Solution du challenge SSTIC 2013

Emilien Girault

9 avril 2013

Résumé

Le challenge SSTIC de 2013 consite à retrouver une adresse e-mail en `@sstic.org` dans une capture réseau. Ce document présente la démarche utilisée par l'auteur pour y parvenir, qui se décompose en quatre étapes distinctes : analyse de la capture réseau, rétroconception d'un FPGA, analyse d'un document PostScript, et récupération de l'adresse e-mail obfusquée au sein d'une vCard. Les scripts et outils développés sont présentés dans ce document au fil du raisonnement.

Table des matières

Table des figures

1 Capture réseau et canaux cachés

1.1 Découverte du challenge

Le challenge se présente sous la forme d'une capture réseau, téléchargeable à l'adresse http://static.sstic.org/challenge2013/dump.bin.

La commande `file` indique qu'il s'agit d'une capture au format PCAP :

```
$ file dump.bin
dump.bin: tcpdump capture file (little-endian) - version 2.4
(Ethernet, capture length 65535)
$ mv dump.bin dump.pcap
```

Cette capture peut être analysée avec des outils tels que Wireshark [1] ou Scapy [2]. La fonction `rdpcap()` de Scapy permet d'ouvrir la capture, dont un aperçu est obtenu grâce à `summary()` :

```
>>> l=rdpcap("dump.pcap")
>>> l
<dump.pcap: TCP:163 UDP:0 ICMP:65 Other:0>
>>> l.summary()
Ether / IP / TCP 192.168.1.13:57648 > 192.168.1.12:1234 S
Ether / IP / TCP 192.168.1.13:57648 > 192.168.1.12:1234 A
Ether / IP / TCP 192.168.1.13:57648 > 192.168.1.12:1234 PA / Raw
Ether / IP / TCP 192.168.1.13:57648 > 192.168.1.12:1234 FA
Ether / IP / TCP 192.168.1.13:57648 > 192.168.1.12:1234 A
Ether / IP / ICMP 192.168.1.13 > 192.168.1.12 echo-request 0 / Raw
Ether / IP / ICMP 192.168.1.13 > 192.168.1.12 echo-request 0 / Raw
Ether / IP / ICMP 192.168.1.13 > 192.168.1.12 echo-request 0 / Raw
[...]
Ether / IP / ICMP 192.168.1.13 > 192.168.1.12 echo-request 0 / Raw
Ether / IP / ICMP 192.168.1.13 > 192.168.1.12 echo-request 0 / Raw
Ether / IP / TCP 192.168.1.13:36008 > 192.168.1.12:ftp S
Ether / IP / TCP 192.168.1.13:36008 > 192.168.1.12:ftp A
Ether / IP / TCP 192.168.1.13:36008 > 192.168.1.12:ftp A
Ether / IP / TCP 192.168.1.13:36008 > 192.168.1.12:ftp PA / Raw
Ether / IP / TCP 192.168.1.13:36008 > 192.168.1.12:ftp PA / Raw
Ether / IP / TCP 192.168.1.13:36008 > 192.168.1.12:ftp A
[...]
Ether / IP / TCP 192.168.1.13:36008 > 192.168.1.12:ftp PA / Raw
Ether / IP / TCP 192.168.1.13:36008 > 192.168.1.12:ftp PA / Raw
Ether / IP / TCP 192.168.1.13:44676 > 192.168.1.12:60733 S
Ether / IP / TCP 192.168.1.13:44676 > 192.168.1.12:60733 A
Ether / IP / TCP 192.168.1.13:36008 > 192.168.1.12:ftp PA / Raw
Ether / IP / TCP 192.168.1.13:44676 > 192.168.1.12:60733 A / Raw
Ether / IP / TCP 192.168.1.13:44676 > 192.168.1.12:60733 A / Raw
[...]
```

La capture fait intervenir deux machines d'adresses IP respectives 192.168.1.13 (noté A dans la suite) et 192.168.1.12 (B), mais ne contient que les échanges de A vers B. Cette capture contient trois séries d'échanges :

1. http://www.wireshark.org
2. http://www.secdev.org/projects/scapy

– une connexion TCP de A vers le port 1234 de B ;
– une série de 65 paquets ICMP de type « Echo Request » ;
– une connexion FTP de A vers B, elle-même composée d'un canal de contrôle (port 21) et d'un envoi de données en mode passif (port 60733).

Les données échangées dans les connexions TCP peuvent être obtenues par la fonction « Follow TCP Stream » de Wireshark. Les données envoyées sur le port 1234 de B sont les suivantes :

```
Bonjour,
J'ai egare la cle pour dechiffrer mon carnet d'adresses.
Tu pourrais m'aider a la retrouver ? J'ai besoin de recuperer
une adresse email a l'interieur.
Pour t'aider, je t'envoie :
- une archive chiffree en AES par FTP
- la cle AES par canaux caches
voici l'iv utilise pour AES : 76C128D46A6C4B15B43016904BE176AC
voici le checksum de l'archive pour verifier le dechiffrement :
61c9392f617290642f9a12499de6b688
merci

PS :
Indication pour les canaux caches : 1 bit de canal cache temporel
concatene a 3 bits de canal cache non temporel.
```

Il est question de récupérer une archive chiffrée envoyée par FTP, la clé AES étant envoyée par canaux cachés[3].

Le canal de contrôle FTP peut être dumpé par la même option que précédemment. Cependant il n'apporte aucune information utile, si ce n'est que le mot de passe utilisé ne respecte pas les recommandations de l'ANSSI[4] :

```
USER sstic
PASS sstic
PWD
FEAT
HELP SITE
CLNT NcFTP 3.2.2 linux-x86-glibc2.9
TYPE I
REST 1
REST 0
SIZE sstic.tar.gz-chiffre
MDTM sstic.tar.gz-chiffre
PASV
STOR sstic.tar.gz-chiffre
SITE UTIME sstic.tar.gz-chiffre 20130318113720
20130318113533 20130318113533 UTC
MDTM 20130318113533 sstic.tar.gz-chiffre
QUIT
```

L'archive est récupérée dans la connexion de données FTP (figure 1).

Le contenu du flux TCP est sauvegardé sur disque, puis décodé en Base64.

3. http://fr.wikipedia.org/wiki/Canal_cach%C3%A9
4. http://www.ssi.gouv.fr/IMG/pdf/NP_MDP_NoteTech.pdf

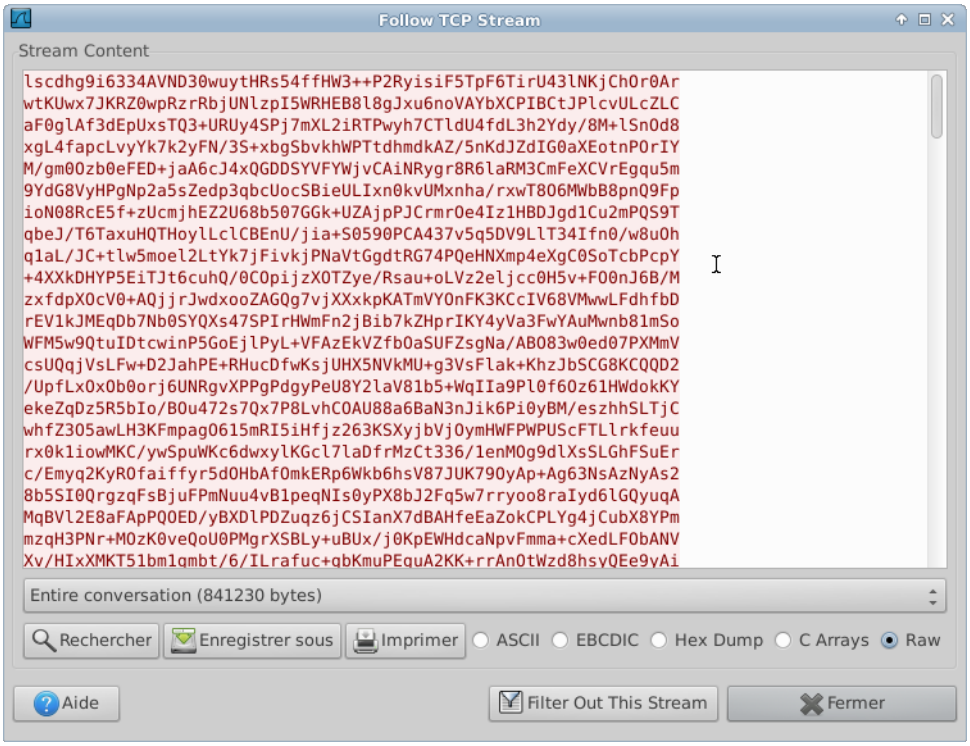

FIGURE 1 – Extrait de l'archive chiffrée

```
$ head -2 data_b64.txt
lscdhg9i6334AVND30wuytHRs54ffHW3++P2RyisiF5TpF6TirU43lNKjChOr0Ar
wtKUwx7JKRZ0wpRzrRbjUNlzpI5WRHEB8l8gJxu6noVAYbXCPIBCtJPlcvULcZLC
$ base64 -d data_b64.txt > data.bin
$ file data.bin
data.bin: data
```

Cependant, l'archive reste chiffrée et il est nécessaire de trouver la clé pour la déchiffrer.

1.2 Identification des canaux cachés

Le message transmis par A indique que la clé est envoyée par deux types de canaux : temporel et non temporel. Les 65 paquets ICMP émis par A laissent penser que ceux-ci véhiculent de l'information susceptible de mener à la clé.

Le canal caché temporel est le plus évident à déterminer. Il s'agit d'observer la différence de temps entre deux paquets ICMP successifs émis par A. L'option « View > Time Display Format > Seconds Since Previous Captured Packet » de Wireshark permet de l'afficher (cf. figure 2).

La délai (appelé `delta` dans la suite) entre deux paquets ICMP successifs étant soit d'une seconde, soit de deux secondes, il est probable que celui-ci code pour un bit d'information de la clé. Cela correspond bien avec le message laissé par A. Les 65 paquets ICMP correspondent à 64 deltas de temps, donc 64 bits de clé. Toutefois, rien n'indique le codage utilisé pour transmettre ces bits. Ainsi, la suite des 7 premiers deltas `[2, 2, 2, 2, 2, 1, 2]` peut aussi bien coder pour la suite de bits `[0, 0, 0, 0, 0, 1, 0]` que pour `[1, 1, 1, 1, 1, 0, 1]`. La seule façon de

No.	Time	Source	Destination
6	0.029843	192.168.1.13	192.168.1.12
7	2.012000	192.168.1.13	192.168.1.12
8	2.013228	192.168.1.13	192.168.1.12
9	2.013045	192.168.1.13	192.168.1.12
10	2.013620	192.168.1.13	192.168.1.12
11	2.013407	192.168.1.13	192.168.1.12
12	1.013347	192.168.1.13	192.168.1.12
13	2.013840	192.168.1.13	192.168.1.12
14	1.013408	192.168.1.13	192.168.1.12
15	1.013339	192.168.1.13	192.168.1.12
16	2.013555	192.168.1.13	192.168.1.12
17	1.013333	192.168.1.13	192.168.1.12
18	2.013527	192.168.1.13	192.168.1.12

FIGURE 2 – Différences de temps entre paquets émis

déterminer le codage utilisé est de tester les deux éventualités.

Les bits transmis par canaux cachés non temporels sont probablement contenus dans les paquets ICMP. La méthodologie permettant de les identifier consiste également à analyser les différences entre les paquets, et de repérer quels sont les champs qui évoluent de façon « anormale ».

Après inspection manuelle de quelques paquets, plusieurs champs semblent comporter des différences :

- dans la couche IP :
  - `ttl`,
  - `tos`, ou `DSCP` sous Wireshark,
  - `checksum` ;
- dans la couche ICMP :
  - `id`,
  - `checksum` ;
- la payload ICMP.

Toutefois, nous pouvons éliminer les deux checksums, dont la valeur n'apporte aucune information supplémentaire non contenue dans les autres champs. Le champ `id` de la couche ICMP semble être un bon candidat ; cependant les variations de celui-ci coïncident avec les variations des délais observées entre les paquets [5]. En effet, si le délai entre les paquets i et $i+1$ est de 2 secondes, la variation correspondante sur l'`id` sera systématiquement de 17. De même, un délai de 1 seconde entraîne une variation de 15 sur l'`id`.

```
>>> t = l[ICMP]
>>> le = lambda x: (x >> 8) | ((x & 0xff) << 8)
>>> [le(t[i+1][ICMP].id) - le(t[i][ICMP].id) for i  in range(10)]
[17, 17, 17, 17, 17, 15, 17, 15, 15, 17]
```

Concernant la payload ICMP, seuls les premiers octets varient. Une analyse rapide semble indiquer qu'il s'agit de deux *timestamps* en little-endian. Le premier est un timestamp standard Unix, l'autre un nombre de microsecondes.

5. Les valeurs de ce champ doivent être considérées en *little-endian*, Scapy ne les affichant qu'en *big-endian* par défaut.

```
>>> from datetime import datetime
>>> timestamps = [struct.unpack("<LL", p[Raw].load[:8]) for p in t]
>>> print '\n'.join('%s %s' % (datetime.fromtimestamp(time), micros) for time, micros in timestamps)
2013-03-18 12:35:34 175450
2013-03-18 12:35:36 187452
2013-03-18 12:35:38 200680
[...]
2013-03-18 12:37:19 94923
2013-03-18 12:37:20 97812
```

Le manuel de l'utilitaire `ping`[6] explique leur présence : *"ECHO_REQUEST datagrams (*`pings`*) have an IP and ICMP header, followed by a* `struct timeval` *and then an arbitrary number of* `pad` *bytes used to fill out the packet"*. L'information contenue dans la payload est donc redondante et peut *a priori* être ignorée.

Le champ `tos` peut valoir 2 ou 4, tandis que le `ttl` peut valoir 10, 20, 30 ou 40. Ces champs sont donc susceptibles de représenter respectivement 1 et 2 bits d'information, le codage de chacun d'entre eux restant encore à déterminer.

```
>>> f = lambda p: (p[IP].tos, p[IP].ttl)
>>> [f(t[i]) for i in range(10)]
[(2, 30), (2, 30), (4, 40), (4, 30), (2, 20), (4, 10), (2, 30), (2, 30), (4, 10), (4, 20)]
```

Avec un `tos` à 0 `ttl` à 1, le dernier paquet ICMP fait office d'exception. En effet, à raison de 4 bits d'information par paquet (un codé par le délai avec le paquet précédent, les 3 autres par `tos` et `ttl`), 64 paquets permettent d'obtenir 256 bits d'information, correspondant à une taille de clé AES valide. Le dernier paquet n'est présent que pour permettre d'obtenir le dernier bit de délai, les valeurs qu'il contient peuvent être ignorées.

Il semble judicieux de supposer que la clé soit composée d'une série de *nibbles*, chacun d'entre eux correspondant à l'interprétation binaire des champs retenus (`tos`, `ttl`, `delta`), pour chaque paquet.

Afin de trouver la clé, il reste à déterminer :
- le codage utilisé pour chacun des champs ;
- la position de ces champs au sein d'un nibble de clé ;
- l'ordre des nibbles au sein de la clé (de gauche à droite ou de droite à gauche) ;

1.3 Bruteforce de la clé

La stratégie retenue consiste à énumérer toutes les possibilités pour les 3 éléments ci-dessus afin de reconstituer la clé, puis de tenter un déchiffrement de l'archive. En plus de ces éléments, il peut sembler judicieux de bruteforcer également le mode d'opération[7] AES, ainsi que l'algorithme de hashage utilisé, puisque A ne les précise pas dans son message laissé à B. Toutefois l'intuition laisse penser qu'il s'agit respectivement de CBC et MD5.

6. http://linux.die.net/man/8/ping
7. http://fr.wikipedia.org/wiki/Mode_d'op%C3%A9ration_(cryptographie)

L'énumération exhaustive est réalisée en Python. Il reste à déterminer la condition d'arrêt à utiliser. La première m'étant venue à l'esprit est la validité du hash du texte déchiffré (`md5(AES.new(key, mode, iv).decrypt(data)).hexdigest() == HASH`). Cependant la bibliothèque PyCrypto [8] ne supprime pas automatiquement le padding PKCS7 [9] lors du déchiffrement. Ayant oublié ce détail lors de la première tentative d'énumération, celle-ci a lamentablement échoué. Pour qu'elle fonctionne, il est nécessaire de s'assurer en premier lieu que le padding est valide et de le retirer manuellement avant de calculer l'empreinte MD5 sur le plaintext.

Une autre solution est d'arrêter l'énumération lorsque le début du bloc déchiffré correspond à la signature (*magic*) du format GZip, soit `\x1f\x8b`, les chances que cela arrive accidentellement étant plutôt rare.

Le script suivant réalise l'énumération, affiche la clé trouvée et déchiffre l'archive.

```
from scapy.all import *
import itertools
from Crypto.Cipher import AES
import hashlib
import sys

l = rdpcap("dump.pcap")
t = l[ICMP]

deltas = [int(t[i+1].time - t[i].time) for i in range(len(t)-1)]
tos_ttl = [(p.tos, p.ttl) for p in t[:-1]]

VALID_HASH = "61c9392f617290642f9a12499de6b688".decode('hex')
IV = "76C128D46A6C4B15B43016904BE176AC".decode('hex')

# Permutations des champs au sein d'un nibble
perms_champs = list(itertools.permutations(range(3), 3))

# Interpretation (codage) de chaque champ
perm_interpret = list(itertools.product(list(itertools.permutations(range(2), 2)), # delta
                                        list(itertools.permutations(range(2), 2)), # tos
                                        list(itertools.permutations(range(4), 4))  # ttl
                                        ))

data = open("data.bin", "rb").read()

# Fonctions d'extraction et de "normalisation" des valeurs des champs
# Prennent en parametre l'indice et l'interpretation du champ voulu
# Retournent un tuple (valeur, taille du champ en bits)
def get_delta(i, pinter):
    return (pinter[deltas[i]-1], 1)
def get_tos(i, pinter):
    return (pinter[tos_ttl[i][0]/2-1], 1)
def get_ttl(i, pinter):
    return (pinter[tos_ttl[i][1]/10-1], 2)

# Concatene 3 champs pour former un nibble
def nibblify(r1, r2, r3):
    return r3[0] | (r2[0] << r3[1]) | (r1[0] << (r3[1]+r2[1]))
```

8. https://www.dlitz.net/software/pycrypto/
9. http://en.wikipedia.org/wiki/Padding_(cryptography)#PKCS7

```
# Permute les champs au sein d'un nibble
def permute(champs, perm):
    return [champs[perm[i]] for i in range(3)]

# Combine les fonctions precedentes
def get_nibble_perm(i, pchamps, pinter):
    n = (get_delta(i, pinter[0]), get_tos(i, pinter[1]), get_ttl(i, pinter[2]))
    r = permute(n, pchamps)
    return nibblify(*r)

# Boucle principale
for pinter in perm_interpret:
  for pchamps in perms_champs:
    nibbles = [get_nibble_perm(i, pchamps, pinter) for i in range(64)]
    for sens in (1,-1): # Gauche a droite ou droite a gauche
      key_hex = "".join("%x" % n for n in nibbles[::sens])
      key = key_hex.decode('hex')
      k = AES.new(key, AES.MODE_CBC, IV)
      dec = k.decrypt(data)

      # Verification du padding
      b = dec[-1]
      if(ord(b) <= 16 and dec[-ord(b):] == b*ord(b)):
        dec = dec[:-ord(b)]

        # Verification du hash et du header GZip
        h = hashlib.md5(dec).digest()
        if(h == VALID_HASH and dec[:2] == "\x1f\x8b"):
          print key_hex, pinter, pchamps, sens
          open("dec.bin", "wb").write(dec)
          sys.exit(0)
```

Le résultat du script est présenté ci-dessous, avec son temps d'exécution.

```
$ time ./combi.py
WARNING: No route found for IPv6 destination :: (no default route?)
dd8cf2d52e69aafb734e3acd0e4a69e83ed93bc4870ecd0d5b6faad86a63ae94
((0, 1), (1, 0), (1, 3, 2, 0)) (0, 2, 1) 1

real        0m3.853s
user        0m3.756s
sys         0m0.080s
```

Le fichier produit ressemble bien à une archive, et son MD5 concorde bien :

```
$ file dec.bin
dec.bin: gzip compressed data, was "archive.tar", from Unix, last modified:
Mon Mar 18 12:24:37 2013
$ mv dec.bin archive.tar.gz
$ md5sum archive.tar.gz
61c9392f617290642f9a12499de6b688  archive.tar.gz
```

2 Analyse du FPGA

2.1 Décompression de l'archive

L'archive qui vient d'être déchiffrée est extraite par la commande `tar` :

```
$ tar -xzvf archive.tar.gz
archive/
archive/smp.py
archive/data
archive/s.ngr
archive/decrypt.py
```

Nous nous retrouvons en présence de quatre fichiers, dont deux scripts Python. Le premier, `smp.py`, se contente de déclarer une liste d'octets :

```
smp = [0x00,
0xb0,
0x10,
# [...]
0xb7,
0x00,
0xbd,]
```

Le script `decrypt.py` est le cœur du problème à résoudre :

```
1  #!/usr/bin/python
2
3  import sys
4  import base64
5  import md5
6
7  import dev
8  import smp
9
10 if len(sys.argv) != 2:
11   print("usage: %s key" % sys.argv[0])
12   sys.exit(1)
13
14 key = int(sys.argv[1], 16)
15 key = [(key >> (i * 8)) & 0xff for i in range(16)]
16
17 result = []
18 d = open("data", "rb").read()
19 dev.init("sp.ngr")
20 for i in range(0, len(d), 224):
21   smd = d[i : (i + 224)]
22   smd = (key, len(smd), smd)
23   dev.send_smd(smd)
24   dev.send_smp(smp.smp)
25   dev.start()
26   dev.wait_finished()
27   result = result + dev.get_data()
28
29 print "".join(result)
```

```
30  result_md5 = md5.new()
31  result_md5.update("".join(result))
32  result_md5 = result_md5.digest()
33  result_md5 = [ord(x) for x in result_md5]
34  target_md5 = "6c0708b3cf6e32cbae4236bdea062979"
35  target_md5 = [int(target_md5[x:(x + 2)], 16) for x in range(0, len(target_md5), 2)]
36  print(["%02x" % x for x in target_md5])
37  print(["%02x" % x for x in result_md5])
38  if result_md5 != target_md5:
39    print("Bad key...")
40    sys.exit(1)
41
42  result = base64.b64decode("".join(result))
43  d = open("atad", "wb")
44  d.write(result)
45  d.close()
46  sys.exit(0)
```

Ce script prend en paramètre une clé de 32 caractères hexadécimaux et la convertit en une liste de 16 octets, constituée en little-endian. Celui-ci utilise ensuite un module Python nommé `dev`, vraisemblablement non standard. La méthode `init()` de ce module est appelée avec comme paramètre la chaîne `sp.ngr`. Après réflexion, il semblerait que cela soit une coquille dans la conception du challenge, et corresponde en réalité au fichier `s.ngr` de l'archive.

Puis, le script ouvre le fichier `data` en lecture, le découpe en blocs de 224 octets maximum, et semble envoyer deux messages `smd` et `smp` au module `dev`. Le premier est composé des 16 octets de clé, et du bloc de données précédé de sa taille en octets. Le second correspond aux octets bruts de la liste contenue dans le fichier `smp.py`. Après avoir envoye ces deux messages, le script se met en attente de données du module `dev`, qu'il concatène au fur et à mesure dans une liste.

Les lignes 31 à 39 correspondent à une comparaison du hash MD5 des données recues avec un hash codé en dur, et peuvent se résumer à `md5.new(".join(result)).hexdump() == "6c070...2979"`.

A ce stade, il semble que le but soit de retrouver la clé utilisée par le module `dev` afin de récupérer les données dont on possède le hash MD5. Il reste à déterminer le rôle de ce module. Celui-ci n'étant *a priori* pas répertorié publiquement, le seul indice dont nous disposons est le fichier `s.ngr`.

La commande `file` ne retourn aucune résultat :

```
$ file s.ngr
s.ngr: data
```

Seules deux chaînes de caractères contenues dans le binaire semblent être intéressantes :

```
XILINX-XDB 0.1 STUB 0.1 ASCII
XILINX-XDM V1.6e
$754=~2?3&bdah!jamcwe*Tbkaoyoek SRVJG+EOIEFNBJK cicoh‘h‘m&zycn!fmqn,twid’hihy#xgd[...]
```

Une recheche Google sur “xilinx ngr” mène à un document de la société Xilinx intitulé *RTL Technology and Schematic Viewers*[10], qui indique que les fichiers au format NGR sont des représentations graphiques de designs FPGA, produits à partir de code de type HDL[11].

Un FPGA[12] est un circuit intégré qui peut être reprogrammé dans le but d'accomplir certaines tâches. L'intuition laisse penser que le module `dev` permet de communiquer avec un FPGA implémentant le design du fichier `s.ngr`. Afin de déterminer quel traitement est appliqué à `data`, `smp` et `smd`, ainsi que les conditions sous lesquelles la clé est valide, il est nécessaire de s'intéresser au design du FPGA en question et de comprendre son fonctionnement.

Le format NGR est propriétaire et peut être lu par les outils de la suite *ISE Design Tools*, disponible sur le site de Xilinx[13]. Après avoir créé un compte sur le site il devient possible de télécharger les 8,5 Go d'installeurs. La suite ISE fonctionne aussi bien sur Linux que Windows, la seule contrainte étant de disposer d'un minimum de 21 Go d'espace disque. Le logiciel nécessite une licence qui peut être obtenue gratuitement sur le site de Xilinx. Une fois l'installation terminée, le fichier `s.ngr` peut être ouvert avec ISE.

2.2 Rétroconception du FPGA

L'interface affiche la liste des composants disponibles et propose de sélectionner ceux que l'on souhaite visualiser. Il semble judicieux de commencer par afficher l'élément racine, `s`. Chaque composant peut être vu comme une boîte noire avec des pins d'entrée (à gauche du composant) et des pins de sortie (à droite). La figure 3 représente le composant `s` vu de l'extérieur.

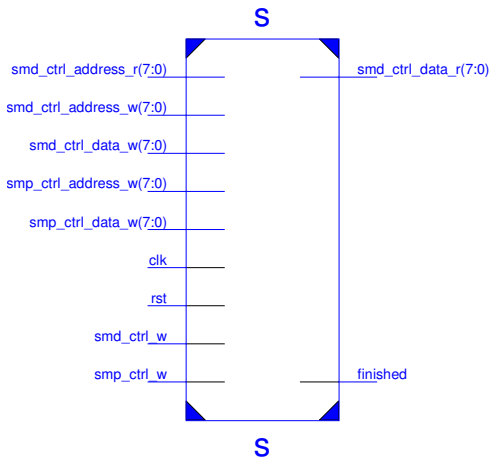

FIGURE 3 – Composant principal du FPGA

Quelques correspondances entre les différents pins du composant et le module `dev` peuvent être faites. On retrouve notemment les mots « smp », « smd », et « finished ». Cependant, n'étant pas programmeur de FPGA, leur rôle ne paraît pas trivial au premier abord. La figure 4 représente l'intérieur de ce composant `s`[14].

10. http://www.xilinx.com/support/documentation/user_guides/ug685.pdf
11. http://en.wikipedia.org/wiki/Hardware_description_language
12. http://en.wikipedia.org/wiki/Field-programmable_gate_array
13. http://www.xilinx.com/support/download/index.htm
14. L'auteur s'excuse pour la piètre lisibilité du schéma

Avant de passer à l'analyse détaillée du circuit, quelques rappels de logique s'imposent.

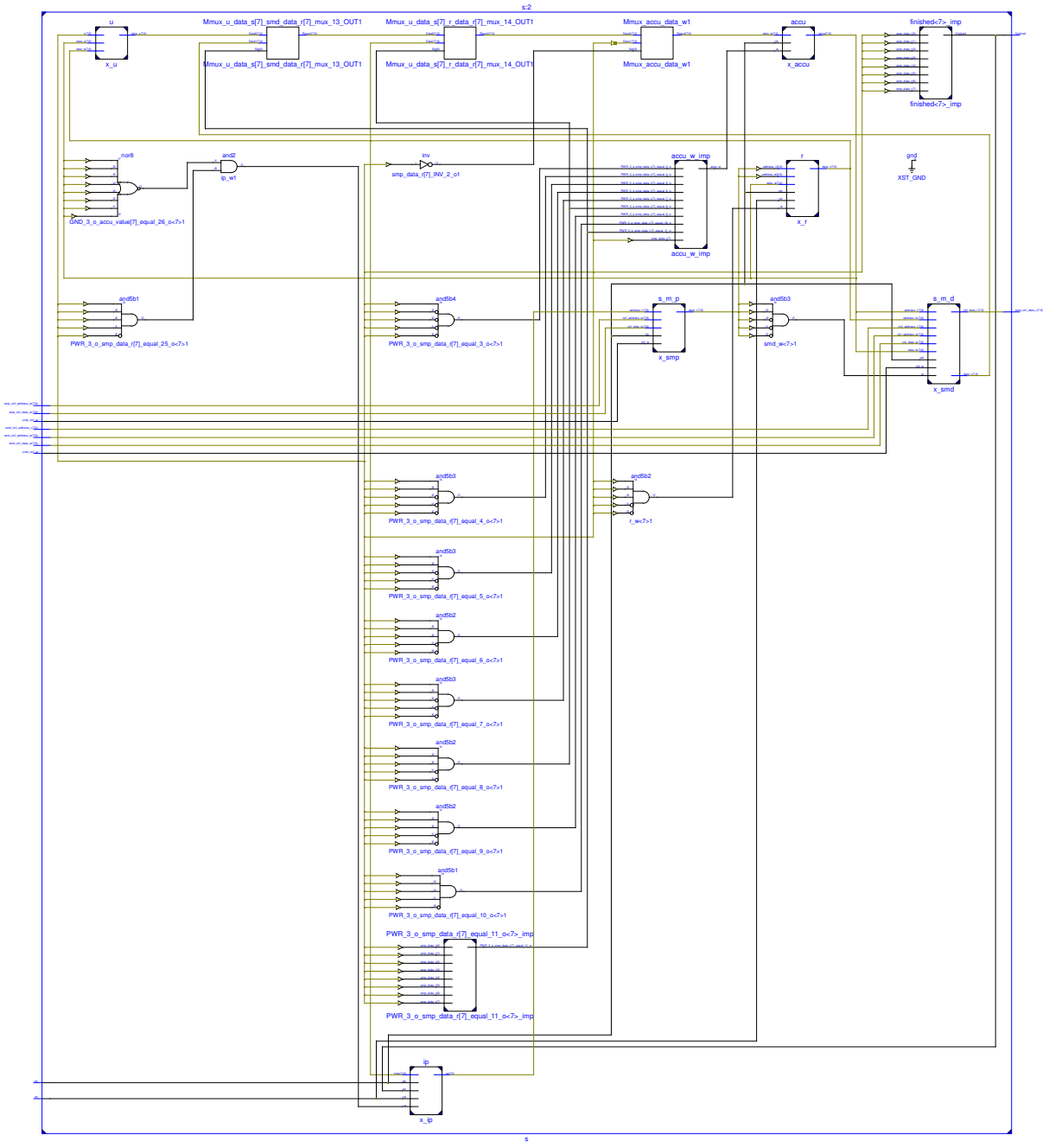

FIGURE 4 – Intérieur de `s`

2.2.1 Eléments de logique

Le composant `s` est constitué d'un ensemble de composants de différents types :
– portes logiques `AND`, `OR`, `NOT` (ou `INV`) [15] ;

15. http://fr.wikipedia.org/wiki/Fonction_logique

– multiplexeurs (`MMUX`) ;
– composants qui sont eux-même constitués d'autres composants (similaire au concept d'agrégation dans la programmation orientée objet).

On retrouve également des composants de la bibliothèque incluse dans la suite ISE, documentés dans le manuel de l'éditeur [16]. Parmi eux, les plus immportants sont les multiplexeurs et les bascules D (ou registres).

Les multiplexeurs sont des composants possédant :
– en entrée :
  – 2^n ensembles de m pins de données ($n \geq 1, m \geq 1$),
  – n pins de sélection ;
– m pins de sortie.

La valeur des pins de sélection indiquent au multiplexeur quels sont les pins de données à laisser passer en sortie. Ce type de composant est généralement utilisé à des fins d'adressage, les pins de sélection correspondant à l'*adresse* de la donnée voulue.

Les bascules D (*flip-flop*), ou registres, font office de mémoires. Elles possèdent une entrée `D` de n bits, un pin d'horloge `C`, et parfois un pin d'activation `CE` (*Clock Enable*). Leur sortie `Q` prend la valeur de `D` lors des fronts montants d'horloge lorsque l'éventuel pin `CE` est à 1. Dans le cas contraire, la valeur de `Q` reste inchangée. Autrement dit, la bascule retourne en permanence sa valeur enregistrée lors du dernier *tick* d'horloge, son pin `CE` faisant office de commande d'écriture.

2.2.2 Analyse du composant `smp`

La méthodologie employée par l'auteur pour décortiquer le FPGA lors du challenge a commencé par une analyse de chaque composant. Dans un premier temps, il faut déterminer d'où viennent les entrées et où vont les sorties du composant en question, puis comprendre la logique qu'il implémente en interne. Les composants de logique combinatoire ne possèdent pas d'entrée de type horloge ni d'état interne, et peuvent se résumer à une simple équation logique (table de vérité). A l'inverse, l'analyse de composant de logique séquentielle (principalement les bascules) nécessite de prendre en compte l'état interne et les valeurs prises dans le passé par celui-ci.

Le composant `smp` est un bon candidat pour commencer l'analyse, car celui-ci comporte un bon nombre de pins d'entrée directement connectés aux pins d'entrée de `s` :
– `ctrl_w` (1 bit) est relié au pin `smp_ctrl_w` de `s` ;
– `clk` (1 bit) est relié au pin `clk` de `s`
– `ctrl_data_w` (8 bits) est relié au pin `smp_ctrl_data_w` de `s` ;
– `ctrl_address_w` (8 bits) est relié au pin `smp_ctrl_address_w` de `s`.
– `address_r` (8 bits) provient du pin de sortie `ip` du composant `ip`.

La sortie `data_r` de `smp` est connectée à un *bus* reliant de nombreux autres composants, principalement des portes `AND`, mais aussi les composants `u`, `r`, `accu_w_imp`, et le multiplexeur `Mmux_accu_data_w1`.

16. http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/spartan6_scm.pdf

La logique interne de `smp` peut être affichée par le logiciel, mais s'agissant d'un composant relativement complexe, celle-ci peut mettre plusieurs minutes avant de s'afficher et ne sera pas représentée ici dans son intégralité.

Il devient rapidement indispensable d'identifier les séquences de composants qui se répètent si l'on ne veut pas se perdre dans l'analyse. L'auteur a tenté de représenter une version simplifiée au maximum du composant, exposée dans la figure 5.

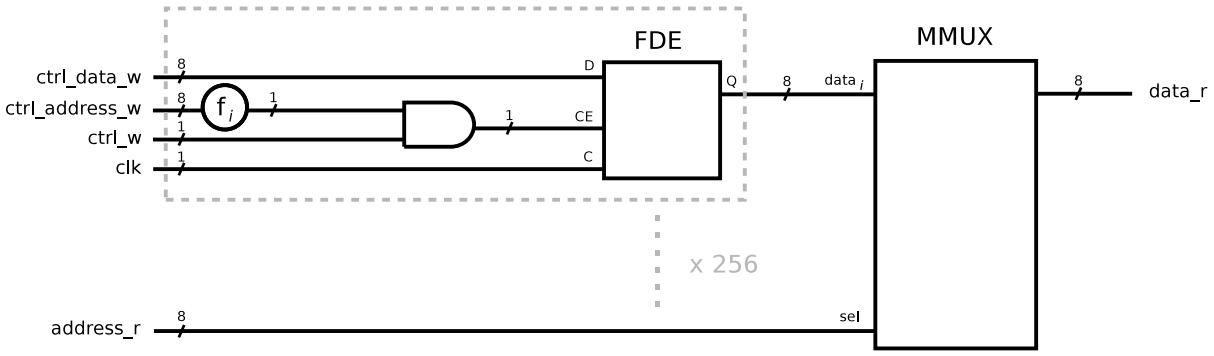

FIGURE 5 – Vue simplifiée de `smp`

Sur ce schéma, f_i représente une fonction logique prenant en entrée la valeur `ctrl_address_w` sur 8 bits et retournant une sortie sur 1 bit. `FDE` représente un registre de 8 bits et `MMUX` un multiplexeur. Le bloc en pointillé est dupliqué 256 fois, chaque fonction f_i étant différente. Le multiplexeur possède au total 256 entrées de données, chacune étant sur 8 bits. Sa sortie correspond à la valeur de sortie du bloc n° `sel`.

Chaque registre prend `ctrl_data_w` en comme donnée d'entrée, mais n'est activé que si f_i(`ctrl_address_w`) $\wedge$ `ctrl_w` est vraie. Autrement dit, f_i détermine si le registre n° i doit mémoriser ou non la valeur `ctrl_data_w` lorsque `ctrl_w` est à 1.

La fonction f_i dépend de chaque bloc. Pour le premier, il s'agit d'un simple `NOR` sur tous les bits de `ctrl_data_w`. f_0 correspond donc à `ctrl_data_w == 0` (cf figure 6)

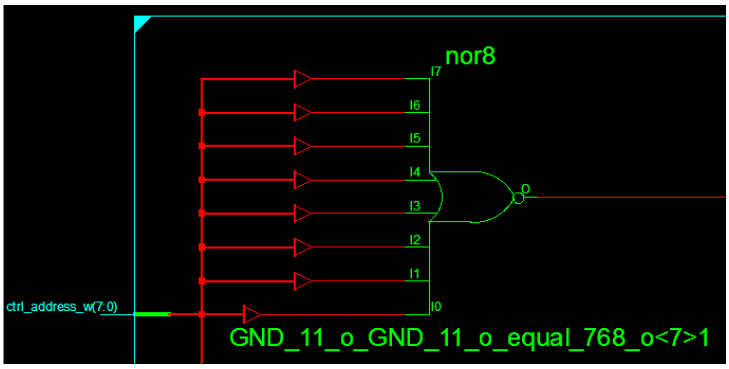

FIGURE 6 – Fonction f_0

Les fonctions f_i pour $i \geq 1$ sont composées de plusieurs portes, mais sont relativement simples. Par exemple, la fonction f_1 est représentée par la figure 7.

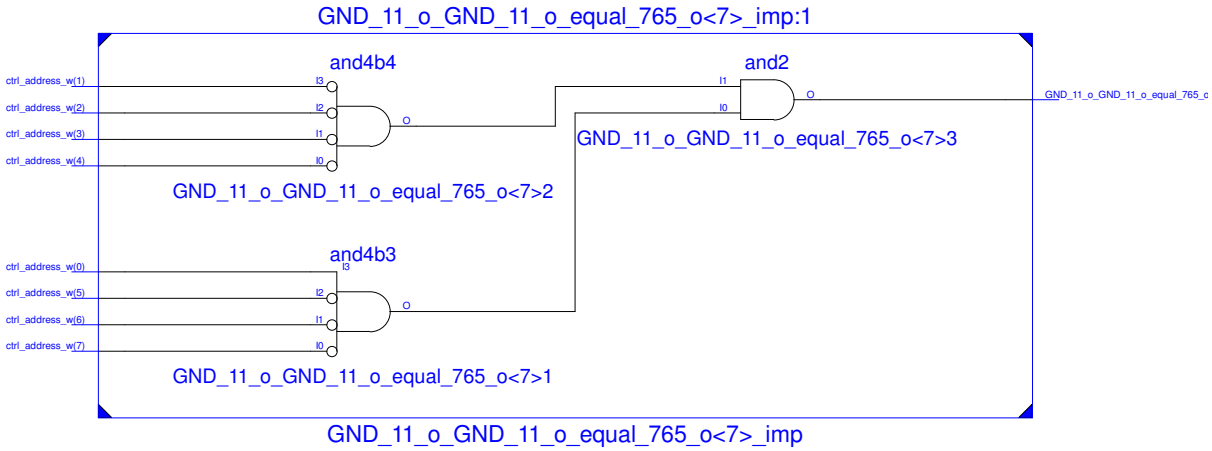

FIGURE 7 – Fonction f_1

On en déduit que f_1 n'est vraie que si le bit 0 de `ctrl_data_w` est à 1, les autres étant à 0. Autrement dit, f_1 correspond à `ctrl_data_w` $== 1$.

Une analyse similaire sur les f_i suivants montre f_i est équivalente à l'équation `ctrl_data_w` $== i$. Autrement dit, l'ensemble des composants figurant dans bloc en pointillés (cf. figure 5) implémente une logique de mémoire adressable de 256 octets. Si l'on souhaite écrire dans cette mémoire, il suffit de positionner l'entrée `ctrl_w` à 1, de placer l'adresse d'écriture dans `ctrl_address_w`, et la donnée dans `ctrl_data_w`. Le multiplexeur situé en sortie de `smp` permet de sélectionner l'adresse de l'octet à lire (`ctrl_address_r`), qui apparaîtra en sortie (`data_r`). L'adresse lue à chaque cycle d'horloge provient systématiquement de la sortie du composant `ip`.

2.2.3 Analyse du composant `smd`

À première vie, le composant `smd` parait encore plus monstrueux [17] que `smp`. Il est cependant très similaire à ce dernier, tout comme son analyse qui ne sera pas détaillée ici. La vue simplifiée du composant est représentée par la figure 8. Pour des raisons de clareté, l'horloge n'est pas représentée [18].

17. en termes de complexité et de temps de chargement dans *ISE*(environ 20 minutes)
18. Celle-ci est simplement relée aux différents registres.

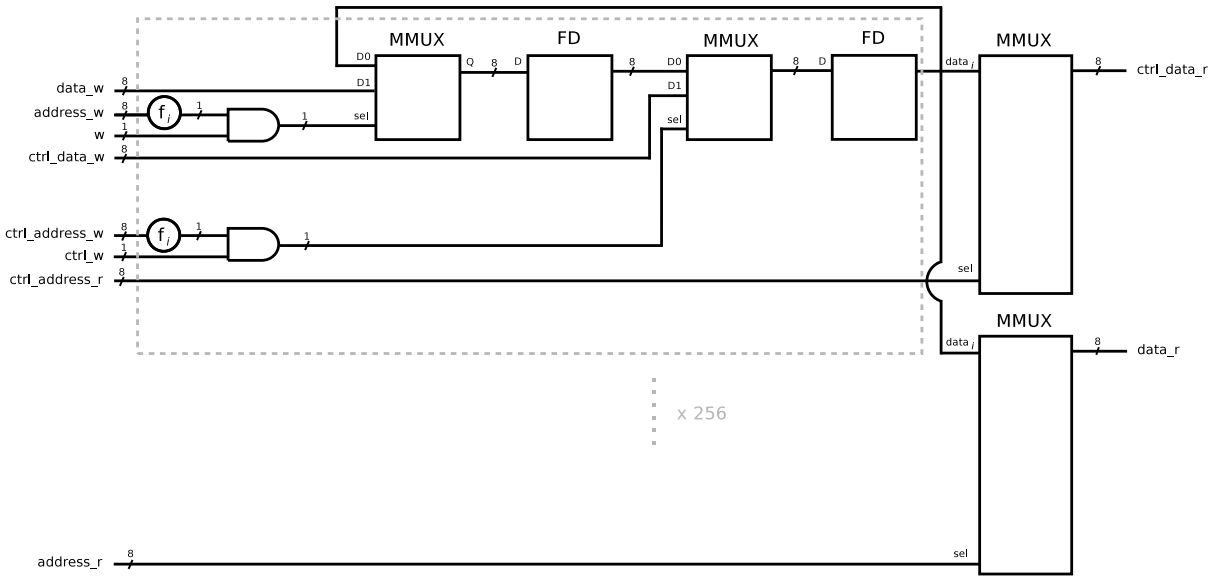

FIGURE 8 – Vue simplifiée de `smd`

Comme `smp`, le composant `smd` fait office de mémoire de 256 octets, à la différence près que celle-ci est manipulable par deux séries de pins différents. Les fonctions f_i sont les mêmes que précédemment et servent d'adressage. Le composant possède deux sorties : `ctrl_data_r` et `data_r`. La première correspond à la valeur stockée en mémoire à l'adresse `ctrl_address_r`. De la même façon, la deuxième permet de lire l'adresse `address_r`. Concernant l'écriture, le comportement est similaire à `smp`. Les pins commençant par `ctrl_`, accessibles depuis l'extérieur de `s` (et utilisés par le module Python), sont prioritaires à leurs homologues.

Le pin d'entrée `w` provient d'une porte logique, qui est à 1 lorsque la valeur sortante de `smp` est de la forme `0b11000xxx`, la valeur des 3 bits `xxx` n'ayant pas d'importance. Lorsque cette condition est vérifiée, la sortie du composant `accu` est écrite dans `smd` à l'adresse provenant du composant `r`.

2.2.4 Point d'étape

Nous venons de rencontrer deux mémoires distinctes. Les deux peuvent être écrites depuis l'extérieur du FPGA, seule `smd` pouvant être lue. Il est supposé que module Python `dev` se charge du transfert de données avec les pins `ctrl_*` de `smp` et `smd`.

Une fois chargée, la mémoire `smp` est systématiquement lue à partir du composant `ip`. La valeur de sortie de `smp` est interprétée, et déclenche sous certaines conditions une action d'écriture dans `smd`.

Ce comportement ressemble étrangement à une architecture matérielle de type Harvard [19]. Puisque sa sortie semble décodée et interprétée, il semblerait que le composant `smp` fasse office de

19. http://en.wikipedia.org/wiki/Harvard_architecture

mémoire de programme (son contenu représentant les instructions à exécuter), dont le pointeur d'instruction courante serait `ip`. De même, `smd` serait la mémoire de données, addressée par le composant `r` (le nom fait penser à un registre), et dont la valeur à écrire provient d'`accu` (accumulateur ?).

Les données des fichiers `data` et `smp.py` sont chargées dans ces deux mémoires, et correspondent respectivement aux données et instructions du microprocesseur implémenté par le FPGA. Afin de comprendre le traitement effectué sur les données (dont la clé), il est nécessaire dans un premier temps de déterminer le jeu d'instructions compris par le microprocesseur, puis de désassembler le code binaire exécuté par celui-ci. Pour ce faire, nous devons tout d'abord terminer l'analyse du circuit logique, et repérer les différents décodages effectués sur l'*opcode* sortant de `smp`. Le nom textuel attribué à chaque type d'instruction est purement arbitraire, son but étant de faire ressortir sa sémantique.

2.2.5 Analyse des composants `r` et `accu`

Ce composant est constitué de 8 registres de 8 bits, toujours adressés de la même façon que précédemment (fonction f_i). Ceux-ci seront notés r_i par la suite. Les entrées `address_w` et `address_r` déterminent quel registre doit être écrit ou lu, et proviennent de la même source : les 3 bits les moins significatifs de l'opcode (sortie de `smp`). La commande d'écriture `w` est active lorsque l'opcode est de la forme `0b10110xxx`. La sortie de `r` est connecté à un *bus* reliant entres autres le pin `address_w` de `smd`.

L'entrée `data_w` vient du composant `accu`. Il s'agit d'un simple registre supplémentaire. Son entrée `data_w` provient d'une série de multiplexeurs, et sa sortie est également reliée à de nombreux composants, dont `smd` via les pins `address_r` et `data_w`.

Ces informations permettent de déduire qu'il existe un type d'instruction que l'on notera `MOV` r_i`, accu`, permettant de recopier la valeur d'`accu` dans le registre r_i. L'opcode doit être égal à `0b10110rrr`, `rrr` correspondant au numéro du registre à écrire.

2.2.6 Analyse du composant `ip`

L'entrée `data` est mémorisée dans un registre lorsque $\texttt{w} \wedge \overline{\texttt{en}}$ est vraie, sachant qu'`en` provient du composant `finished`. Dans le cas contraire, il semblerait que la valeur stockée à l'état précédent soit incrémentée de 1. Cette supposition est basée sur le nom du composant chargé de l'incrémentation, mais ne peut pas être vérifiée explicitement car celui-ci n'est pas inspectable avec le logiciel, et ne semble pas documenté documenté dans le manuel utilisateur.

Ce comportement rappelle effectivement le rôle d'un pointeur d'instruction. La condition d'écriture de celui-ci correspond à un saut. Le pin `w` provient d'une série de portes, dont la sortie est vraie si `accu` vaut 0 et `smp` est de la forme `0b10111rrr`. Comme l'entrée `data` provient de `r`, on en déduit que `rrr` désigne une fois de plus au numéro du registre à utiliser pour fournir l'adresse[20] de saut. L'instruction correspondante est un saut conditionnel (la condition étant

20. Il s'agit d'une adresse absolue.

accu == 0), et sera notée JZ r_i.

2.2.7 Analyse du composant u

Le composant u dispose de deux entrées de données data_a et data_b, abrégées a et b par la suite et reliées respectivement à accu et r. La troisième entrée c correspond à l'opcode et détermine quelle opération sera faite sur a et b. Il s'agit d'une unité logique. Celle-ci est représentée à la figure 9.

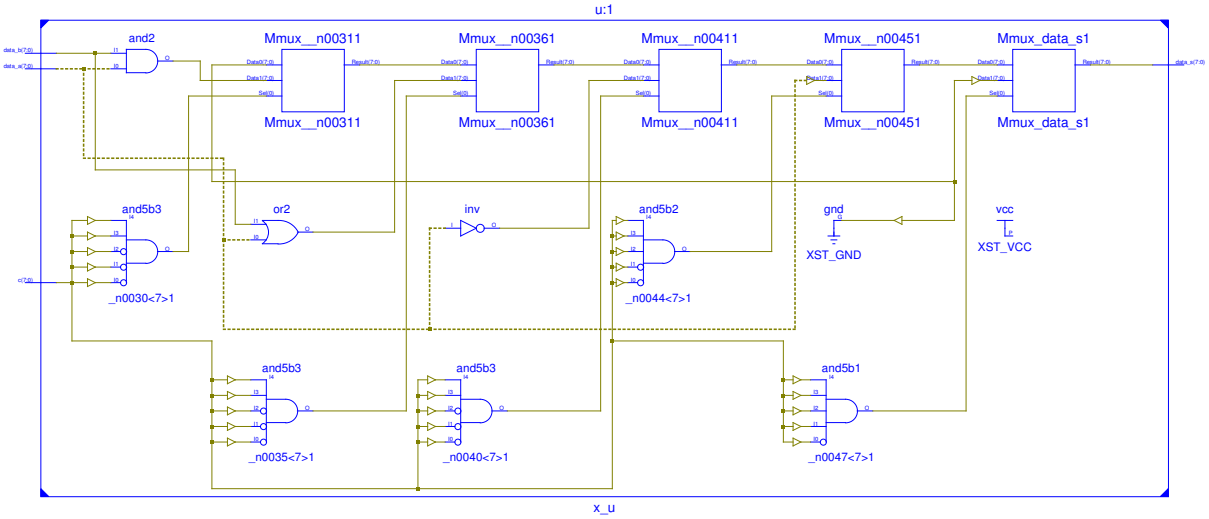

FIGURE 9 – Composant u

Les fonctions a & b, a | b et ~a sont facilement reconnaissables. Celles-ci sont activées lorsque l'opcode vaut respectivement 0b10001xxx, 0b10010xxx, et 0b10100xxx.

Afin de bien comprendre les deux fonctions restantes, il est préférable d'utiliser l'option « Toogle bit/bus mode » d'ISE, et de cocher les cases « Pin names » et « Signal names » dans la boîte de dialogue « Select block pin annotation ».

Une d'entre elle constitue à fixer le bit le plus significatif de a à 1, soit a | 0x80. Cela est réalisé en reliant le dernier pin à VCC, de niveau logique 1. La dernière fonction correspond à a >> 1. En effet, chaque bit n n° i de la sortie correspond au bit $i+1$ de a, le bit le plus significatif étant relié à la masse, donc à 0. Les opcodes respectifs sont 0b11100xxx et 0b11011xxx.

2.2.8 Analyse des composants restants

La sortie du composant u est connectée à une série de 3 multiplexeurs reliés en cascade, chacun ayant 2 entrées de données sur 8 bits. Le pin de sélection de chacun d'entre eux provient d'un décodage de l'opcode courant. Le dernier multiplexeur est connecté à accu.

Le premier multiplexeur retourne en sortie ou bien la valeur de `u`, ou celle de `smd`, si l'opcode est égal à `0b11010000`. Cela correspond donc à une instruction de lecture en mémoire. Comme `smd` n'est adressable en lecture que via `accu`, il semble judicieux de nommer cette instruction `LOAD accu, [accu]`.

Le multiplexeur retourne soit la valeur précédente, soit celle en sortie de `r`, si l'opcode est du type `0b10101rrr`. S'agissant d'une copie de la valeur d'un registre dans `accu`, l'instruction est notée `MOV accu,` r_i.

Le dernier multiplexeur de la série permet de charger les 7 bits les moins significatifs de l'opcode dans `accu` si l'opcode est de la forme `0b0ccccccc`. Il s'agit d'un chargement de constante, noté `MOV accu, CONST`.

Le composant `finished` détermine si le programme a terminé son exécution, auquel cas le registre `ip` n'est plus incrémenté. Bien qu'il s'agisse d'une instruction similaire à une boucle infinie, l'auteur préfère la noter `RET`. La valeur de l'opcode doit être égale à `0b11001000`, comme le montre la figure 10.

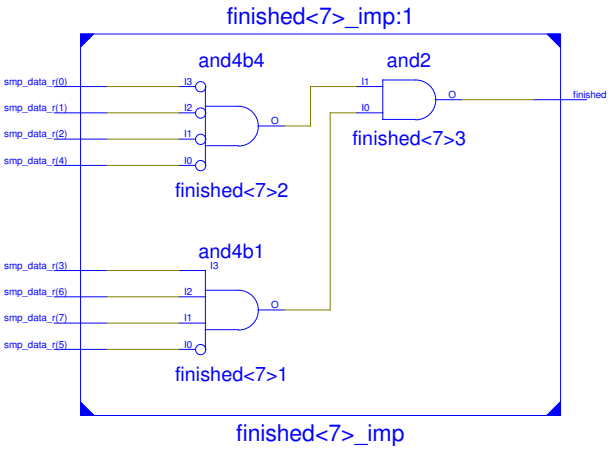

FIGURE 10 – Composant `finished`

2.2.9 Résumé du jeu d'instruction

Après recoupement des informations précédentes, les instructions du microprocesseur implémenté par le FPGA sont les suivantes :

- `AND accu,` r_i, si `opcode == 0b10001rrr` ;
- `OR accu,` r_i, si `opcode == 0b10010rrr` ;
- `NOT accu,` r_i, si `opcode == 0b10100rrr` ;
- `OR accu, 0x80`, si `opcode == 0b11100xxx` (voir ci-dessous) ;
- `SHL accu, 1`, si `opcode == 0b11011xxx` (idem) ;
- `MOV reg, accu`, si `opcode == 0b10110rrr` ;
- `MOV accu,` r_i, si `opcode == 0b10101rrr` ;
- `MOV accu, CONST`, si `opcode == 0b0ccccccc` ;
- `LOAD [accu], accu`, si `opcode == 0b11010000` ;
- `STORE [`r_i`], accu`, si `opcode == 0b11000rrr` ;

– `JZ` r_i, si `opcode == 0b10111rrr` ;
– `RET`, si `opcode == 0b11001000`.

Les instructions `OR accu, 0x80` et `SHL accu, 1` ne prenant aucun paramètre qui puisse varier (i.e. pas de numéro de registre), il est raisonnable de penser que la partie `xxx` de l'opcode soit fixe. Cette hypothèse est vérifiée en parcourant la liste des instructions contenues dans `smp.py` : `xxx` vaut systématiquement `0b000`.

Afin de décoder le programme `smp.py`, il est indispensable de coder un désassembleur. Celui-ci pourra être complété par un émulateur, pour être en mesure de faire tourner le programme. L'ajout de fonctionnalités de débogage dans celui-ci se révèle très intéressant pour l'analyse dynamique du programme.

2.3 Développement d'un désassembleur et d'un émulateur

Un désassembleur minimal est facilement implémentable en Python. Celui-ci se contente d'afficher les instructions du module `smp` sous forme textuelle. Quelques bribes d'exécution symbolique peuvent être ajoutées afin de rendre plus agréable la lecture du listing assembleur produit. L'auteur s'est contenté d'ajouter des commentaires sur chaque ligne pour simplifier les affectations (`MOV accu, CONST` ; `MOV` r_i`, accu` est simplifié en `MOV` r_i`, CONST`) et les sauts inconditionnels (`MOV accu, 0; JZ` r_i est simplifié en `JMP` r_i). Le code du désassembleur est le suivant :

```
1  #!/usr/bin/python
2
3  import smp
4  from collections import defaultdict
5
6  OPCODES_RAW = {
7    0b11001000: 'RET',
8    0b11010000: 'LOAD accu, [accu]',
9    0b11100000: 'OR accu, 0x80',
10   0b11011000: 'SHL accu, 1',
11   0b10100000: 'NOT accu',
12 }
13
14 OPCODES_REG = {
15   0b10001: 'AND accu, %s',
16   0b10010: 'OR accu, %s',
17   0b10110: 'MOV %s, accu',
18   0b10101: 'MOV accu, %s',
19   0b11000: 'STORE [accu], %s',
20   0b10111: 'JZ %s',
21 }
22
23 ACCU = None
24 REGS = dict((i, None) for i in range(8))
25
26 def reg_name(reg):
27   return 'r%d' % reg
28
29
30 def disas_opcode(opcode):
```

```
    """
    Retourne la description de l'instruction, sous forme de chaine.
    Si c'est un saut, retourne un tuple (chaine, adresse de destination du saut)
    """

    global REGS, ACCU

    if(opcode in OPCODES_RAW):
      if(opcode == 0b11100000):
        ACCU |= 0x80
      else:
        ACCU = None
      return OPCODES_RAW[opcode]

    elif(opcode >> 7) == 0:
      s = 'MOV accu, 0x%02x' % (opcode & 0b1111111)
      ACCU = (opcode & 0b1111111)
      return s

    else:
      op, reg = (opcode >> 3, opcode & 0b111)
      if(op in OPCODES_REG):
        s = OPCODES_REG[op] % reg_name(reg)
        if(op == 0b10110 and ACCU is not None):
          REGS[reg] = ACCU
          s = s.ljust(20) + "# r%d = 0x%02x" % (reg, ACCU)
        elif(op == 0b10111):
          dest = "??"
          if(REGS[reg] is not None):
            dest = REGS[reg]
            s = s.ljust(20) + "# %s %02x" % (("JZ", "JMP")[ACCU==0], dest)
            ACCU = None
            REGS[reg] = None
        else:
          ACCU = None
          REGS[reg] = None
        return s

    return 'WTFBBQ?!' # Au cas ou l'instruction ne parvienne pas a etre decodee

print "\n".join('%02x: %s' % (i, disas_opcode(o)) for i,o in enumerate(smp.smp))


```

Voici un extrait du listing produit :

```
$ ./disas.py
00: MOV accu, 0x00
01: MOV r0, accu        # r0 = 0x00
02: MOV accu, 0x10
03: LOAD accu, [accu]
04: MOV r7, accu
05: MOV accu, r0
06: NOT accu
07: MOV r6, accu
08: MOV accu, 0x0e
09: MOV r5, accu        # r5 = 0x0e
0a: MOV accu, 0x71
0b: MOV r4, accu        # r4 = 0x71
```

```
0c: MOV accu, 0x00
0d: JZ r4               # JMP 71
[...]
```

De nombreux sauts sont présents, ce qui ralentit l'analyse. Étant habitué à IDA [21] et à sa visualisation basée sur des graphes, mais n'ayant pas la motivation nécessaire pour ajouter le support d'un processeur, l'auteur reproduit dans un premier temps le graphe de flot de contrôle du programme à la main, sur papier.

L'appel à un ami (Yoann Guillot) permet d'obtenir le support du processeur dans Metasm [22], grâce auquel un graphe nettement plus propre est obtenu.

Il apparaît que le programme est constitué d'une boucle principale, dans laquelle le registre `r0` est incrémenté. Celui-ci débute à 0 et est incrémenté à chaque tour. Le nombre maximal de tours est égal à `SMD[0x10]`, ce qui correspond à la longueur du bloc de données, soit 224 ou moins (pour le dernier bloc).

Afin de faciliter la compréhension de l'algorithme utilisé, un émulateur est développé, toujours en Python. Celui-ci permet de visualiser l'évolution des registres et de la mémoire `SMD` lors de l'exécution du programme. L'ajout de points d'arrêt ainsi que d'instructions de débogage simplifie grandement l'analyse. Le cœur de cet émulateur est présenté dans l'extrait suivant

```
1  import smp
2
3  class FPGADebugger:
4
5    OPCODE_RET = 0b11001000
6    OPCODE_LOAD = 0b11010000
7    # [...]
8
9    def handle_opcode(self, opcode, debug = True, debug_op=False):
10     '''
11     Decode l'opcode et l'execute. Retourne False si le programme doit etre arrete (RET)
12     debug: affiche les lectures et ecritures dans SMD
13     debug_op: affiche chaque instruction executee
14     '''
15
16     if(debug_op):
17       print "%02x: %s" % (self.ip, self.opcode_str(opcode))
18
19     ip_inc = True
20
21     if(opcode in self.OPCODES_RAW):
22
23       if(opcode == self.OPCODE_RET):
24         return False
25
26       elif(opcode == self.OPCODE_LOAD):
27         if(debug):
28           print "\taccu <- SMD[%02x] = %02x" % (self.accu, self.smd[self.accu])
29         self.accu = self.smd[self.accu]
30
31       elif(opcode == self.OPCODE_SHL_ACCU_1):
```

21. https://www.hex-rays.com/products/ida/index.shtml
22. http://metasm.cr0.org/

```
32          self.accu = (self.accu << 1) & 0xff
33
34        elif(opcode == self.OPCODE_NOT_ACCU):
35          self.accu = (~self.accu) & 0xff
36
37        elif(opcode == self.OPCODE_OR_ACCU_80):
38          self.accu = self.accu | 0x80
39
40        elif(op == OPCODE_RET):
41          return False
42
43        else:
44          raise Exception("Instruction non geree")
45
46      # Mov const
47      elif(opcode >> 7 == 0):
48        self.accu = (opcode & 0b1111111)
49
50      else:
51        op, reg = (opcode >> 3, opcode & 0b111)
52        r = self.regs[reg]
53
54        if(op in self.OPCODES_REG):
55
56          if(op == self.OPCODE_AND):
57            self.accu &= r
58          elif(op == self.OPCODE_OR):
59            self.accu |= r
60          elif(op == self.OPCODE_MOV_REG_ACCU):
61            self.regs[reg] = self.accu
62          elif(op == self.OPCODE_MOV_ACCU_REG):
63            self.accu = r
64          elif(op == self.OPCODE_STORE):
65            if(debug):
66              print "\tSMD[%02x] <- %02x" % (r, self.accu)
67            self.smd[r] = self.accu
68          elif(op == self.OPCODE_JZ):
69            if(self.accu == 0):
70              self.trace.append((self.ip, r))
71              self.ip = r
72              ip_inc = False
73              if(debug_op):
74                print
75
76          else:
77            raise Exception("Instruction non geree")
78
79      if(ip_inc):
80        self.ip += 1
81
82      return True
83
84    def next(self, debug=True, debug_op=False):
85      opcode = self.smp[self.ip]
86      return self.handle_opcode(opcode, debug, debug_op)
87
88
89    def run(self, debug=True, debug_op=False):
90      '''
91      Lance l'execution jusqu'au prochain breakpoint ou RET.
```

```
92      Retourne True si la cause d'arret est un bp, False si c'est un RET
93      '''
94      cont = True
95
96      while cont:
97        cont = self.next(debug, debug_op)
98
99        # breakpoint?
100       if(self.ip in self.bps):
101         # [...]
102         return True
103
104     return False
105
106 ######################################################
107 # Main
108 ######################################################
109
110 data = open("data", "rb").read()
111
112 key_str = "11223344556677889 9aabbccddeeff00"
113 key = int(key_str, 16)
114 bloc = map(ord, data[:224])
115 smd = [(key >> (i * 8)) & 0xff for i in range(16)] + [len(bloc)] + bloc
116
117 f = FPGADebugger(smp.smp, smd)
118 f.run()
```

Voici un extrait d'une trace produite par l'émulateur et commentée par les soins de l'auteur :

```
$ ./fdb.py
        accu <- SMD[10] = e0 # (1)
        accu <- SMD[11] = d4 # (2)
        accu <- SMD[00] = 00 # (3)
        accu <- SMD[11] = d4 # (2 bis)
        accu <- SMD[00] = 00 # (3 bis)
        SMD[11] <- 2b
        accu <- SMD[10] = e0 # (1)
        accu <- SMD[12] = 8f # (2)
        accu <- SMD[01] = ff # (3)
        accu <- SMD[12] = 8f # (2 bis)
        accu <- SMD[01] = ff # (3 bis)
        SMD[12] <- 0e
[...]
```

2.4 Rétroconception du programme

Un motif récurrent se distingue, composé de cinq lectures (certaines sont redondantes) suivies d'une écriture dans `SMD`. La première lecture correspond à la longueur du bloc de données, la suivante à l'octet i de ce bloc, et la troisième à l'octet i de la clé. Suite à un traitement encore non identifié, le premier octet du bloc de données est réécrit.

En combinant l'analyse statique essentiellement manuelle et l'analyse dynamique à l'aide de l'émulateur, le programme parvient à être compris dans son intégralité et réécrit de façon

équivalente et simplifiée en Python.

Une sous-boucle située au sein de la boucle principale du programme a pour objectif de réaliser l'addition des registres `r6` et `r7`, bits par bits[23]. Il semble que le coeur du programme soit issu d'une transformation de type `CFG Flattening`. Il s'agit d'un type d'obfuscation qui transforme une suite de blocs d'instructions séquentiels en boucle. A chaque itération, un bloc d'instructions différent est exécuté, en fonction d'une variable d'état. Dans le cas du programme étudié ici, la variable d'état correspond au registre `r5` et contient l'adresse du bloc à exécuter, qui peut prendre 4 valeurs : `0e`, `16`, `24` et `4c`. Les instructions `MOV accu, 0 ; JZ r5` situées à la fin du programme ont pour but de rediriger le flot d'exécution vers le bloc adapté.

Le code simplifié du programme est le suivant :

```
for r0 in xrange(SMD[0x10]):

  r7 = (SMD[r0+0x11] ^ SMD[r0 & 0x0f])

  # r4 = bitreverse(r7)
  r5 = 1
  r4 = 0
  while(r5):
    r4 = (r4 << 1) & 0xff
    if(r5 & r7):
      r4 |= 1
    r5 = (r5 << 1) & 0xff

  SMD[r0+0x11] = r4

  r0 = (r0 + 1) & 0xff
```

Ainsi, chaque octet de données est xorrés avec l'octet correspondant de la clé, avant de voir ses bits permutés.

2.5 Déchiffrement des données

Afin de trouver la clé, il suffit de bruteforcer chaque octet et de telle sorte à ce que les données résultantes soient interprétables en Base64, chaque octet pouvant être bruteforcé indépendamment des autres.

Les premières tentatives d'énumération se soldent par un échec : il s'avère qu'aucune clé ne permet d'obtenir des données en pure Base64. L'auteur rajoute alors les caractères *saut de ligne* et *retour charriot* à la liste des caractères valides, ce qui permet de retrouver la clé. Le code suivant permet de réaliser l'énumération :

```
data = open("data", "rb").read()
nb_blocs = len(data)/16
data_split = [[ord(c) for c in data[i*16 : (i+1)*16]] for i in range(nb_blocs+1)]

base64_charset = range(ord('A'), ord('Z')+1) + \
                 range(ord('a'), ord('z')+1) + \
```

23. Le processeur n'implémentant pas d'instruction d'addition, celle-ci est réalisée de façon logicielle.

```
                range(ord('0'), ord('9')+1) + \
                [ord('+'), ord('/'), ord('\r'), ord('\n')]

# Pour chaque octet de la cle
key = []
for i in range(16):

  # Bruteforce l'octet
  for c in range(256):

    ok = True
    for bloc in data_split[:-1]:

      # Calcule l'octet dechiffre
      d = int(bin(bloc[i]^c)[2:].rjust(8, '0')[::-1], 2)
      if(d not in base64_charset):
        ok = False
        break

    if(ok):
      key.append(chr(c))
```

Le résultat est instantané :

```
$ time ./dexor.py
25a16d1ed323ac65c658ef16bcdc83e6

real        0m0.206s
user        0m0.200s
sys         0m0.004s
```

Les données déchiffrées sont récupérées en utilisant la clé obtenue :

```
def decrypt(key):
  #res = [0 for i in range(256)]
  res = []
  for i, c in enumerate(data):
    n = int(bin(ord(c)^ord(key[i&0xf]))[2:].rjust(8, '0')[::-1], 2)
    res.append(chr(n))

  return ''.join(res)

KEY = "25a16d1ed323ac65c658ef16bcdc83e6"
print decrypt(KEY.decode('hex')[::-1])
```

Le résultat est bien du Base64 et est décodé dans le fichier output.bin.

```
$ ./dexor.py  | base64 -d > output.bin
```

3 Analyse du fichier PostScript

3.1 Échauffement

Le fichier `output.bin` obtenu précédemment peut être ouvert dans un éditeur de texte et ressemble à ceci :

```
/I1 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter /ReusableStreamDecode filter
cf760bc77db1f282e881ede9a10122b220887466b973b854218b85c230d6733ab459fda9a8799736641303l2d5ff3e1a8e2f25
[...]
dup length string 0 3 -1 roll { 3 -1 roll dup 4 1 roll exch 2 index exch put 1 add } forall pop 4 -1
roll dup 5 1 roll 3 1 roll dup 4 1 roll putinterval exch pop } for 0 1 1 {pop pop} for cvx exec } if
false } { (no key provided\n) error true } ifelse } { (missing '--' preceding script file\n) error
true } ifelse { (usage: gs -- script.ps key\n) error flush } if } bind def main clear quit
```

La réaction de l'auteur est alors la suivante :

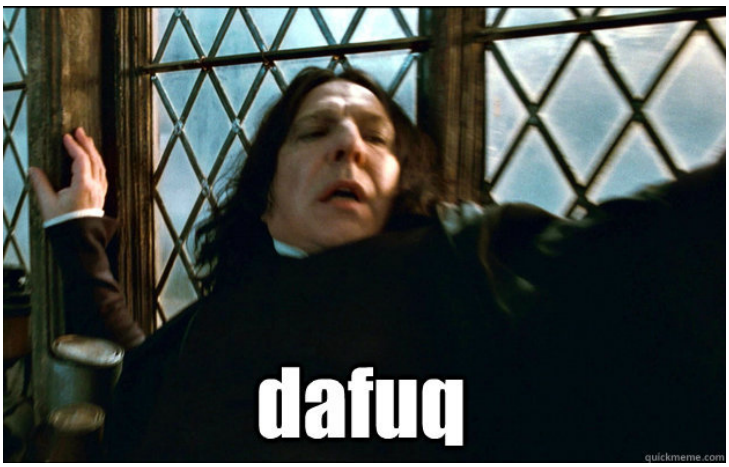

FIGURE 11 – Réaction de l'auteur

D'après l'indication présente vers la fin du fichier, il semblerait qu'il s'agisse d'un fichier PostScript[24]. Celui-ci est exécutable avec l'interpréteur `gs` et attend une clé passée en argument :

```
$ mv output.bin script.ps
$ gs script.ps key
GPL Ghostscript 9.05 (2012-02-08)
Copyright (C) 2010 Artifex Software, Inc.  All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
missing '--' preceding script file
usage: gs -- script.ps key
$ gs -- script.ps key
GPL Ghostscript 9.05 (2012-02-08)
Copyright (C) 2010 Artifex Software, Inc.  All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
$ gs -- script.ps 00112233445566778899aabbccddeeff
GPL Ghostscript 9.05 (2012-02-08)
Copyright (C) 2010 Artifex Software, Inc.  All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
```

24. http://fr.wikipedia.org/wiki/PostScript

Aucun message d'erreur n'étant affiché, il apparaît que seule une analyse approfondie du script permettra de trouver une clé valide.

Dans un premier temps, il est nécessaire de se familiariser avec PostScript. Il s'agit d'un langage de description de documents utilisé par les imprimantes, mais c'est en réalité un véritable langage de programmation Turing-complet. Celui-ci est essentiellement basé sur une pile, chaque instruction (appelée également *opérateur*) agissant sur celle-ci. L'interpréteur PostScript a la particularité de fonctionner comme un calculateur en notation polonaise inversée. Ainsi, les opérateurs apparaissent après leurs arguments. La documentation du langage et de ses opérateurs est contenue dans le manuel de référence d'Adobe [25].

Avant de se lancer dans l'analyse, il paraît indispensable d'indenter le programme. L'auteur a choisi utiliser l'outil `py-ps-parser` [26], qui fait office de *lexer*. Celui-ci produit une sortie avec un lexème par ligne, qu'il est ensuite possible de réindenter grâce à un script tel que :

```
#!/usr/bin/python
# indent.py : indente un script PostScript comportant un lexeme par ligne
import sys

tab = 0
n = 1
for l in sys.stdin.read().split('\n'):
  l = l.rstrip()

  if(l == '}'):
    tab -= 1

  print ('  '*tab) + l
  n += 1

  if(l == 'def'):
    print
    n += 1

  if(l == '{'):
    tab += 1
```

Bien que toujours indigeste à première vue, le résultat est nettement plus lisible :

```
$ git clone git://github.com/haxwithaxe/py-ps-parser.git
Cloning into 'py-ps-parser'...
remote: Counting objects: 223, done.
remote: Compressing objects: 100% (218/218), done.
remote: Total 223 (delta 132), reused 0 (delta 0)
Receiving objects: 100% (223/223), 25.43 KiB, done.
Resolving deltas: 100% (132/132), done.
$ cd py-ps-parser/src/
$ cat ../../script.ps | ./main.py | ../../indent.py > ../../script_indent.gs
$ cd ../../ && cat script_indent.gs
/I1
currentfile
0
(cafebabe) /SubFileDecode
```

25. http://www.adobe.com/products/postscript/pdfs/PLRM.pdf
26. https://github.com/haxwithaxe/py-ps-parser/

```
filter
/ASCIIHexDecode
filter
/ReusableStreamDecode
filter
cf760bc77db1f282e881ede9a10122b220887466b973b854218b85c230d6733ab
[...]
/error
{
  (%stderr) (w) file
  exch
  writestring
}
bind
def
[...]
  {
    (missing '--' preceding script file\n) error
    true
  }
  ifelse
  {
    (usage: gs -- script.ps key\n) error
    flush
  }
  if
}
bind
def

main
clear
quit
```

Le script débute par la déclaration de 4 variables `I1`, `I2`, `I3` et `I4`. Ces variables sont encodées en héxadécimal, et sont décodées à la volée par le mécanisme de filtres PostScript. La suite d'opérateurs `main { [...] } bind def` définit un nouvel opérateur nommé `main`. Il peut s'agir d'une fonction prenant en paramètre des valeurs placés sur la pile, ou d'une simple variable (ne prenant aucun paramètre). La séquence délimitée par les accolades est vue comme un bloc de code, qui est empilé au moment de sa déclaration, et qui ne sera exécuté que lors de l'appel à `main`. Cette dernière est visiblement la fonction principale du programme.

Dans le but de faciliter l'analyse statique du script, un parser minimal a été écrit en Python. Celui-ci prend en entrée la liste de lexèmes et construit un dictionnaire dont chaque clé correspond à un niveau d'indentation. Le code de l'outil est le suivant :

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

import sys
from collections import defaultdict
from pprint import pprint

class Bloc(list):
  pass

class Tree(defaultdict):

```

```
  def __getitem__(self, i):
    if(type(i) == slice):
      start, step, stop = i.start, i.step, i.stop
      if(start is None):
        start = 0
      if(step is None):
        step = 1
      if(stop is None):
        stop = len(self)
      return [self[j] for j in range(start, stop, step)]

    else:
      return defaultdict.__getitem__(self, i)

f = Tree(list)
level = 0

for n, l in enumerate(open(sys.argv[1], "rb").read().split('\n')):
  l = l.rstrip().lstrip()
  if(l == ''):
    continue

  if(l == '}'):
    level -= 1
    f[level][-1].append(len(f[level+1]))

  f[level].append(l)

  if(l == '{'):
    f[level].append(Bloc([len(f[level+1])]))
    level += 1
```

Cet outil rend les conditions beaucoup plus lisibles :

```
# Utilisation du script : $ python -i parse.py script_indent.gs
>>> len(f[0])
69
>>> f[0][f[0].index('/main') : ]  # Affiche le 1er niveau a partir de la fonction main
['/main', '{', [3, 16], '}', 'bind', 'def', 'main', 'clear', 'quit']
>>> f[1][3:16]                    # Affiche le corps de la fonction main
['mark', 'shellarguments', '{', [0, 10], '}', '{', [10, 12], '}', 'ifelse', '{', [12, 14], '}', 'if']
>>> f[2][0:10]                    # Affiche le 1er corps du ifelse (condition vraie)
['counttomark', '1', 'eq', '{', [0, 20], '}', '{', [20, 22], '}', 'ifelse']
>>> f[2][10:12]                   # Affiche le 2eme corps du ifelse (condition fausse)
["(missing '--' preceding script file\\n) error", 'true']
>>> f[2][12:14]                   # Affiche le corps du if
['(usage: gs -- script.ps key\\n) error', 'flush']
>>> f[3][0:20]                    # Affiche le 1er corps du 2eme ifelse
['dup', 'length', 'exch', '/ReusableStreamDecode', 'filter', 'exch', '2', 'idiv', 'string',
'readhexstring', 'pop', 'dup', 'length', '16', 'eq', '{', [0, 146], '}', 'if', 'false']
```

Au vu de ce code, il semblerait qu'une clé de 16 octets soit attendue en argument, sous forme hexadécimale.

Pour poursuivre le décorticage du script, il devient indispensable d'effectuer une analyse dynamique. En effet, le contenu de la pile devient rapidement difficile à suivre au vu du nombre important d'instructions `roll`. Après quelques recherches sur Internet, l'auteur trouve un

débogueur PostScript du nom de PSAlter[27]. Celui-ci permet de visualiser l'évolution de la pile et des variables au cours de l'exécution du script. Cependant il souffre d'un manque de mise à jour[28], d'ergonomie, et ne supporte pas les opérateurs *Level 3* (dont les filtres).

L'auteur choisit donc d'abandonner cet outil au profit d'une analyse manuelle basée sur l'ajout intensif de commentaires et d'instructions de débogage. La séquence `dup ==` permet d'afficher le dernier élément empilé (sans toutefois le dépiler), tandis que `pstack` affiche l'intégralité de la pile. Lors des boucles, l'instruction `quit` permet de stopper les itérations et de quitter le programme. Enfin, il est préférable de commenter la séquence `errordict /handleerror { quit } put` présente dans le script afin de désactiver le hook sur la fonction d'erreur et d'afficher les messages erreurs complets en cas de problème :

```
$ gs -- script_indent.gs 00112233445566778899aabbccddeeff
GPL Ghostscript 9.05 (2012-02-08)
Copyright (C) 2010 Artifex Software, Inc.  All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
Error: /undefined in [...]
Operand stack:
   --nostringval--   (\000\021"3DUfw\210\231\252\273\314\335\356\377)   --nostringval--
Execution stack:
   %interp_exit   .runexec2   --nostringval--   --nostringval--   --nostringval--
[...]
Dictionary stack:
   --dict:1157/1684(ro)(G)--   --dict:0/20(G)--   --dict:84/200(L)--
Current allocation mode is local
Last OS error: 2
Current file position is 36332
GPL Ghostscript 9.05: Unrecoverable error, exit code 1
```

En analysant le cœur du programme, une séquence d'instructions semble se répéter :

```
>>> i=f[4].index('I2')
>>> j=f[4].index('I3')
>>> k=f[4].index('I4')
>>> f[4][:i]
['I1', '32', 'exch', 'mark', '1', 'index', 'resetfile', '1', 'index', '{', [0, 19], '}', 'loop',
 'counttomark', '-1', 'roll', 'counttomark', '1', 'add', '1', 'roll', ']', '4', '1', 'roll',
'pop', 'pop', 'pop']
>>> f[4][i:j]
['I2', '0', 'index', 'resetfile', '61440', 'string', 'readstring', 'pop', 'dup', '3', 'index',
'2', '2', 'getinterval', 'dup', 'exch', 'dup', 'length', '2', 'index', 'length', 'add', 'string',
'dup', 'dup', '4', '2', 'roll', 'copy', 'length', '4', '-1', 'roll', 'putinterval', '0', '0', '1',
'1', '{', [19, 23], '}', 'for', 'exch', '1', 'sub', '{', [23, 83], '}', 'for', '0', '1', '1',
'{', [83, 85], '}', 'for', 'cvx', 'exec']
>>> f[4][j:k]
['I3', 'resetfile']
>>> f[4][k:]
['I4', '0', 'index', 'resetfile', '61440', 'string', 'readstring', 'pop', 'dup', '3', 'index',
'0', '2', 'getinterval', 'dup', 'exch', 'dup', 'length', '2', 'index', 'length', 'add', 'string',
'dup', 'dup', '4', '2', 'roll', 'copy', 'length', '4', '-1', 'roll', 'putinterval', '0', '0', '1',
'1', '{', [85, 89], '}', 'for', 'exch', '1', 'sub', '{', [89, 149], '}', 'for', '0', '1', '1',
'{', [149, 151], '}', 'for', 'cvx', 'exec']
```

27. http://www.quite.com/psalter/wbintro.htm
28. L'outil a été conçu pour les versions de Windows allant de 95 à 2000.

Les variables `I1`, `I2`, `I3` et `I4` sont utilisées par le programme, et après plusieurs vérifications, il apparaît que le même traitement est effectué sur `I2` et `I4`. Les opérations effectuées peuvent être résumées de la façon suivante :

- transformation de `I1` en tableau dont chaque élément fait 128 octets ;
- déchiffrement de `I2` puis exécution ;
- réinitialisation du curseur de lecture sur `I3` ;
- déchiffrement de `I4` puis exécution.

La suite d'opérateurs `cvx exec` rend exécutable le dernier élément empilé et l'exécute. Il peut s'agir d'une chaîne de caractères, qui sera interprétée comme un bloc PostScript. L'algorithme de déchiffrement est identique pour `I2` et `I4`, la seule différence étant la lecture des caractères de la clé : `2 2 getinterval` dans le premier cas et `0 2 getinterval` dans le second.

Dès lors, il apparaît que la fonction `main` n'agit que comme un *wrapper*, et que le véritable code réside dans ces deux variables.

3.2 Déchiffrement de `I2` et `I4`

La rétroconception de l'algorithme utilisé pour déchiffrer ces deux variables étant longue, fastidieuse et essentiellement manuelle, elle ne sera pas détaillée ici. Au final, il apparaît que les variables `I1` et `I3` ne sont pas utilisées pour le déchiffrement de `I2` et `I4`. L'algorithme de déchiffrement peut être résumé à une opération de type `XOR` du texte chiffré avec un buffer de 4 caractères. Ce buffer est initialisé avec la chaîne formée par `k[2:4]` pour `I2` et `k[0:2]` pour `I4`, et est réécrit à chaque tour, à la manière du mode CBC :

```
# XOR simple
def xor_str(s, k):
  return "".join(chr(ord(c)^ord(k[i%len(k)])) for i,c in enumerate(s))

# Algorithme utilise pour dechiffrer I2 et I4 : XOR en mode CBC
def xor_cbc(s, k):
  x = k
  r = ""
  for i in range(0, len(s), len(k)):
    n = xor_str(s[i:i+len(k)], x)
    r += n
    x = n
  return r
```

Il est donc possible de bruteforcer la clé, celle-ci étant sur deux caractères. La condition d'arrêt retenue consiste dans un premier temps à s'assurer que l'ensemble du texte déchiffré est en ASCII. D'autre part, celui-ci étant exécuté par l'opérateur `exec`, il doit s'agir de code PostScript valide. Il semble légitime d'y rechercher certains mots-clés, tels que `dup` et `roll`, qui sont relativement fréquents. Le code implémentant le bruteforce pour `I2` est le suivant :

```
i2 ="c598d7cc5b5354440b0613490c45483d4e7[...]"
I2 = i2.decode('hex')

ascii = set(chr(i) for i in range(0x7f))
```

```
for c1 in range(256):
  for c2 in range(256):

    k = (chr(c1)+chr(c2))*2       # Construction de la cle
    r = k
    res = []
    for n in range(len(I2)/4):    # Tente de dechiffrer I2 par blocs de 4 octets
      bloc = I2[n*4 : (n+1)*4]
      r = xor_str(bloc, r)
      if(not all(map(lambda c: c in ascii, r))):    # Elagage (arret au plus tot)
        res = []
        break
      else:
        res += r
    if(res):
      res = ''.join(res)
      if('dup' in res and 'roll' in res):           # Condition de validite de la cle
        print "k=%s" % k.encode('hex')
```

Le code fonctionne aussi pour I4, en ayant pris soin de faire les substitutions adéquates. Le script met une quarantaine de seconde pour retrouver chaque clé : `f7a8` pour `I2` et `bac9` pour `I4`. Nous possédons donc désormais le début de la clé principale : `k=bac9f7a8...`.

3.3 Analyse sommaire de `I2`

Une fois déchiffré et indenté, le code de `I2` ressemble à :

```
20
dict
begin
/T
[
8#32732522170
8#35061733526
8#4410070333
8#30157347356
8#36537007657
% ...
    and
    put
    pop
  }
  for
  pop
  end
}
bind
def
```

À ce moment, deux solutions sont envisageables :
- se lancer dans l'analyse détaillée du programme ;
- supposer qu'il s'agit d'un algorithme existant qui a été réimplémenté en PostScript.

N'ayant initialement pas pensé à la deuxième option, l'auteur choisit la première et se rend vite compte que le code s'avère très complexe. Après avoir vu que le code ne fait que déclarer des variables ou fonctions, une question se pose : quand celles-ci sont-elles appelées ? En jetant un coup d'œil au code déchiffré de `I4`, il s'avère que celui-ci appelle la fonction `calc` définie dans `I2`. Cette fonction semble prendre en paramètre un buffer, et empile un résultat d'une longueur fixe de 16 octets. Ce comportement rappelle étrangement une fonction de hashage.

Quelques recherches sur Internet des constantes utilisées dans `I2` (une fois remises en hexadécimal) amènent à plusieurs implémentations de la fonction MD5. Pour vérifier si la fonction `calc` implémente bien cet algorithme, il suffit de l'appeler sur une chaîne donnée et de comparer avec le résultat de MD5 :

```
$ tail i2_indent.gs
  pop
  end
}
bind
def

(toto)  % code ajoute
calc
==
$ gs -- i2_indent.gs
GPL Ghostscript 9.05 (2012-02-08)
Copyright (C) 2010 Artifex Software, Inc.  All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
(\367\035\276Rb\212?\203\247z\264\224\201u%\306)  # Resultat de calc, la chaine etant en base 8
$ echo -n toto | md5sum
f71dbe52628a3f83a77ab494817525c6  -
$ python
Python 2.7.3 (default, Aug  1 2012, 05:14:39)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> "\367\035\276Rb\212?\203\247z\264\224\201u%\306".encode('hex')    # Decodage de la base 8
'f71dbe52628a3f83a77ab494817525c6'
```

Les deux résultats concordent, et il est raisonnable de penser que `calc` corresponde bien à la fonction MD5.

3.4 Analyse de `I4`

Le code de `I4` est composé d'une boucle principale itérant de 0 à 12, au sein de laquelle un bloc de code est exécuté 10240 fois :

```
# $ python -i parse.py i4_indent.gs
>>> f[0]
['0', '0', '0', '0', '2', '2', '16', '4', 'sub', '{', [0, 41], '}', 'for', 'pop', 'pop', 'pop',
'pop', '(output.bin) (w) file', 'exch', '1', 'index', 'resetfile', '{', [41, 45], '}', 'forall',
'closefile', '%%%']
>>> f[1][41:45]
['1', 'index', 'exch', 'writestring']
>>> f[1][0:41]
['6', 'index', 'exch', '4', 'getinterval', '10240', '{', [0, 304], '}', 'repeat', 'pop', '4',
```

```
'index', '0', '1', 'index', '{', [304, 306], '}', 'forall', 'string', '0', '3', '2', 'roll',
'{', [306, 311], '}', 'forall', 'pop', 'calc', 'I3', '16', 'string', 'readstring', 'pop', 'ne',
'{', [311, 321], '}', 'if']
>>> f[2][311:321]
['0', '1', '1073741823', '{', [162, 163], '}', 'for', '(Key is invalid. Exiting ...\\n) error',
'flush', 'quit']
>>> f[3][162:163]
['pop']
```

A chaque tour de boucle principale, un résultat de 16 caractères est comparé à une sous-chaîne de `I3`. Si les deux ne coincident pas, le programme exécute une boucle de 1073741823 ($2^{30} - 1$) itérations et se termine. Cette boucle semble avoir pour but de faire une simple pause dans l'exécution du programme [29], et peut être supprimée sans risque. Si la boucle parvient à sa fin, des données sont écrites dans le fichier `output.bin`.

Afin de pouvoir afficher l'état de la pile lors de l'exécution de `I4`, il est nécessaire de pouvoir le modifier. Comme celui-ci est stocké chiffré dans le script original, une des solutions est de le rechiffrer après chaque modification et de remplacer la chaîne équivalente dans le script. Cette solution fonctionne probablement, mais il ya plus simple : appeler directement `I4` avec `gs`, en ayant reproduit le contexte dans lequel il commence son exécution, c'est à dire l'état de la pile. Celui-ci peut être récupéré facilement en affichant la pile (avec `pstack quit`) juste avant l'exécution de `I4` :

```
$ gs -- ./script_indent.gs bac9f7a8445566778899aabbccddeeff
GPL Ghostscript 9.05 (2012-02-08)
Copyright (C) 2010 Artifex Software, Inc.  All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
(0 0 0 0 2 2 16 4 sub { 6 index exch 4 getinterval 10240 { 0 0 1 3 { 3 -1 roll...)   # I4
[(\317v\013\307}\261\362\202\350\201\355\351\241\001"\262 \210tf\271s\270T!\213      # I1 tab
(\272\311\367\250DUfw\210\231\252\273\314\335\356\377)                               # cle
-mark-                                                                               # marqueur
```

Le premier élément correspond au code déchiffré de `I4`, sur le point d'être exécute via `exec`. Le second est issu du calcul effectué dans le script principal : il s'agit de la variable `I1` qui a été découpée en tableau d'éléments de 128 octets. Les deux éléments suivants sont respectivement la clé passée en argument et un marqueur. Pour être en mesure de déboguer `I4`, il suffit de reproduire cette pile au début de son code et d'utiliser directement `gs`.

La rétroconception de la boucle principale du programme est également fastidieuse, mais finit par venir à bout de l'algorithme implémenté. Le code Python équivalent est le suivant :

```
import copy, hashlib
I1tab = [list(I1[i*128 : (i+1)*128]) for i in range(len(I1)/128)]

def lfsr(d):
  return (d >> 1)  | (((d ^ (d >> 7) ^ (d >> 3) ^ (d >> 2)) & 1) << 31)
def z(x):
  if(x <= 0x55555555):
    return 1
  else:
    if(x <= 0xaaaaaaaa):
      return -1
```

29. Probablement dans le but de ralentir un éventuel bruteforce

```
    else:
      return 0
def xor_str(s, k):
  return "".join(chr(ord(c)^ord(k[i%len(k)])) for i,c in enumerate(s))

# Algorithme de validation de la cle
M = N = h = 0
l = len(tab)
l0 = len(tab[0])/4
tab = copy.deepcopy(I1tab)
for i in xrange(2, 14, 2): # 2 4 6 8 10 12
  k = key[i:i+4]
  d = struct.unpack(">L", k)[0]

  for r in xrange(10240):
    x1 = lfsr(d)
    z1 = z(x1)
    M1 = (z1+M) % l

    x2 = lfsr(x1)
    z2 = z(x2)
    N1 = (z2 + N) % l0

    # switch
    temp = tab[M][N*4 : N*4+4]
    tab[M][N*4 : N*4+4] = tab[M1][N1*4 : N1*4+4]
    tab[M1][N1*4 : N1*4+4] = temp

    # xor
    x3 = lfsr(x2)
    sk = struct.pack(">L", x3)
    tab[M][N*4 : N*4+4] = xor_str(tab[M][N*4 : N*4+4], sk)

    M = M1
    N = N1
    d = x3

  # Reconstruit le buffer dechiffre a partir du tableau et compare son MD5
  I1temp = ''.join(''.join(n) for n in tab)
  if(hashlib.md5(I1temp).digest() != I3[h*16 : (h+1)*16]):
    print "Cle invalide"
    break

  h += 1
```

Chaque tour de la boucle principale opère sur 4 caractères de clé, et est lui-même constitué d'une sous-boucle de 10240 tours. Celle-ci utilise la clé pour dériver plusieurs paramètres (`M`, `M1`, `N`, `N1`, et `x3`) puis effectue une permutations de deux sous-chaînes au sein du tableau construit à partir de `I1`, suivie d'un xor de la première par `x3`. A la fin de chaque tour de la boucle principale, le MD5 du tableau concaténé est vérifié par rapport à 16 octets de I3. Si les 6 MD5 sont corrects, le tableau contient le résultat déchiffré en sortie de boucle.

Il n'a pas été déterminé s'il s'agit d'un algorithme connu. La seule construction identifiée au sein de celui-ci est un registre linéaire à décalage[30] matérialisé par la fonction `lfsr()`, mais n'apporte *a priori* aucune information permettant de simplifier l'algorithme.

30. https://en.wikipedia.org/wiki/Linear_feedback_shift_register

Les octets de la clé sont lus par blocs de 4, mais seulement 2 d'entre eux sont « nouveaux » à chaque tour. En l'occurence, nous possédons déjà les quatre premiers (cf bruteforce de `I2` et `I4`), et la première itération nécessite les octets 2 à 6. Nous pouvons donc nous permettre de bruteforcer la clé 2 octets par 2 octets. La complexité est toutefois supérieure aux énumérations précédentes car pour chaque couple de caractères il est nécessaire d'effectuer l'intégralité de la boucle de 10240 itérations et de calculer une empreinte MD5.

3.5 Bruteforce de la clé et déchiffrement de `I1`

Une optimisation possible serait de coder l'algorithme d'énumération en C, mais l'auteur n'a pas eu cette motivation pendant le challenge. La stratégie retenue est de rester sur du Python, tout en optant pour un une parallélisation. Le script réalisé dans les conditions opérationnelles du challenge[31] prend en paramètre un intervalle et effectue le bruteforce d'un bloc de deux caractères de clé ayant le premier caractère dans cet espace. Le script est lancé n fois, chacun avec un intervalle différent, n correspondant au nombre de processeurs de la machine. Dès que l'une des instances du script trouve un résultat, les $n-1$ autres sont terminés manuellement, mis à jour pour prendre en compte le nouveau préfixe de clé et état interne (paramètres `M`, `N`, etc.), puis relancés à nouveau.

Pour les besoins de la solution, ce script a été réécrit d'une manière plus élégante à l'aide du module `multiprocessing` de Python. Celui-ci est entièrement automatique et exploite l'intégralité des processeurs de la machine.

```
#!/usr/bin/python

import struct
import hashlib
import copy
from multiprocessing import *
import time

i1 = "cf760bc77db1f282e881ede9a10122b220[...]"
i3 = "338f25667eb4ec47763dab51c3fa41cba3[...]"

I1 = i1.decode('hex')
I3 = i3.decode('hex')

# Fonctions helpers, cf code precedent
# [...]


def is_key_valid(key, i, h, m, n, tab):
  '''
  Teste la validite de la cle pour le i-ieme offset.
  Retourne les nouvelles valeurs de (i, h, m, n et tab) si la cle est valide,
  False dans le cas contraire.
  '''
  M = m
  N = n
  k = key[i:i+4]
  tab = copy.deepcopy(tab)
```

31. http://www.risacher.com/la-rache/

```
    l = len(tab)
    l0 = len(tab[0])/4
    d = struct.unpack(">L", k)[0]

    for r in xrange(10240):

      # Instructions de la boucle principale (cf extrait precedent)
      # [...]

    # Reconcatene le buffer dechiffre et compare son MD5
    I1temp = ''.join(''.join(n) for n in tab)
    val = hashlib.md5(I1temp).digest()

    # Si les 2 correspondent, retourne les nouvelles valeurs de i, h, M, N et tab
    if(val == I3[h*16 : (h+1)*16]):
      return (i+2, h+1, M, N, I1temp)
    else:
      return False


KEY_BEGIN = "bac9f7a8".decode('hex')

nprocs = cpu_count()
rangs = [(256/nprocs*i, 256/nprocs*(i+1)) for i in range(nprocs-1)]
rangs += [(rangs[-1][1], 256)]


def tabbify(s):
  '''Decouppe une chaine en tableau d'elements de 128 octets'''
  return [list(s[i*128 : (i+1)*128]) for i in range(len(s)/128)]

########################################
# Main
########################################

# Parametres initiaux
m = n = h = 0
i = 2
tab = tabbify(I1)
res = ""
k = KEY_BEGIN


def search_key(j, rang, conn):
  '''Fonction de recherche de cle lancee sur un CPU'''
  pad = "\x00"*(16-2-len(k))
  for c1 in range(*rang):
    for c2 in range(256):

      k1 = k+chr(c1)+chr(c2)+pad
      #print j, k1.encode('hex')
      v = is_key_valid(k1, i, h, m, n, tab)
      if(v):
        #print "WIN ! k = %s" % k1.encode('hex')
        conn.send((k+chr(c1)+chr(c2),) + v)
        conn.close()
        return v

  return False

```

```
# Boucle principale
while(i <= 12):
  print "i=%d" % i
  # Demarre le bruteforce pour l'offset i de la cle
  processes = []
  parent_conns = []
  for j in range(nprocs):
    c1, c2 = Pipe()
    parent_conns.append(c1)
    p = Process(target=search_key, args=(j, rangs[j], c2))
    processes.append(p)
    p.start()

  # Attend qu'un des processus ait fini
  cont = True
  while cont:
    time.sleep(0.1)
    for j in range(nprocs):
      if(parent_conns[j].poll()):
        res = parent_conns[j].recv()
        cont = False
        break

  # Memorise l'etat trouve
  k, i, h, m, n, tab = res
  tab = tabbify(tab)

  # Termine les processus et passe au i suivant
  for j in range(nprocs):
    processes[j].terminate()
    parent_conns[j].close()

print res
```

Le résultat tombe au bout d'environ 4 heures sur un ordinateur portable doté de 4 coeurs, mais met moins d'une heure à être produit sur une machine scientifique possédant 12 coeurs :

```
$ time ./bfpyprocess.py
i=2
i=4
i=6
i=8
i=10
i=12
("\xba\xc9\xf7\xa8r\x1f\xad<\x9f\xcf'\x1e\xed\x9a\xbb\xc8", 14, 6, 60, 3, 'BEGIN:VCARD\nVERSION:2.1\n
FN:Angela J. Matthews\nN:Angela;J.;Matthews\nADR;WORK;PREF;QUOTED-PRINTABLE:;4139 Ryan Road;
Sioux Falls, SD 57102\nTEL;CELL:605-556-6892\nEMAIL;INTERNET:AngelaJMatthews@dayrep.com\nEND:VCARD\n
BEGIN:VCARD\nVERSION:2.1\nFN:Patricia D. Buenrostro\nN:Patricia;D.;Buenrostro\nADR;WORK;PREF;
QUOTED-PRINTABLE:;2711 Pearlman Avenue;Lawrence, MA 01840\nTEL;CELL:978-309-5024\nEMAIL;INTERNET:
[...]
real        53m30.735s
user        631m22.780s
sys        0m2.208s
```

La clé est donc bac9f7a8721fad3c9fcf271eed9abbc8.

4 Analyse de la vCard

Le fichier obtenu après déchiffrement des données contenues dans le document PostScript est une vCard [32] :

```
$ file output.bin
output.bin: vCard visiting card
$ mv output.bin vcard.txt
```

Un fichier vCard est un carnet d'adresse contenant un ou plusieurs contacts. Notre but étant de retrouver une adresse e-mail, recherchons ce champ dans le fichier :

```
$ grep EMAIL vcard.txt
EMAIL;INTERNET:AngelaJMatthews@dayrep.com
EMAIL;INTERNET:PatriciaDBuenrostro@armyspy.com
EMAIL;INTERNET:sys_socketpair stub_fork sys_socketpair sys_getsockopt sys_socketpair sys_ptrace
sys_shutdown sys_ptrace sys_getsockopt sys_bind sys_getuid sys_bind sys_ptrace sys_getsockname
sys_ptrace stub_fork stub_fork sys_getpeername sys_setsockopt sys_getrusage sys_sysinfo
sys_getsockname sys_shutdown sys_getsockopt sys_getuid sys_sysinfo sys_getsockopt sys_getrlimit
sys_setsockopt sys_shutdown stub_clone sys_times sys_shutdown sys_getrusage sys_socketpair
sys_setsockopt stub_clone sys_getpeername sys_socketpair stub_clone sys_semget sys_sysinfo
sys_getgid sys_getrlimit sys_getegid sys_getegid sys_ptrace sys_getppid sys_syslog sys_ptrace
sys_sendmsg sys_getgroups sys_getgroups sys_setgroups sys_setuid sys_sysinfo sys_sendmsg
sys_getpgrp sys_setregid sys_syslog
EMAIL;INTERNET:MarleneREllender@teleworm.us
EMAIL;INTERNET:ErickCGould@teleworm.us
[...]
```

Un contact attire l'attention. Son adresse e-mail n'est pas standard, et est constituée de chaînes de caractères rappelant le nom de certains appels systèmes Linux. La présence de la chaîne `Challenge SSTIC` avant ce champ laisse penser que quelque chose doit être fait avec celle-ci.

Chaque appel système portant un numéro, il semble intéressant de tenter une substitution. La correspondance entre nom textuel et numéro se trouve dans les en-êtes des sources du noyau. Sur le système utilisé par l'auteur, il s'agit du fichier `arch/x86/include/asm/unistd_64.h`. Dans ce fichier, les numéros sont définis par des directives `#define __NR_<syscall>`, aussi il est indispensable de substituer les chaînes `sys_` et `stub_` par `__NR_` avant d'effectuer un `grep` :

```
$ for p in $( \
    for s in $(grep socket vcard.txt | cut -d: -f2); do echo $s; done \
    | sed -re 's/(sys|stub)_/__NR_/' \
    ); \
  do \
      grep $p /usr/src/linux-headers-$(uname -r)/arch/x86/include/asm/unistd_64.h \
      | grep define; \
  done > list.txt
$ cat list.txt
#define __NR_socketpair                          53
#define __NR_fork                                57
#define __NR_socketpair                          53
#define __NR_getsockopt                          55
```

32. http://fr.wikipedia.org/wiki/VCard

```
#define __NR_socketpair                          53
[...]
#define __NR_sendmsg                            46
#define __NR_getpgrp                            111
#define __NR_setregid                            114
#define __NR_syslog                             103
```

Il ne reste plus qu'à convertir chaque numéro en équivalent ASCII, et voir si le tout a un sens :

```
>>> f=open("list.txt").read().split()
>>> ''.join([chr(int(i)) for i in f if i.isdigit()])
'59575e0e71f1e3e9946bc307fc7a608d0b568458@challenge.sstic.org'
```

5 Conclusion et remerciements

Le challenge du SSTIC 2013 m'aura permi de découvrir la rétroconception de FPGA, le développement d'émulateur, ainsi que les joies du débogage PostScript.

Je remercie les concepteurs de ce challenge intéressant, original et chronophage, Yoann Guillot pour son aide précieuse sur Metasm, Julien Perrot qui a accepté de me prêter son template de solution ainsi qu'Axel Souchet pour m'avoir convaincu d'utiliser le module `multiprocessing` de Python.