données, et certains messages plus longs, qui surviennent uniquement au début de l'émulation et pourraient correspondre à du code.

(Les messages ci-dessous sont nommées suivant le schéma msg_[source]_[destination]_-[numéro du message])

```
$ ls -l msg*
-rw-r--r-- 1 fred fred  37 Apr 11 07:03 msg_0_1_11.bin
-rw-r--r-- 1 fred fred  37 Apr 11 07:03 msg_0_1_15.bin
-rw-r--r-- 1 fred fred  37 Apr 11 07:03 msg_0_1_19.bin
-rw-r--r-- 1 fred fred  80 Apr 11 07:03 msg_0_1_47.bin
-rw-r--r-- 1 fred fred  80 Apr 11 07:03 msg_0_1_53.bin
-rw-r--r-- 1 fred fred 140 Apr 11 07:03 msg_0_1_59.bin
-rw-r--r-- 1 fred fred  37 Apr 11 07:03 msg_0_2_23.bin
-rw-r--r-- 1 fred fred  37 Apr 11 07:03 msg_0_2_27.bin
-rw-r--r-- 1 fred fred  37 Apr 11 07:03 msg_0_2_31.bin
-rw-r--r-- 1 fred fred 100 Apr 11 07:03 msg_0_2_65.bin
-rw-r--r-- 1 fred fred 156 Apr 11 07:03 msg_0_2_71.bin
-rw-r--r-- 1 fred fred  84 Apr 11 07:03 msg_0_2_77.bin
-rw-r--r-- 1 fred fred  37 Apr 11 07:03 msg_0_3_35.bin
-rw-r--r-- 1 fred fred  37 Apr 11 07:03 msg_0_3_39.bin
-rw-r--r-- 1 fred fred  37 Apr 11 07:03 msg_0_3_43.bin
-rw-r--r-- 1 fred fred 152 Apr 11 07:03 msg_0_3_83.bin
-rw-r--r-- 1 fred fred 112 Apr 11 07:03 msg_0_3_89.bin
-rw-r--r-- 1 fred fred 132 Apr 11 07:03 msg_0_3_95.bin
$ ls -l endpoints/msg*
-rw-r--r-- 1 fred fred  68 Apr 11 07:03 endpoints/msg_1_4_51.bin
-rw-r--r-- 1 fred fred  68 Apr 11 07:03 endpoints/msg_1_5_57.bin
-rw-r--r-- 1 fred fred 128 Apr 11 07:03 endpoints/msg_1_6_63.bin
-rw-r--r-- 1 fred fred  88 Apr 11 07:03 endpoints/msg_2_7_69.bin
-rw-r--r-- 1 fred fred 144 Apr 11 07:03 endpoints/msg_2_8_75.bin
-rw-r--r-- 1 fred fred  72 Apr 11 07:03 endpoints/msg_2_9_81.bin
-rw-r--r-- 1 fred fred 140 Apr 11 07:03 endpoints/msg_3_10_87.bin
-rw-r--r-- 1 fred fred 100 Apr 11 07:03 endpoints/msg_3_11_93.bin
-rw-r--r-- 1 fred fred 120 Apr 11 07:03 endpoints/msg_3_12_99.bin
```

Il est intéressant de constater que les transputeurs en bout de chaîne recoivent chacun un message long de leur transputeur intermédiaire "parent", et ce après avoir démarré, et avant de recevoir des messages courts à traiter. Cela semble indiquer l'envoi de code dynamique après le démarrage.

Puisqu'il n'est pas envisageable de casser la clé en boîte noire dans un temps raisonnable, on passe à la méthode suivante: l'analyse en boîte blanche.

Analyse en boîte blanche

On travaille sur la sortie d'IDA Pro. Il y environ 2 ko de code à analyser, cela reste un volume raisonnable. Notre but est de comprendre grossièrement le fonctionnement général, et finement les parties liées à l'algorithme de déchiffrement proprement dit, pour pouvoir les ré-écrire en C.

Transputeur 0

On commence par regarder le programme initial du transputeur 0, au tout début de "input.bin". Le code émet le message "Boot ok" sur le canal de sortie principale, puis route des paquets de données vers ses voisins.

Listing 20: routage des paquets

```
0018                init_neighbors:
0018 24 19                          ldlp    49h
001A 24 F2                          mint
001C 54                             ldnlp   4
001D 4C                             ldc     12
001E F7                             in
001F                ;
001F 24 79                          ldl     49h
0021 21 A5                          cj      first_input_msg_word_zero
0023 2C 4D                          ldc     (msg_buf - loc_27)
0025 21 FB                          ldpi
0027
0027                loc_27:
0027 24 F2                          mint
0029 54                             ldnlp   4
002A 24 79                          ldl     49h
002C F7                             in
002D                ;
002D 2C 43                          ldc     (msg_buf - loc_31)
002F 21 FB                          ldpi
0031
0031                loc_31:
0031 24 7A                          ldl     4Ah
0033 24 79                          ldl     49h
0035 FB                             out
0036 61 00                          j       init_neighbors
```

Les paquets de données commencent par un en-tête de 12 octets divisé en trois mots de 32 bits: une longueur de charge utile, un canal de destination, et un paramètre auxiliaire dont l'usage varie.

Le transputeur 0 relaie les paquets en les routant vers le canal indiqué dans l'en-tête de paquet, et ce jusqu'à atteindre un paquet dont la longueur de charge utile est 0. Après quoi il émet le message "Code Ok", lit la clé de déchiffrement, émet le message "Decrypt", écarte le nom de fichier, et entre en "régime permanent" de déchiffrement (à l'offset 0x78 de "input.bin").

Les paquets sont relayés sans leur en-tête de routage. Le transputeur destinataire ne reçoit que la charge utile qui suit l'en-tête de 12 octets.

Ensuite le transputeur 0 boucle sur sa routine de déchiffrement qui, à chaque itération, lit un octet du texte chiffré depuis l'entrée principale, dissémine un état interne de douze octets à ses voisins, lit leurs réponses, combinent celles-ci par un "ou exclusif", met à jour son état interne, et émet un octet déchiffré sur la sortie principale.

Listing 21: boucle de déchiffrement

```
0078                loc_78:
0078 11                             ldlp    1
0079 24 F2                          mint
007B 54                             ldnlp   4
007C 41                             ldc     1
007D F7                             in
007E                ;
007E 15                             ldlp    5
007F 24 F2                          mint
0081 51                             ldnlp   1
0082 4C                             ldc     12
```

```
0083 FB                          out
[...]
00CC 74                          ldl     4
00CD 81                          adc     1
00CE 25 FA                       dup
00D0 D4                          stl     4
00D1 CC                          eqc     0Ch
00D2 A3                          cj      loc_D6
00D3 80                          adc     0
00D4 40                          ldc     0
00D5 D4                          stl     4
00D6
00D6             loc_D6:
00D6 10                          ldlp    0
00D7 24 F2                       mint
00D9 41                          ldc     1
00DA FB                          out
00DB 66 0B                       j       loc_78
```

Le comportement du transputeur 0 nous fournit donc le format de "input.bin" suivant le programme initial du transputeur 0. On repère les en-têtes de paquets avec les informations de routage. Il y a 21 paquets avant le dernier paquet, vide, qui précède la clé.

Listing 22: en-têtes de paquets

```
00F9 71 00 00 00                 dd 71h
00FD 04 00 00 80                 dd 80000004h
0101 00 00 00 00                 dd 0
[...]
0176 71 00 00 00                 dd 71h
017A 08 00 00 80                 dd 80000008h
017E 00 00 00 00                 dd 0
[...]
[...]
08DD 90 00 00 00                 dd 90h
08E1 0C 00 00 80                 dd 8000000Ch
08E5 0C 00 00 00                 dd 0Ch
```

Transputeurs intermédiaires

Connaissant les informations de routage, nous en déduisons que le programme initial du transputeur 1 commence à l'offset 0x105. La longueur du code, 0x70, est suivie du programme à l'offset 0x106.

Tout comme le transputeur 0, le transputeur 1 commence par une phase de routage de paquets jusqu'à atteindre le paquet vide, après quoi il amorce sa boucle de déchiffrement.

Les transputeurs 2 et 3 fonctionnent comme le 1.

Transputeurs en bout de chaîne

Le transputeur 4 reçoit son programme initial depuis le transputeur 1. Il s'agit de la charge utile du paquet commençant à l'offset 0x270 de "input.bin":

Listing 23: premier paquet destiné au transputeur 4

```
0270 31 00 00 00                dd 31h
0274 04 00 00 80                dd 80000004h
0278 00 00 00 00                dd 0
027C 25 00 00 00                dd 25h
0280 04 00 00 80                dd 80000004h
0284 00 00 00 00                dd 0
0288 24                         dd 24h          ; longueur du programme
0289 60 BD                      ajw     -3h     ; programme initial
[...]
```

Le double en-tête indique que le paquet transite par le canal 1 du transputeur 0, puis par le canal 1 du transputeur 1.

Le programme initial du tranputeur 4 charge un morceau de code dynamique depuis son canal de lecture 0, puis appelle le point d'entrée du code dynamique avec l'instruction `gcall`. L'offset du point d'entrée dans le code dynamique est indiqué dans le paramètre auxiliaire de l'en-tête de paquet.

Le premier paquet de code dynamique apparaît dans "input.bin" à l'offset 0x4ad:

Listing 24: en-tête du code dynamique du transputeur 4

```
04AD 44 00 00 00                dd 44h
04B1 00 00 00 00                dd 0
04B5 0C 00 00 00                dd 0Ch
```

Le premier mot est la longueur de la charge utile, autrement dit la taille du code dynamique, le second mot est inutilisé, et le troisième (0xC) est l'offset du point d'entrée dans le code dynamique.

Les transputeurs 1, 2, et 3 partagent la même fonction de relais des programmes initiaux, des codes dynamiques, et des paquets de données.

Les transputeurs 4 à 12 recoivent chacun un code dynamique différent, mais fonctionnent selon la même logique: la réception d'un paquet de douze octets donne lieu à un calcul, à un éventuel changement d'état interne, et à l'émission d'un octet en sortie.

Les transputeurs 11 et 12 sont un peu spéciaux puisqu'ils sont liés l'un à autre et coopèrent. Le 12 lit une entrée du 11, et le 11 a besoin de la réponse du 12 pour finir son propre traitement. C'est équivalent à un appel de 12 par 11, et assez facile à "sérialiser".

Le découpage final du code dans le fichier "input.bin" est le suivant:

Offset (hex)	Transputeur	Nature du code
1	0	Programme initial
106	1	Programme initial
183	2	Programme initial
200	3	Programme initial
289	4	Programme initial
2C6	5	Programme initial
303	6	Programme initial
340	7	Programme initial
37D	8	Programme initial
3BA	9	Programme initial
3F7	10	Programme initial
434	11	Programme initial
471	12	Programme initial
4B9	4	Code dynamique
521	5	Code dynamique
589	6	Code dynamique
62D	7	Code dynamique
6A9	8	Code dynamique
75D	9	Code dynamique
7C9	10	Code dynamique
879	11	Code dynamique
901	12	Code dynamique

Algorithme de chiffrement

On constate que certains transputeurs conservent un état interne qui évolue au fil de leur exécution (par exemple le 4), alors que d'autres sont purement "fonctionnels" (par exemple le 7).

Tous les transputeurs reçoivent l'état du transputeur 0 à chaque tour de boucle, et le combinent éventuellement avec leur propre état interne pour renvoyer un octet pseudo-aléatoire. Les transputeurs intermédiaires 1, 2, et, 3, et le transputeur 0 combinent par un "ou exclusif" les octets produit par les transputeurs situés en bout de chaîne (4 à 12). Le transputeur 0 fait évoluer son état interne en fonction des réponses reçues de ses pairs.

Le transputeur 0 possède un état interne composé d'un tableau de douze octets et d'un compteur. Le tableau est initialisé avec les octets de la clé. Le compteur fonctionne modulo 12: il est incrémenté de 1 à chaque octet déchiffré, et remis à 0 lorsqu'il atteint la valeur 12. Le compteur sert d'index dans le tableau.

La traduction en C de la boucle de déchiffrement du transputeur 0 est la fonction `t0` du programme présenté en annexe E.

Le principe de fonctionnement est un chiffrement de flux (*Stream Cipher*). Le flux de clé (*keystream*) résulte de l'agrégation des flux de sorties de tous les transputeurs. Dans le transputeur 0 le flux de clé est xoré avec le texte chiffré pour produire le texte clair (et vice versa pour l'opération de chiffrement).

S'il l'on désigne par i le compteur/index du transputeur 0 et par k_i le $i^{\text{ème}}$ octet du

tableau, l'octet du flux de clé vaut $i + 2 * k_i$. À chaque itération, seul l'octet k_i du tableau est modifié. On comprend pourquoi chaque octet du premier bloc déchiffré ne dépend que d'un seul octet de la clé.

La multiplication par 2, équivalente à un décalage vers la gauche, explique pourquoi les bits de poids fort de la clé n'ont aucune influence sur le premier bloc déchiffré.

De manière plus générale, la multiplication par 2 implique que le bit de poids faible de chaque octet du texte chiffré ne dépend que du bit de poids faible de l'octet correspondant du texte clair et du bit de poids faible du compteur, qui alterne entre 0 et 1. Nous pouvons donc retrouver facilement tous les bits de poids faible du texte clair.

Les modifications apportées à k_i pendant les douze premiers tours de boucle dépendent du tableau k complet. À partir du treizième tour de boucle, la diffusion des bits de clé s'est donc opérée dans l'état interne du transputeur 0, et le second bloc déchiffré dépend de la clé de manière moins triviale que le premier.

Les boucles de déchiffrement des transputeurs sont modélisables par des fonctions C, qui maintiennent éventuellement un état sous forme de variables déclarées `static` ou globales. La traduction en C est un peu fastidieuse mais le code final est assez lisible. Voir l'annexe E.

Recherche de la clé

Nous mettons en œuvre une attaque par force brute sur les bits de poids fort des dix premiers octets de la clé, et sur la valeur des deux derniers octets.

À l'usage on constate que le déchiffrement de tout le texte pour chaque clé possible est très lent. Pour gagner du temps, il est intéressant de trouver des conditions d'arrêt qui permettent de ne déchiffrer que le début du texte dans la majorité des cas.

Pour éliminer une majorité des clés candidates, nous pouvons vérifier que le bit "randomised" de l'en-tête bzip2, qui est obsolète, vaut 0. Nous pouvons vérifier que les 4 bits de poids fort du champ "origPtr", encodé sur 24 bits, sont également à 0. En effet, "origPtr" représente un index dans une table d'au plus 900 ko (la table de Burrows-Wheeler), il est donc toujours inférieur à 2^{20}.

En raison de l'agencement des bits dans les flux bzip2, qui est *big endian*, le bit "randomised" et les 4 bits de poids fort de "origPtr" sont contigus, et apparaîssent à l'offset 14 du texte clair. On peut donc se contenter de déchiffrer 15 octets, dans un premier temps, et vérifier que ces 5 bits valent 0. Cette vérification élimine 97% des clés candidates.

Une fois ce premier filtrage réalisé, on peut déchiffrer 256 octets de texte, puis tenter de lancer une décompression sur le texte clair en utilisant la bibliothèque bzip2 elle-même. L'interface de la bibliothèque est bien conçue, en particulier de par la précision des codes d'erreur renvoyés.

La fonction `BZ2_bzBuffToBuffDecompress` de la bibliothèque bzip2 [20] permet de décompresser un flux, y compris partiel, en mémoire, et renvoie le code `BZ_DATA_ERROR` en cas d'erreur d'intégrité du flux compressé. Nous l'employons pour filtrer les clés candidates. Seules les clés produisant un début de texte clair décompressable sont retenues pour le déchiffrement complet du texte.

[20] `http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.html#bzbufftobuffdecompress`

Le programme de recherche des clés est présenté en annexe F.

On l'exécute, et en quelques minutes, on obtient une clé candidate qui s'avère être la bonne:

```
$ ./csim -bf input.bin
5e541b71567c64...
5e541b71567ce4...
5e541b7156fc64...
[...]
5ed49b7156fce4...
keys vetted: 26925766, first k test: 860882, fully decrypted candidates: 1
cands/5ed49b7156fce47de976dac5.tar.bz2
^C
$ cd cands
$ sha256sum 5ed49b7156fce47de976dac5.tar.bz2
9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321  5ed4...
```

Ceci conclue l'étape 5.

6 Images emboîtées

Le fichier "congratulations.tar.bz2" contient une image "congratulations.jpg":

```
$ bunzip2 congratulations.tar.bz2
$ tar tvf congratulations.tar
-rw-r--r-- test/test    252569 2015-03-23 05:34 congratulations.jpg
```

Figure 13: congratulations.jpg

En parcourant un dump hexadécimal du fichier JPEG, on tombe sur une chaîne de caractères qui nous est très familière grâce à l'étape précédente: un en-tête de fichier bzip2.

```
$ xxd congratulations.jpg | less
000d7d0: 425a 6839 3141 5926 5359 beec b4d2 0092  BZh91AY&SY......
000d7e0: 4bff ffff ffff ffff ffff ffff ffff ffff  K...............
```

On extrait le fichier bzip2 et on le décompresse. Il contient une archive tar, contenant elle-même un fichier "congratulations.png".

```
$ dd if=congratulations.jpg of=embedded.bz2 bs=1 skip=$((0xd7d0))
197321+0 records in
197321+0 records out
197321 bytes (197 kB) copied, 0.257046 s, 768 kB/s
$ bunzip2 embedded.bz2
$ xxd embedded | head -3
```

```
0000000: 636f 6e67 7261 7475 6c61 7469 6f6e 732e  congratulations.
0000010: 706e 6700 0000 0000 0000 0000 0000 0000  png.............
0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
$ tar tvf embedded
-rw-r--r-- test/test     197557 2015-03-23 05:34 congratulations.png
```

Figure 14: congratulations.png

Voilà une petite blague qui pourrait durer longtemps.

En se renseignant sur le format PNG [21], on découvre qu'il divise le fichier en une série de morceaux (*chunks*) possédant chacun une longueur, un type, des données, et un CRC.

Le type d'un chunk détermine la nature de son contenu. Par exemple le type "IDAT" indique des données de l'image proprement dite, alors que le type "PLTE" indique des données décrivant une palette de couleurs.

En observant les en-têtes de chunks à la main, on remarque de nombreux chunks d'un type particulier: "sTic". Tous sauf le dernier ont une longueur fixée à 0x1337 octets. Cela suggère d'extraire ces morceaux de l'image, ce qu'on réalise avec un petit parseur ad hoc G.

```
$ xxd congratulations.png | head -10
0000000: 8950 4e47 0d0a 1a0a 0000 000d 4948 4452  .PNG........IHDR
0000010: 0000 027c 0000 01da 0806 0000 009a 4094  ...|..........@.
0000020: 3800 0000 0662 4b47 4400 ff00 ff00 ffa0  8....bKGD.......
```

[21] http://en.wikipedia.org/wiki/Portable_Network_Graphics

```
0000030: bda7 9300 0000 0970 4859 7300 000d d700  .......pHYs.....
0000040: 000d d701 4228 9b78 0000 0007 7449 4d45  ....B(.x....tIME
0000050: 07df 021b 0d28 1303 5ffb 8300 0013 3773  .....(.._.....7s
0000060: 5469 6378 9c84 b67b 3813 eefb 383e ccda  Ticx...{8...8>..
$ xxd congratulations.png | grep sTic
00013a0: 1337 7354 6963 d7e5 f6b4 3a96 6c52 8fa6  .7sTic....:.lR..
00026e0: 3200 0013 3773 5469 6373 9904 db9f 81f6  2...7sTics......
0003a20: 75aa 3393 0000 1337 7354 6963 2930 59c6  u.3....7sTic)0Y.
[...]
$ cat congratulations.png | ./splitpng > /dev/null
$ ls -l sTic.*
-rw-r--r-- 1 fred fred 4919 Apr 13 16:04 sTic.004
-rw-r--r-- 1 fred fred 4919 Apr 13 16:04 sTic.005
[...]
-rw-r--r-- 1 fred fred 4919 Apr 13 16:04 sTic.030
-rw-r--r-- 1 fred fred   38 Apr 13 16:04 sTic.031
```

On tente de réassembler les morceaux et on lance la commande `file` pour voir si l'amalgame est un type de fichier connu.

```
$ cat sTic.* > sTic
$ file sTic
sTic: zlib compressed data
```

Il s'agit donc d'un flux zlib, que l'on peut décompresser avec Python en mode interactif:

```
$ python
Python 2.7.9 (default, Mar  1 2015, 12:57:24)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import zlib
>>> c = open('sTic', 'rb').read()
>>> d = zlib.decompress(c)
>>> open('u', 'wb').write(d)
>>> quit()
$ file u
u: bzip2 compressed data, block size = 900k
```

On récupère ainsi une nouvelle archive tar.bz2, contenant un fichier "congratulations.tiff".

```
$ tar tvf u.tar
-rw-r--r-- test/test    904520 2015-03-23 05:34 congratulations.tiff
```

Figure 15: congratulations.tiff

L'examen des composantes RGB (Red Green Blue) de l'image, à partir de l'offset 0x80, révèle une caractéristique intéressante: de nombreux bits de poids faible des composantes R et G sont à 1 dans la zone d'apparence uniformément noire qui débute l'image.

```
$ dd if=congratulations.tiff bs=1 skip=128 | xxd -c 3 -g 1 | less
0000000: 00 01 00  ...
0000003: 00 00 00  ...
0000006: 00 00 00  ...
0000009: 01 00 00  ...
000000c: 00 01 00  ...
000000f: 00 01 00  ...
0000012: 01 00 00  ...
0000015: 01 00 00  ...
0000018: 00 01 00  ...
000001b: 01 00 00  ...
000001e: 01 00 00  ...
0000021: 00 00 00  ...
0000024: 00 00 00  ...
0000027: 01 01 00  ...
000002a: 01 00 00  ...
000002d: 00 01 00  ...
0000030: 00 00 00  ...
```

On retrouve la même déviance des bits de poids faibles dans les zones d'apparence uniformément blanche, et on suppose qu'elle pourrait s'étendre à toute l'image. Un signal

semble être caché dans les bits de poids faible des composantes R et G.

```
000cd95: ff ff ff  ...
000cd98: ff ff ff  ...
000cd9b: fe ff ff  ...
000cd9e: ff ff ff  ...
000cda1: ff ff ff  ...
000cda4: ff ff ff  ...
000cda7: fe fe ff  ...
000cdaa: fe fe ff  ...
000cdad: ff fe ff  ...
000cdb0: ff fe ff  ...
000cdb3: ff fe ff  ...
000cdb6: fe fe ff  ...
```

Essayons alors de sélectionner les bits de poids faible de ces deux composantes et de former un flux d'octets à partir d'eux.

Listing 25: rg.c

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

int main()
{
    struct { char r,g,b; } rgb[4];

    while (!feof(stdin)) {
        fread(rgb, 3, 4, stdin);
        putchar(((rgb[0].r & 1) << 7) +
                ((rgb[0].g & 1) << 6) +
                ((rgb[1].r & 1) << 5) +
                ((rgb[1].g & 1) << 4) +
                ((rgb[2].r & 1) << 3) +
                ((rgb[2].g & 1) << 2) +
                ((rgb[3].r & 1) << 1) +
                ((rgb[3].g & 1) << 0));
    }
}
```

```
$ cc -o rg rg.c
$ dd if=congratulations.tiff bs=1 skip=128 | ./rg | xxd | head -1
0000000: 425a 6839 3141 5926 5359 4202 25c8 0019  BZh91AY&SYB.%...
$ dd if=congratulations.tiff bs=1 skip=128 | ./rg > foo.tbz2
$ bunzip2 foo.tbz2
bunzip2: foo.tbz2: trailing garbage after EOF ignored
$ tar tvf foo.tar
-rw-r--r-- test/test     28755 2015-03-23 05:34 congratulations.gif
```

On reconnaît un nouvel en-tête de flux bzip2. Une fois décompressé, on trouve une archive tar, contenant un fichier "congratulations.gif".

Figure 16: congratulations.gif

Le format GIF [22] contient entre autres éléments une table des couleurs décrivant jusqu'à 256 couleurs présentes dans l'image. Chaque couleur dans la table est décrite par ses composantes RGB. Les valeurs des pixels de l'image se réfèrent ensuite à des index dans la table des couleurs.

Si l'on observe attentivement la table des couleurs de notre image, on voit des trous: certaines entrées dans la table sont mises à 0 (couleur noire) alors qu'elles sont situées n'importe où dans la table, entre des couleurs qui semblent décrire des teintes valides.

```
$ dd if=congratulations.gif bs=1 skip=22 | xxd -c 3 -g 1 | head -256 | less
[...]
0000051: 1e 20 1d  . .
0000054: 00 00 00  ...
0000057: 4b 17 16  K..
[...]
0000081: 2c 2a 0a  ,*.
0000084: 00 00 00  ...
0000087: 82 12 15  ...
[...]
00000bd: 7f 20 1e  . .
00000c0: 00 00 00  ...
00000c3: 00 00 00  ...
00000c6: b6 14 11  ...
```

[22] http://en.wikipedia.org/wiki/GIF

```
[...]
00000e7: 44 41 35  DA5
00000ea: 00 00 00  ...
00000ed: 7f 34 19  .4.
00000f0: 00 00 00  ...
00000f3: 00 00 00  ...
00000f6: ac 2d 15  .-.
```

Cela donne envie de faire apparaître ces couleurs mystérieuses. On remplace donc leurs entrées par FFFFFF (couleur blanche), et on visualise le résultat:

Figure 17: solution.gif

L'adresse courriel convoitée apparaît alors comme par magie.

7 Conclusion

Je remercie le concepteur du challenge pour ces épreuves nombreuses et variées. J'ai trouvé amusante l'épreuve de la souris où il faut reconstruire un dessin d'après une trace USB, et j'ai tout particulièrement apprecié la partie ST20, qui était l'occasion de se familiariser avec une architecture originale.

Avec le recul, je réalise qu'il existait d'autres méthodes de résolution plus directes pour certaines des épreuves, et mettant en jeu moins de programmation.

L'étape 7, la rédaction en LaTeX, était presque aussi difficile que le ST20. J'en garde un goût amer.

Enfin, je remercie mes collègues à l'ANSSI, dont l'enthousiasme partagé pour ce genre de challenge fait régner une ambiance de saine émulation.

A Interpréteur de capture souris

Listing 26: drawmouse.ml

```
open Graphics
open Printf
open Scanf

let sint8 x = if x land 0x80 <> 0 then -(256 - x) else x

let main() =
  open_graph " 1000x800";
  moveto 0 700;
  let rec draw_evt() =
    bscanf Scanning.stdin "%x %x %x\n" (fun btns dx dy ->
      let dx = sint8 dx in
      let dy = sint8 dy in
      let x = current_x() + dx in
      let y = current_y() - dy in
      if btns = 0 then
        moveto x y
      else if btns = 1 then
        lineto x y
      else
        assert false
    );
    draw_evt()
  in
  try
    draw_evt()
  with End_of_file -> ()
;;

let () = main()
```

B Déchiffrement en Serpent-CBC-CTS

Listing 27: decrypt_serpent.py

```
from blake import BLAKE
from CryptoPlus.Cipher import python_Serpent

X = "The quick brown fox jumps over the lobster dog"
IV = b'SSTIC2015-Stage3'

blake = BLAKE(256)
blake.update(X)
key = blake.digest()

encrypted = open('encrypted','rb').read()

total = len(encrypted)
blksz = 16

serpent = python_Serpent.new(key, python_Serpent.MODE_CBC, IV)

with open('decrypted', 'wb') as d:

    even = (total / blksz) * blksz

    tof = encrypted[:even-blksz]
    eof = encrypted[even-blksz:]

    d.write(serpent.decrypt(tof))

    if total == even:
        assert(False) # don't bother...
    else:
        # implement CTS (CipherText Stealing) mode
        cnm1 = eof[:blksz]
        sp = python_Serpent.new(key)
        dn = sp.decrypt(cnm1)
        partial = total - even
        cn = eof[blksz:] + dn[partial:blksz]
        assert(len(cn) == blksz)
        ptxt = serpent.decrypt(cn + cnm1)
        d.write(ptxt[:blksz+partial])
```

C Recherche du User Agent

Listing 28: findua.py

```
from Crypto.Cipher import AES
from Crypto.Hash import SHA
from itertools import chain

windows = [
    'Windows NT 5.0',
    'Windows NT 5.1',
    'Windows NT 5.2',
    'Windows NT 6.0',
    'Windows NT 6.1',
    'Windows NT 6.2',
    'Windows NT 6.3',
    'Windows NT 10.0',
]

def gen_windows():
    for w in windows:
        yield([w])
        yield([w, 'WOW64'])
        yield([w, 'Win64', 'x64'])

linux = [
    'Linux x86_64',
    'Linux i686',
    'Linux i686 on x86_64',
    'Linux i686 (x86_64)',
]

bsd = [
    'FreeBSD',
    'FreeBSD i386',
    'FreeBSD amd64',
    'NetBSD',
    'NetBSD i386',
    'NetBSD amd64',
    'OpenBSD',
    'OpenBSD i386',
    'OpenBSD amd64',
]

def gen_unix():
    for l in linux:
        yield(['X11', l])
        yield(['X11; U', l])
        yield(['X11; debian', l])
    for b in bsd:
        yield(['X11', b])
        yield(['X11; U', b])

android = [
    ['Android; Mobile'],
    ['Android; Tablet'],
]
```

```
def gen_mac():
    for mac in ['Macintosh', 'Macintosh; U', 'Macintosh; I']:
        for cpu in ['Intel', 'PPC']:
            yield([mac, cpu + ' Mac OS X'])
            yield([mac, cpu + ' Mac OS X Mach-O'])
            for ver in xrange(11):
                yield([mac, cpu + ' Mac OS X 10.%d' % ver])
                for subver in xrange(20):
                    yield([mac, cpu + ' Mac OS X 10_%d_%d' %
                                                    (ver, subver)])
                    yield([mac, cpu + ' Mac OS X 10.%d.%d' %
                                                    (ver, subver)])

ff_revs = [
    '40.0',
    '39.0',
    '38.0',
    '37.0',
    '36.0',
    '35.0',
    '34.0',
    '33.0', '33.1',
    '32.0',
    '31.0', '31.1', '31.2', '31.3', '31.4', '31.5', '31.6',
    '30.0',
    '29.0',
    '28.0',
    '27.0',
    '26.0',
    '25.0',
    '24.0', '24.1', '24.2', '24.3', '24.4', '24.5', '24.6', '24.7',
                                                        '24.8',
    '23.0',
    '22.0',
    '21.0',
    '20.0',
]

def gen_ff_revs():
    yield([])
    for r in ff_revs:
        yield(['rv:' + r])

def gen_ff_ua(locale):
    for os in chain(gen_unix(), gen_windows(), android, gen_mac()):
        for rev in gen_ff_revs():
            if locale is None:
                yield(os + rev)
            else:
                yield(os + [locale] + rev)

def join_ua(items):
    return 'Mozilla/5.0 (' + '; '.join(items) + ') '

enc = open('encrypted','rb').read()

hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c".decode('hex')

print 'hash length:', len(hash) * 8, 'bits'
```

```
seen = dict()

def try_agent(ag):
    print ag
    try:
        lparen = ag.index('(')
        rparen = ag.index(')')
    except:
        pass

    iv = ag[lparen+1:][:16]
    key = ag[rparen-16:][:16]

    if len(iv) < 16 or len(key) < 16:
        return

    if seen.get((iv, key)) is not None:
        return

    seen[(iv, key)] = True

    cipher = AES.new(key, mode=AES.MODE_CBC, IV=iv)
    dec = cipher.decrypt(enc[:32])
    print dec[:32].encode('hex')

    if dec[:4] == 'PK\x03\x04':
        dec += cipher.decrypt(enc[32:])
        padding = ord(dec[-1])
        dec = dec[:-padding]
        sha = SHA.new()
        sha.update(dec)
        if sha.digest() == hash:
            open('decrypted', 'wb').write(dec)
            print '===\ngot it!\n'
            raw_input()

locales = [ None, 'fr', 'fr-FR', 'en-US' ]

def t(items): try_agent(join_ua(items))

for loc in locales:
    for items in gen_ff_ua(loc):
        t(items)
```

D Émulateur ST20

Listing 29: st20.ml

```
type addr = int32
type msg_sz = int32
type msg = string

type channel = {
  src : int;
  dst : int;
  mutable msg : (Buffer.t * int) option
}

type core_state =
    Boot
  | Run
  | In_ of channel * addr * msg_sz
  | Out_ of channel

type core = {
  id : int;
  a : int32 option; (* register stack *)
  b : int32 option;
  c : int32 option;
  iptr : int32; (* instruction pointer *)
  wptr : int32; (* workspace pointer *)
  state : core_state;
  links : (channel * channel) option array; (* in chan/out chan *)
  mem : string;
  mutable cout : out_channel;
}

let membase = 0x8000_0000l
let memstart = 0x8000_0140l
let rebase addr = Int32.(sub addr membase |> to_int)

let new_core id = {
  id;
  a = None;
  b = None;
  c = None;
  iptr = memstart;
  wptr = 0x8000_1000l;
  state = Boot;
  links = [| None; None; None; None |];
  mem = String.make 0x10000 'M';
  cout = stdout;
}

(*****************************************************************)

let quiet = ref false

let reg_str = function None -> "????????"
                     | Some x -> Printf.sprintf "%08lx" x

let print_core_state core =
```

```
  Printf.fprintf core.cout "A=%s B=%s C=%s Wptr=%08lx\n"
                                   (reg_str core.a)
                                   (reg_str core.b)
                                   (reg_str core.c)
                                    core.wptr;
  Utils.dump core.cout core.mem (rebase core.wptr) 32
;;

(*****************************************************************)

let buf_of_str s =
  let buf = Buffer.create (String.length s) in
  Buffer.add_string buf s;
  buf
;;

let dump_msg msg =
  String.iter (fun c -> Char.code c |> Printf.printf "%02x ") msg;
  if String.length msg = 1 then
    print_char (Utils.printable msg.[0])
  else
    Printf.printf "[length = %x]" (String.length msg)
;;

let new_channel ?init_data src dst =
  match init_data with
  | None -> { msg = None; src; dst }
  | Some data -> { msg = Some (buf_of_str data, 0); src; dst }
;;

let read_chan_msg chan sz =
  match chan.msg with
  | None -> None
  | Some (buf, i) ->
      assert (Buffer.length buf - i >= sz);
      let msg = Buffer.sub buf i sz in
      if not !quiet then begin
        Printf.printf "Read  %d->%d: " chan.src chan.dst;
        dump_msg msg;
        print_newline()
      end;
      chan.msg <- Some (buf, i+sz);
      Some msg
;;

let read_chan_byte chan =
  match chan.msg with
  | None -> None
  | Some (buf, i) ->
      if Buffer.length buf <= i then None else
      let by = Char.code (Buffer.nth buf i) in
      if not !quiet then
        Printf.printf "%d -> %d: read byte %02x\n%!"
                                        chan.src chan.dst by;
      chan.msg <- Some (buf, i+1);
      Some by
;;

let read_chan chan len =
```

```
  match chan.msg with
  | None -> None
  | Some (buf, i) ->
      let j = i + len in
      assert (j >= 0);
      if Buffer.length buf < j then None else
      let data = Buffer.sub buf i len in
      chan.msg <- Some (buf, i + len);
      Some data
;;

let write_chan_msg chan msg =
  if not !quiet then begin
    Printf.printf "Write %d->%d: " chan.src chan.dst;
    dump_msg msg;
    print_newline()
  end;
  match chan.msg with
  | None -> chan.msg <- Some (buf_of_str msg, 0)
  | Some (buf, i) ->
      Buffer.add_string buf msg
;;

let chan_len chan =
  match chan.msg with
  | None -> 0
  | Some (buf, i) -> Buffer.length buf - i
;;

(*****************************************************************)

type insn =
    Ajw of int32
  | Ldc of int32
  | Adc of int32
  | Stl of int32
  | Mint
  | Ldnlp of int32
  | Gajw
  | Ldpi
  | In
  | Out
  | Ldlp of int32
  | Ldl of int32
  | Cj of int32
  | J of int32
  | Lb
  | Sb
  | Xor
  | Bsub
  | Ssub
  | Dup
  | Eqc of int32
  | Gcall
  | Call of int32
  | Ret
  | And
  | Gt
  | Shr
```

```
  | Shl
  | Prod
  | Wsub
  | Rem
  | Unknown


(*****************************************************************)

let str_of_insn = function
  | Ajw x -> Printf.sprintf "ajw %lx" x
  | Ldc x -> Printf.sprintf "ldc %lx" x
  | Adc x -> Printf.sprintf "adc %lx" x
  | Stl x -> Printf.sprintf "stl %lx" x
  | Ldnlp x -> Printf.sprintf "ldnlp %lx" x
  | Ldlp x -> Printf.sprintf "ldlp %lx" x
  | Ldl x -> Printf.sprintf "ldl %lx" x
  | Eqc x -> Printf.sprintf "eqc %lx" x
  | Cj x -> Printf.sprintf "cj $+%lx" x
  | J x -> Printf.sprintf "j $+%lx" x
  | Call x -> Printf.sprintf "call $+%lx" x
  | Mint -> "mint"
  | Gajw -> "gajw"
  | Ldpi -> "ldpi"
  | In -> "in"
  | Out -> "out"
  | Lb -> "lb"
  | Sb -> "sb"
  | Xor -> "xor"
  | Bsub -> "bsub"
  | Ssub -> "ssub"
  | Dup -> "dup"
  | Gcall -> "gcall"
  | Ret -> "ret"
  | And -> "and"
  | Gt -> "gt"
  | Shr -> "shr"
  | Shl -> "shl"
  | Prod -> "prod"
  | Wsub -> "wsub"
  | Rem -> "rem"
  | Unknown -> "?"
;;

(*****************************************************************)

external st20_decode : string -> int32 -> (int * int32 * int) =
                                                    "st20_decode"

let disas mem addr =
  let opcode, operand, ilen = st20_decode mem addr in
  let ilen = ref ilen in
  let insn =
    match opcode with
    | 0 -> J operand
    | 1 -> Ldlp operand
    | 4 -> Ldc operand
    | 5 -> Ldnlp operand
    | 7 -> Ldl operand
```

```
    | 8 -> Adc operand
    | 9 -> Call operand
    | 0xa -> Cj operand
    | 0xb -> Ajw operand
    | 0xc -> Eqc operand
    | 0xd -> Stl operand
    | 0x11 -> Lb
    | 0x12 -> Bsub
    | 0x16 -> Gcall
    | 0x17 -> In
    | 0x18 -> Prod
    | 0x19 -> Gt
    | 0x1a -> Wsub
    | 0x1b -> Out
    | 0x2b -> Ldpi
    | 0x2f -> Rem
    | 0x43 -> Xor
    | 0x4b -> Sb
    | 0x4c -> Gajw
    | 0x50 -> Shr
    | 0x51 -> Shl
    | 0x52 -> Mint
    | 0x56 -> And
    | 0x6a -> Dup
    | 0xd1 -> Ssub
    | 0x30 -> Ret
    | _ ->
      Printf.printf "unknown opcode %x\n" opcode;
      ilen := 1;
      Unknown
  in
  insn, !ilen
;;

(*****************************************************************)

let lsbyte x = (Int32.to_int x) land 0xff

(*****************************************************************)

let push core x =
  { core with a = Some x; b = core.a; c = core.b }
;;

let push_opt core x =
  { core with a = x; b = core.a; c = core.b }
;;

let a_ core =
  match core.a with
  | Some x -> x
  | None -> assert false
;;

let b_ core =
  match core.b with
  | Some x -> x
  | None -> assert false
;;
```

```
let c_ core =
  match core.c with
  | Some x -> x
  | None -> assert false
;;

let pop core =
  { core with a = core.b; b = core.c; c = None }, a_ core
;;

let local_addr core local = Int32.(add core.wptr (mul local 4l))

let mem_read32 core addr =
  Serialze.get_uint32 core.mem (rebase addr)
;;

let mem_read8 core addr =
  Char.code core.mem.[rebase addr]
;;

let mem_write8 core addr by =
  core.mem.[rebase addr] <- char_of_int by
;;

let mem_write32 core addr x =
  let ofs = rebase addr in
  Serialze.set_uint32 core.mem ofs x
;;

let read_local core local =
  mem_read32 core (local_addr core local)
;;

let write_local core local x =
  mem_write32 core (local_addr core local) x
;;

(*****************************************************************)

let chan_id_of_addr = function
  | 0x8000_0000l
  | 0x8000_0010l -> 0
  | 0x8000_0004l
  | 0x8000_0014l -> 1
  | 0x8000_0008l
  | 0x8000_0018l -> 2
  | 0x8000_000cl
  | 0x8000_001cl -> 3
  | _ -> failwith "invalid channel address"
;;

(*****************************************************************)

let retire_in core =
  { core with iptr = Int32.succ core.iptr; state = Run }
;;

let retire_out core =
```

```
  { core with iptr = Int32.succ core.iptr; state = Run }
;;

(*****************************************************************)

let save_bootstraps = ref false

let core_step core =
  let result = ref core in
  begin match core.state with
  | Boot -> (* attempt to dload bootstrap code from link 0 *)
      let in0 =
        match core.links.(0) with
        | None -> assert false
        | Some (i, o) -> i
      in
      begin match read_chan_byte in0 with
      | None ->
          if not !quiet then
            Printf.fprintf core.cout
              "%d: waiting for bootstrap code\n%!" core.id
      | Some len ->
        begin match read_chan in0 len with
        | None -> Printf.eprintf "bootstrap code error\n%!"
        | Some bytes ->
            if not !quiet then
              Printf.fprintf core.cout
               "got %d bytes of bootstrap code; starting\n%!"
                                          (String.length bytes);
            String.blit bytes 0 core.mem (rebase memstart)
                                        (String.length bytes);
            if !save_bootstraps then begin
              let name = Printf.sprintf "bootstrap.%d" core.id in
              let c = open_out_bin name in
              output_string c bytes;
              close_out c
            end;
            result := { core with state = Run }
        end
      end
  | Run -> (* emulate one instruction *)
      let insn, ilen = disas core.mem core.iptr in
      if not !quiet then begin
        print_core_state core;
        Printf.fprintf core.cout "%d: %lx %s\n%!"
                          core.id core.iptr (str_of_insn insn);
      end;
      let iptr = Int32.add core.iptr (Int32.of_int ilen) in
      begin match insn with
      | Ajw x ->
          let wptr = Int32.(add core.wptr (mul x 4l)) in
          result := { core with wptr; iptr }
      | Ldc x ->
          let core = push core x in
          result := { core with iptr }
      | Ldl local ->
          let x = read_local core local in
          let core = push core x in
          result := { core with iptr }
```

```
      | Stl local ->
        let core, x = pop core in
        write_local core local x;
        result := { core with iptr }
      | Mint ->
        let core = push core 0x8000_0000l in
        result := { core with iptr }
      | Ldnlp x ->
        let a = Some Int32.(add (a_ core) (mul x 4l)) in
        result := { core with a; iptr }
      | Ldlp local ->
        let core = push core (local_addr core local) in
        result := { core with iptr }
      | Gajw ->
        let core, wptr = pop core in
        result := { core with wptr; iptr }
      | Ldpi ->
        let a = Some (Int32.add (a_ core) iptr) in
        result := { core with a; iptr }
      | Lb ->
        let core, addr = pop core in
        let valu = mem_read8 core addr in
        let core = push core (Int32.of_int valu) in
        result := { core with iptr }
      | Sb ->
        let core, addr = pop core in
        let core, valu = pop core in
        mem_write8 core addr (lsbyte valu);
        result := { core with iptr }
      | Dup ->
        let core = push core (a_ core) in
        result := { core with iptr }

      (* control flow *)

      | J x ->
          let iptr = Int32.add iptr x in
          result := { core with iptr; a = None
                                ; b = None
                                ; c = None }
      | Cj x ->
        if a_ core = 0l then
          let iptr = Int32.add iptr x in
          result := { core with iptr }
        else
          let core, _ = pop core in
          result := { core with iptr }
      | Gcall ->
          let core, dst = pop core in
          result := { core with a = Some iptr; iptr = dst }
      | Call x ->
          let wptr = local_addr core (-4l) in
          let core = { core with wptr } in
          write_local core 0l iptr;
          write_local core 1l (a_ core);
          write_local core 2l (b_ core);
          write_local core 3l (c_ core);
          let iptr = Int32.add iptr x in
          result := { core with a = None
```

```
                          ; b = None
                          ; c = None; iptr; wptr }
  | Ret ->
      let iptr = read_local core 0l in
      let wptr = local_addr core 4l in
      result := { core with wptr; iptr }

  (* ALU *)

  | Adc 0l ->
      result := { core with iptr }
  | Adc x ->
      let a = Some (Int32.add (a_ core) x) in
      result := { core with a; iptr }
  | Bsub ->
      let core, a = pop core in
      let core, b = pop core in
      let core = push core (Int32.add a b) in
      result := { core with iptr }
  | Ssub ->
      let core, a = pop core in
      let core, b = pop core in
      let core = push core (Int32.(add a (mul b 2l))) in
      result := { core with iptr }
  | Wsub ->
      let core, a = pop core in
      let core, b = pop core in
      let core = push core (Int32.(add a (mul b 4l))) in
      result := { core with iptr }
  | Prod ->
      let core, a = pop core in
      let core, b = pop core in
      let core = push core (Int32.mul a b) in
      result := { core with iptr }
  | Gt ->
      let core, a = pop core in
      let core, b = pop core in
      let gt = if b > a then 1l else 0l in
      let core = push core gt in
      result := { core with iptr }
  | Eqc x ->
      let core, a = pop core in
      let eq = if a = x then 1l else 0l in
      let core = push core eq in
      result := { core with iptr }
  | And ->
      let core, a = pop core in
      let core, b = pop core in
      let core = push core (Int32.logand a b) in
      result := { core with iptr }
  | Xor ->
      let core, a = pop core in
      let core, b = pop core in
      let core = push core (Int32.logxor a b) in
      result := { core with iptr }
  | Rem ->
      let core, a = pop core in
      let core, b = pop core in
      let v =
```

```
            if a = 0l then None else
            Some (Int32.rem b a)
          in
          let core = push_opt core v in
          result := { core with iptr }
      | Shl ->
        let core, n = pop core in
        let core, valu = pop core in
        let shftd =
          if n < 32l then
            Some (Int32.(shift_left valu (to_int n)))
          else
            None
        in
        let core = push_opt core shftd in
        result := { core with iptr }
      | Shr ->
        let core, n = pop core in
        let core, valu = pop core in
        let shftd =
          if n < 32l then
            Some (Int32.(shift_right valu (to_int n)))
          else
            None
        in
        let core = push_opt core shftd in
        result := { core with iptr }

      (* Comm *)

      | In ->
        let core, msg_sz = pop core in
        let core, chan_addr = pop core in
        let i = chan_id_of_addr chan_addr in
        let chan =
          match core.links.(i) with
          | None -> failwith "input from missing channel"
          | Some (c, _) -> c
        in
        let core, addr = pop core in
        let state = In_ (chan, addr, msg_sz) in
        result := { core with state }
      | Out ->
        let core, msg_sz = pop core in
        let core, chan_addr = pop core in
        let i = chan_id_of_addr chan_addr in
        let chan =
          match core.links.(i) with
          | None -> failwith "output to missing channel"
          | Some (_, c) -> c
        in
        let core, addr = pop core in
        let msg = String.sub core.mem (rebase addr)
                                      (Int32.to_int msg_sz) in
        write_chan_msg chan msg;
        result := { core with state = Out_ chan }
      | Unknown -> failwith "unhandled insn"
      end
  | In_ _
```

```
  | Out_ _ -> ()
  end;
  !result
;;

(*****************************************************************)

type grid = { transputers : core array; sink : channel }

let new_grid ?key feedfile =

  let transputers = Array.init 13 new_core in

  let link_cores a i b j =
    let cab = new_channel a b in
    let cba = new_channel b a in
    assert(transputers.(a).links.(i) = None);
    assert(transputers.(b).links.(j) = None);
    transputers.(a).links.(i) <- Some (cba, cab);
    transputers.(b).links.(j) <- Some (cab, cba)
  in

  let feed feedfile =
    let cin = open_in_bin feedfile in
    let open Unix in
    let stat = fstat (descr_of_in_channel cin) in
    let init_data = String.make stat.st_size 'X' in
    really_input cin init_data 0 stat.st_size;
    close_in cin;
    begin match key with
    | None -> ()
    | Some key ->
      assert (String.sub init_data 0x985 4 = "KEY:");
      String.blit key 0 init_data 0x989 12
    end;
    new_channel ~init_data (-1) 0
  in

  let sink = new_channel 0 (-1) in
  transputers.(0).links.(0) <- Some (feed feedfile, sink);

  link_cores 0 1 1 0;
  link_cores 0 2 2 0;
  link_cores 0 3 3 0;
  link_cores 1 1 4 0;
  link_cores 1 2 5 0;
  link_cores 1 3 6 0;
  link_cores 2 1 7 0;
  link_cores 2 2 8 0;
  link_cores 2 3 9 0;
  link_cores 3 1 10 0;
  link_cores 3 2 11 0;
  link_cores 3 3 12 0;
  link_cores 11 1 12 1;

  { transputers; sink }
;;

(*****************************************************************)
```

```
let bin_sink = ref None

let pump_sink grid =
  let n = chan_len grid.sink in
  if n > 0 then begin
    grid.transputers.(0) <- retire_out grid.transputers.(0);
    match read_chan grid.sink n with
    | None -> assert false
    | Some bytes ->
      if not !quiet then begin
        print_endline "got bytes from sink:";
        print_endline (String.map Utils.printable bytes)
      end;
      match !bin_sink with
      | None -> ()
      | Some cout -> output_string cout bytes; flush cout
  end
;;

let save_messages = ref false
let msg_count = ref 0

let sync_channels grid =

  (* retire all pending out's first *)

  Array.iteri (fun i t ->
    match t.state with
    | Out_ chan when chan.dst <> -1 ->
        begin match grid.transputers.(chan.dst).state with
        | In_ _ ->
            grid.transputers.(i) <- retire_out t
        | _ -> ()
        end
    | _ -> ()
  ) grid.transputers;

  (* then retire all pending in's *)

  Array.iteri (fun i t ->
    match t.state with
    | In_ (chan, addr, msg_sz) ->
        let n = Int32.to_int msg_sz in
        if chan_len chan >= n then begin
          grid.transputers.(i) <- retire_in t;
          match read_chan_msg chan n with
          | Some msg ->
              String.blit msg 0 t.mem (rebase addr) n;
              incr msg_count;
              if !save_messages then begin
                let name = Printf.sprintf
                    "msgs/msg_%d_%d_%d.bin" chan.src chan.dst
                                                    !msg_count in
                let c = open_out_bin name in
                output_string c msg;
                close_out c
              end
          | None -> assert false
```

```
        end
    | _ -> ()
  ) grid.transputers;
;;

let system_step grid =
  Array.iteri (fun i t ->
    grid.transputers.(i) <- core_step t) grid.transputers;
  sync_channels grid;
  pump_sink grid
;;

(*****************************************************************)

let usage() =
  Printf.printf "Usage: st20 <input.bin> [out1 out2 ...]\n";
  exit 0
;;

let echo_on() =
  let open Unix in
  if isatty stdout then
  let term_io = tcgetattr stdout in
  tcsetattr stdout TCSANOW { term_io with c_echo = true }
;;

let echo_off() =
  let open Unix in
  if isatty stdout then
  let term_io = tcgetattr stdout in
  tcsetattr stdout TCSANOW { term_io with c_echo = false }
;;

(*****************************************************************)

let decode_hex s =
  assert (String.length s mod 2 = 0);
  let n = String.length s / 2 in
  let buf = String.make n 'X' in
  for i = 0 to n-1 do
    Scanf.sscanf (String.sub s (2*i) 2) "%02x"
                 (fun by ->  buf.[i] <- Char.chr by)
  done;
  buf
;;

let main() =
  if Array.length Sys.argv < 2 then usage() else

  let key = ref None in
  let nowait = ref false in
  let arg_no = ref 1 in
  let rec parse_opts() =
    match Sys.argv.(!arg_no) with
    | "-q" ->
        quiet := true;
        incr arg_no;
        parse_opts()
    | "-key" ->
```

```
        key := Some (decode_hex Sys.argv.(!arg_no + 1));
        incr arg_no;
        incr arg_no;
        parse_opts()
    | "-sink" ->
        bin_sink := Some (open_out_bin Sys.argv.(!arg_no + 1));
        incr arg_no;
        incr arg_no;
        parse_opts()
    | "-go" ->
        nowait := true;
        incr arg_no;
        parse_opts()
    | "-save" ->
        save_bootstraps := true;
        incr arg_no;
        parse_opts()
    | "-msgs" ->
        save_messages := true;
        incr arg_no;
        parse_opts()
    | _ -> ()
  in
  parse_opts();
  let arg_no = !arg_no in
  let grid = new_grid ?key:!key Sys.argv.(arg_no) in
  for i = 0 to 12 do
    grid.transputers.(i).cout <-
      try open_out Sys.argv.(arg_no+1+i)
      with Invalid_argument _ -> stdout
  done;
  (*echo_off();*)
  let rec emu() =
    system_step grid;
    if !nowait then emu() else
    let c = input_char stdin in
    if c <> 'q' then emu()
  in
  emu();
  (*echo_on()*)
;;

(*****************************************************************)

let disas_test() =
  let cin = open_in "input.bin" in
  let mem = String.make 4096 'X' in
  really_input cin mem 0 4096;
  close_in cin;
  try
    let rec loop addr =
      let insn, ilen = disas mem addr in
      Printf.printf "%lx %s\n" addr (str_of_insn insn);
      loop (Int32.(add addr (of_int ilen)))
    in
    loop 0x8000_0000l
  with Invalid_argument err ->
    Printf.eprintf "%s" err
;;
```

```
let () = if not !Sys.interactive then main()
```

E Traduction en langage C du code ST20

Listing 30: csim.c - partie 1

```
typedef uint8_t by, *pby;
typedef int32_t wd;

int verbose = 0;

struct state {
    struct { wd y; by state[12]; } t0;
    struct { by sum; } t4;
    struct { by x; } t5;
    struct { wd state, init; } t6;
    struct { wd y, state[12]; } t8;
    struct { wd y, state[12]; } t10;
    struct { by state[12]; } t12;
} gs;

void reset_global_state()
{
    memset(&gs, 0, sizeof(gs));
}

by t4(by msg[12])
{
    int i;

    for (i = 0; i < 12; i++) {
        gs.t4.sum += msg[i];
    }

    return gs.t4.sum;
}

by t5(by msg[12])
{
    int i;

    for (i = 0; i < 12; i++) {
        gs.t5.x ^= msg[i];
    }

    return gs.t5.x;
}

by t6(by msg[12])
{
    int i;

    if (!gs.t6.init) {
        for (i = 0; i < 12; i++) {
            gs.t6.state += msg[i];
        }
        gs.t6.init = 1;
    }

    gs.t6.state = (gs.t6.state << 1)
```

```
                ^ ((gs.t6.state & 0x8000) >> 15)
                ^ ((gs.t6.state & 0x4000) >> 14);

    return (by) gs.t6.state;
}

by t1(by msg[12])
{
    by r4 = t4(msg);
    by r5 = t5(msg);
    by r6 = t6(msg);
    by x = r4 ^ r5 ^ r6;

    if (verbose) {
        printf("Write 4 -> 1: %02x\n", r4);
        printf("Write 5 -> 1: %02x\n", r5);
        printf("Write 6 -> 1: %02x\n", r6);
        printf("Write 1 -> 0: %02x\n", x);
    }

    return x;
}

by t7(by msg[12])
{
    wd v1 = 0, v2 = 0;
    int i;

    for (i = 0; i < 6; i++) {
        v1 += msg[i];
        v2 += msg[i+6];
    }

    return (by) (v1 ^ v2);
}

by t8(by msg[12])
{
    int i, j;
    by x, sum;

    memcpy(&gs.t8.state[3*gs.t8.y], msg, 12);

    if (4 == ++gs.t8.y)
        gs.t8.y = 0;

    x = 0;
    for (i = 0; i < 4; i++) {
        sum = 0;
        for (j = 0; j < 12; j++)
            sum += *((pby) &gs.t8.state[3 * i] + j);
        x ^= sum;
    }

    return x;
}

by t9(by msg[12])
{
```

```
    by x = 0;
    int i;

    for (i = 0; i < 12; i++) {
        x ^= msg[i] << (i & 7);
    }

    return x;
}

by t2(by msg[12])
{
    by r7 = t7(msg);
    by r8 = t8(msg);
    by r9 = t9(msg);
    by x = r7 ^ r8 ^ r9;

    if (verbose) {
        printf("Write 7 -> 2: %02x\n", r7);
        printf("Write 8 -> 2: %02x\n", r8);
        printf("Write 9 -> 2: %02x\n", r9);
        printf("Write 2 -> 0: %02x\n", x);
    }

    return x;
}

by t10(by msg[12])
{
    by sum = 0;
    int i;

    memcpy(&gs.t10.state[3 * gs.t10.y], msg, 12);

    if (4 == ++gs.t10.y)
        gs.t10.y = 0;

    for (i = 0; i < 4; i++) {
        sum += *((pby) &gs.t10.state[3 * i]);
    }

    return *((pby) &gs.t10.state[3 * (sum & 3)]
             + ((sum >> 4) % 12));
}

void t11_12(by msg[12], pby r11, pby r12)
{
    /* 11 vars */
    by x = 0;
    /* 12 vars */
    by y, z;

    x = msg[0] ^ msg[3] ^ msg[7];

    goto _12;
_11:
    /* x contains the answer message from 12 */

    x %= 12;
```

```
    *r11 = msg[x];
    return;

_12:
    y = gs.t12.state[1] ^ gs.t12.state[5] ^ gs.t12.state[9];

    memcpy(gs.t12.state, msg, 12);

    z = x;
    x = y;

    z %= 12;
    *r12 = gs.t12.state[z];

    goto _11;
}

by t3(by msg[12])
{
    by r10 = t10(msg);
    by r11, r12;
    t11_12(msg, &r11, &r12);
    by x = r10 ^ r11 ^ r12;

    if (verbose) {
        printf("Write 10 -> 3: %02x\n", r10);
        printf("Write 11 -> 3: %02x\n", r11);
        printf("Write 12 -> 3: %02x\n", r12);
        printf("Write 3 -> 0: %02x\n", x);
    }

    return x;
}

by t0(by enc)
{
    by r1 = t1(gs.t0.state);
    by r2 = t2(gs.t0.state);
    by r3 = t3(gs.t0.state);
    by x, dec;

    x = r1 ^ r2 ^ r3;

    dec = (gs.t0.y + 2 * gs.t0.state[gs.t0.y]) ^ enc;

    gs.t0.state[gs.t0.y] = x;

    if (12 == ++gs.t0.y)
        gs.t0.y = 0;

    return dec;
}
```

F Attaque par force brute

Listing 31: csim.c - partie 2

```
by input[300000];
by decrypted[300000];
int input_len = 0;
int key_pos, enc_pos;
FILE *out;

int try_key(by key[12])
{
    int i;

    reset_global_state();

    memcpy(gs.t0.state, key, 12);

    for (i = enc_pos; i < input_len; i++) {
        by dec = t0(input[i]);
        if (verbose)
            printf("dec(%02x) = %02x (%c)\n", input[i], dec, dec);
        if (out)
            fwrite(&dec, 1, 1, out);
    }

    return 1;
}

int vetted = 0;
int fully_decrypted = 0;
int first_k_test = 0;

#define DUMMY_BUF_SZ (10 * 1024)
#define TRIAL_SZ 256

int vet_key(by key[12])
{
    int i;
    by *tail;
    char dummy[DUMMY_BUF_SZ];
    unsigned int dummy_buf_sz;
    int rc;

    vetted++;

    reset_global_state();

    memcpy(gs.t0.state, key, 12);

    for (i = enc_pos; i < enc_pos + 15; i++) {
        decrypted[i] = t0(input[i]);
    }

    // check the top 5 bits of the 14th byte
    // ('randomized' and 'origPtr' high nybble)

    if (decrypted[enc_pos + 14] >> 3) // must be 0
```

```
    return 0;

first_k_test++;

// decrypt part of 1k and attempt to decompress

for (; i < enc_pos + TRIAL_SZ; i++) {
    decrypted[i] = t0(input[i]);
}

dummy_buf_sz = DUMMY_BUF_SZ;
rc = BZ2_bzBuffToBuffDecompress(dummy, &dummy_buf_sz,
            (char *) &decrypted[enc_pos], TRIAL_SZ, 0, 0);

switch (rc) {
    case BZ_CONFIG_ERROR:
    case BZ_PARAM_ERROR:
    case BZ_MEM_ERROR:
        printf("Unexpected bzip2 error code: rc = %d\n", rc);
        break;

    case BZ_DATA_ERROR_MAGIC:
        printf("Unexpected bzip2 error code: BZ_DATA_ERROR_MAGIC\n");
        return 0;

    case BZ_OUTBUFF_FULL:
    case BZ_UNEXPECTED_EOF:
    case BZ_OK:
        break;  // could be legit

    case BZ_DATA_ERROR:
        return 0;   // bad data stream, bad key
}

// finish decryption

for (; i < input_len; i++) {
    decrypted[i] = t0(input[i]);
}

fully_decrypted++;

// check the end marker at 8 different bit alignments

tail = &decrypted[input_len - 10];

if (0 == memcmp(tail, "\x17\x72\x45\x38\x50\x90", 6) ||
    0 == memcmp(tail, "\x2e\xe4\x8a\x70\xa1", 5) ||
    0 == memcmp(tail, "\x5d\xc9\x14\xe1\x42", 5) ||
    0 == memcmp(tail, "\x77\x24\x53\x85\x09", 5) ||
    0 == memcmp(tail, "\xb9\x22\x9c\x28\x48", 5) ||
    0 == memcmp(tail, "\xbb\x92\x29\xc2\x84", 5) ||
    0 == memcmp(tail, "\xdc\x91\x4e\x14\x24", 5) ||
    0 == memcmp(tail, "\xee\x48\xa7\x0a\x12", 5))
    return 1;

// otherwise not a .bz2 file

return 0;
```

```
}

by c0[] = { 0x5e, 0xde };
by c1[] = { 0x54, 0xd4 };
by c2[] = { 0x1b, 0x9b }; // -> 'h'
          // 0x2f, 0xaf }; // -> '0' (bzip1 deprecated)
by c3[] = { 0x71, 0xf1 }; // -> '9'
          // 0x73, 0x74, 0x75, 0x76, 0xf3, 0xf4, 0xf5, 0xf6 };
by c4[] = { 0x56, 0xd6 };
by c5[] = { 0x7c, 0xfc };
by c6[] = { 0x64, 0xe4 };
by c7[] = { 0x7d, 0xfd };
by c8[] = { 0x69, 0xe9 };
by c9[] = { 0x76, 0xf6 };
by *choices[] = { c0, c1, c2, c3, c4, c5, c6, c7, c8, c9 };
int num_choices[] = { sizeof(c0),
                      sizeof(c1),
                      sizeof(c2),
                      sizeof(c3),
                      sizeof(c4),
                      sizeof(c5),
                      sizeof(c6),
                      sizeof(c7),
                      sizeof(c8),
                      sizeof(c9) };

void enum_keys()
{
    int i0, i1, i2, i3, i4, i5, i6, i7, i8, i9;
    int k10, k11;
    by key[12];

    for (i0 = 0; i0 < num_choices[0]; i0++) {
        key[0] = c0[i0];
    for (i1 = 0; i1 < num_choices[1]; i1++) {
        key[1] = c1[i1];
    for (i2 = 0; i2 < num_choices[2]; i2++) {
        key[2] = c2[i2];
    for (i3 = 0; i3 < num_choices[3]; i3++) {
        key[3] = c3[i3];
    for (i4 = 0; i4 < num_choices[4]; i4++) {
        key[4] = c4[i4];
    for (i5 = 0; i5 < num_choices[5]; i5++) {
        key[5] = c5[i5];
    for (i6 = 0; i6 < num_choices[6]; i6++) {
        key[6] = c6[i6];
        printf("%02x%02x%02x%02x%02x%02x%02x...\n",
            key[0], key[1], key[2], key[3], key[4], key[5], key[6]);
    for (i7 = 0; i7 < num_choices[7]; i7++) {
        key[7] = c7[i7];
    for (i8 = 0; i8 < num_choices[8]; i8++) {
        key[8] = c8[i8];
    for (i9 = 0; i9 < num_choices[9]; i9++) {
        key[9] = c9[i9];
    for (k10 = 0; k10 < 256; k10++) {
        key[10] = k10;
    for (k11 = 0; k11 < 256; k11++) {
        key[11] = k11;
        if (vet_key(key)) {
```

```
            printf("keys vetted: %d, first k test: %d, "
                   "fully decrypted candidates: %d\n", vetted,
                        first_k_test, fully_decrypted);
            char name[100];
            FILE *hnd;
            sprintf(name, "cands/%02x%02x%02x%02x%02x%02x"
                          "%02x%02x%02x%02x%02x%02x.tar.bz2",
                key[0], key[1], key[2], key[3], key[4], key[5],
                key[6], key[7], key[8], key[9], key[10], key[11]);
            printf("%s\n", name);
            if ((hnd = fopen(name, "wb"))) {
                fwrite(&decrypted[enc_pos], 1, input_len - enc_pos, hnd);
                fclose(hnd);
            }
        }
    }}}}}}}}}}}}
}

int main(int argc, char *argv[])
{
    int a = 1, i;
    char *key = NULL;
    int brute_force = 0;

    while (a < argc && '-' == argv[a][0]) {
        if (0 == strcmp(argv[a], "-v")) {
            verbose = 1;
        } else if (a+1 < argc && 0 == strcmp(argv[a], "-k")) {
            key = argv[a+1];
            if (strlen(key) != 12 * 2)
                exit(1);
            a++;
        } else if (0 == strcmp(argv[a], "-bf")) {
            brute_force = 1;
        }
        a++;
    }

    if (a < argc) {
        FILE *hnd = fopen(argv[a], "rb");
        input_len = fread(input, 1, 300000, hnd);
        fclose(hnd);
    }
    else
        exit(0);

    if (a+1 < argc) {
        out = fopen(argv[a+1], "wb");
    }

    for (i = 0; i + 4 <= input_len; i++) {
        if (0 == memcmp(&input[i], "KEY:", 4))
            break;
    }

    i += 4;
    key_pos = i;

    if (key) {
```

```
        int j, k;
        by binkey[12];
        for (j = 0; j < 12; j++) {
            sscanf(&key[2*j], "%02x", &k);
            binkey[j] = (by) k;
        }
        memcpy(&input[key_pos], binkey, 12);
    }

    i += 12;
    i += 1 + input[i];
    enc_pos = i;

    if (brute_force)
        enum_keys();
    else
        try_key(&input[key_pos]);

    if (out)
        fclose(out);

    return 0;
}
```

G Parseur de PNG

Listing 32: splitpng.c

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

int32_t bswap(int32_t x)
{
    return ((x >> 24) | (x >> 8) & 0xff00
          | (x << 8) & 0xff0000 | (x << 24) & 0xff000000);
}

void assert(int cond)
{
    if (! cond)
        exit(1);
}

int main(void)
{
    int32_t length, crc;
    char type[] = "XXXX.xxx";
    int chunk_id = 0;
    char *buf, sig[8];
    FILE *hnd;

    fread(&sig, 1, 8, stdin);

    assert(0 == memcmp(sig, "\x89\x50\x4e\x47\x0d\x0a\x1a\x0a", 8));

    while (! feof(stdin)) {
        assert(1 == fread(&length, 4, 1, stdin));
        assert(1 == fread(type, 4, 1, stdin));
        printf("%.4s\n", type);
        length = bswap(length);
        if (length > 0) {
            assert(!!(buf = malloc(length)));
            assert(length == fread(buf, 1, length, stdin));
            sprintf(type+5, "%03d", chunk_id);
            assert(!!(hnd = fopen(type, "wb")));
            assert(fwrite(buf, 1, length, hnd));
            fclose(hnd);
            free(buf);
        }
        assert(1 == fread(&crc, 4, 1, stdin));
        chunk_id++;
    }

    return 0;
}
```