Challenge SSTIC 2013

Solution par Mathieu GASPARD

Table des matières

1 Disclaimer....3
2 Introduction 3
3 Partie 1 : PCAP....3
4 Partie 2 : FPGA....6
4.1 Analyse des unités....9
4.1.1 U : ALU....9
4.1.2 ACCU : accumulateur....10
4.1.3 r : ensemble de registres....11
4.1.4 s_m_p : mémoire des instructions....12
4.1.5 s_m_d : mémoire du programme....12
4.1.6 ip : pointeur d'instruction....12
4.2 Analyse du programme....13
5 Partie 3 : PostScript ou comment PS m'a tuer....16
5.1 Script général....16
5.2 Déchiffrement de I2....18
5.3 Déchiffrement et analyse de I4....22
6 Partie 3.5 : vCard....25
7 Annexes....26
7.1 Partie 1 : Déchiffrement fichier sstic.tar.gz-chiffre....26
7.2 Partie 2 : Désassembleur pour instructions du FPGA....30
7.3 Partie 2 : Désassemblage de l'ensemble des instructions....32
7.4 Partie 3 : fichier postscript obtenu....36
7.5 Partie 3 : fichier postscript indenté....36
7.6 Partie 3 : I2 déchiffré....42
7.7 Partie 3 : I4 déchiffré....43

1 Disclaimer

La résolution de ce challenge a été effectuée en utilisant la méthode « la RACHE ©» (http://www.la-rache.com/) et se veut donc naturellement la plus clair et complète possible.

Aucun python n'a été blessé durant la rédaction de cette solution, mais les rétines des lecteurs pourraient être impactées par le code présenté dans ce document.

2 Introduction

Dixit le site du challenge, *le défi consiste à analyser une trace réseau. L'objectif est d'y retrouver une adresse e-mail (...@challenge.sstic.org).*

On télécharge le fichier et on commence l'analyse.

3 Partie 1 : PCAP

Le commande *file* sur le fichier nous indique que c'est un fichier pcap. On l'ouvre avec tcpdump ou wireshark et on découvre les flux suivants (seules les données client->serveur sont présentes) :

- une connexion sur le port TCP 1234 avec envoi des instructions
- 65 paquets ICMP echo request (ping)
- une connexion FTP (identifiant sstic/sstic) avec dépôt du fichier sstic.tar.gz-chiffre
- une connexion TCP port haut (FTP passif) contenant les données de sstic.tar.gz-chiffre

Les instructions 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.
```

On extrait également le contenu chiffré (*follow TCP stream* puis *save as*) et on le *de-base64* dans le fichier *payload.bin*.

A partir des instructions et de l'analyse du fichier PCAP, on en déduit facilement que la clé est transmise dans les paquets ICMP echo request.

8	2.013228	192.168.1.13	192.168.1.12	ICMP	98 Echo (ping) request	id=0x1333, seq=1/256, ttl=40
9	2.013045	192.168.1.13	192.168.1.12	ICMP	98 Echo (ping) request	id=0x2433, seq=1/256, ttl=30
10	2.013620	192.168.1.13	192.168.1.12	ICMP	98 Echo (ping) request	id=0x3533, seq=1/256, ttl=20
11	2.013407	192.168.1.13	192.168.1.12	ICMP	98 Echo (ping) request	id=0x4633, seq=1/256, ttl=10
12	1.013347	192.168.1.13	192.168.1.12	ICMP	98 Echo (ping) request	id=0x5533, seq=1/256, ttl=30

Paquets ICMP transmis

En analysant plus finement les paquets transmis, on se rend compte que peu de données intéressantes diffèrent d'un paquet sur l'autre :

- le champ *TTL* (Time To Live) : **10, 20, 30, 40**
- le champ *DSF* (Differenciated Service Field) : **2** ou **4**
- le champ *Timestamp from ICMP data*
- le champ *ICMP ID*

```
⊞ Frame 7: 98 bytes on wire (784 bits), 98 bytes captured (784 bits)
⊞ Ethernet II, Src: 3com_08:fa:cb (00:01:02:08:fa:cb), Dst: AsustekC_4e:ce:db (90:e6:ba:4e:ce:db)
⊟ Internet Protocol Version 4, Src: 192.168.1.13 (192.168.1.13), Dst: 192.168.1.12 (192.168.1.12)
    Version: 4
    Header length: 20 bytes
  ⊞ Differentiated Services Field: 0x02 (DSCP 0x00: Default; ECN: 0x02: ECT(0) (ECN-Capable Transport))
    Total Length: 84
    Identification: 0x0000 (0)
  ⊞ Flags: 0x02 (Don't Fragment)
    Time to live: 30
    Protocol: ICMP (1)
  ⊞ Header checksum: 0xd93d [correct]
    Source: 192.168.1.13 (192.168.1.13)
    Destination: 192.168.1.12 (192.168.1.12)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
⊟ Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x7c9f [correct]
    Identifier (BE): 563 (0x0233)
    Identifier (LE): 13058 (0x3302)
    Sequence number (BE): 1 (0x0001)
    Sequence number (LE): 256 (0x0100)
    Timestamp from icmp data: Mar 18, 2013 12:35:36.187452000 Paris, Madrid
    [Timestamp from icmp data (relative): 0.000009000 seconds]
  ⊞ Data (48 bytes)
```

Champs variant d'un paquet à l'autre

- Le champ *TTL* a 4 valeurs possibles (10, 20, 30, 40) et peut donc représenter 2 bits de clé (non temporel).

- Le champ *DSF* a 2 valeurs possibles (2,4) et peut donc représenter 1 bit de clé (non temporel).

- La différence entre 2 paquets a 2 valeurs possibles (1, 2) et peut donc représenter 1 bit de clé (temporel) :

> Les champs *Timestamp from ICMP data* et *ICMP ID* vont fournir la même information :
>
> il va falloir récupérer la différence entre le paquet X+1 et X.
>
> Par exemple, chaque paquet est envoyé avec une pause de 1 ou 2 secondes entre chaque
>
> (*Timestamp from ICMP data X+1 - Timestamp from ICMP data X == 1 ou 2*).

Cette information peut être récupérée dans Wireshark directement en modifiant l'affichage du timestamp de chaque paquet.

De même pour l'ID ICMP (en little endian),

ICMP ID X+1 - ICMP ID X == 0x11 ou 0x9

On se rend compte que quand le temps d'envoi entre 2 paquet est de 2 secondes, la différence d'ICMP ID est de 0x11 , et de 0x9 quand le temps d'envoi est de 1 seconde.

La même « information » est donc obtenue via les *timestamp* ou les *ICMP ID*.

Chacun des 64 paquets fournit donc 4 bits de clé, on peut donc récupérer une clé AES de 256 bits.

Toutes les permutations des bits de clés (*timestamp*|*DSF*|*TTL* , *TTL*|*DSF*,*timestamp*....) doivent être testées. Le fait de posséder un condensat des données déchiffrées est un réel plus pour pouvoir automatiser le déchiffrement et le test du clair obtenu.

AES étant un algorithme de chiffrement par bloc (on suppose que le mode CBC est utilisé), la taille des données à chiffrer doit être un multiple de la taille d'un bloc (16 octets pour AES). Le standard PKCS7 ,couramment utilisé, indique que X octets de valeur X sont ajoutés aux données en clair afin de compléter le dernier bloc si la taille des données à chiffrer n'est pas un multiple de la taile d'un bloc.

On peut **naïvement** tester la présence de padding en testant la valeur du dernier octet des données déchiffrées : si celui ci est > 0 et < 16, on enlève X octets des données déchiffrées.

On calcule ensuite le condensat des données sans le padding et on teste par rapport au condensat de référence. On déchiffre finalement l'archive ***sstic.tar.gz*** (cf code en annexe 7.1) .

4 Partie 2 : FPGA

Le fichier *sstic.tar.gz* contient un répertoire *archive* contenant 4 fichiers :

- data
- decrypt.py
- s.ngr
- smp.py

Le fichier *data* contient des données (chiffrées)

Le fichier *smp.py* contient un tableau (*smp*) de 231 octets.

Le fichier *decrypt.py* est le suivant :

```
#!/usr/bin/python

import sys
import base64
import md5

import dev
import smp

if len(sys.argv) != 2:
  print("usage: %s key" % sys.argv[0])
  sys.exit(1)

key = int(sys.argv[1], 16)
key = [(key >> (i * 8)) & 0xff for i in range(16)]

result = []
d = open("data", "rb").read()
dev.init("sp.ngr")
for i in range(0, len(d), 224):
  smd = d[i : (i + 224)]
  smd = (key, len(smd), smd)
  dev.send_smd(smd)
  dev.send_smp(smp.smp)
  dev.start()
  dev.wait_finished()
  result = result + dev.get_data()

print "".join(result)
result_md5 = md5.new()
result_md5.update("".join(result))
result_md5 = result_md5.digest()
result_md5 = [ord(x) for x in result_md5]
target_md5 = "6c0708b3cf6e32cbae4236bdea062979"
target_md5 = [int(target_md5[x:(x + 2)], 16) for x in range(0, len(target_md5),
2)]
print(["%02x" % x for x in target_md5])
print(["%02x" % x for x in result_md5])
if result_md5 != target_md5:
  print("Bad key...")
  sys.exit(1)

result = base64.b64decode("".join(result))
d = open("atad", "wb")
d.write(result)
d.close()
sys.exit(0)
```

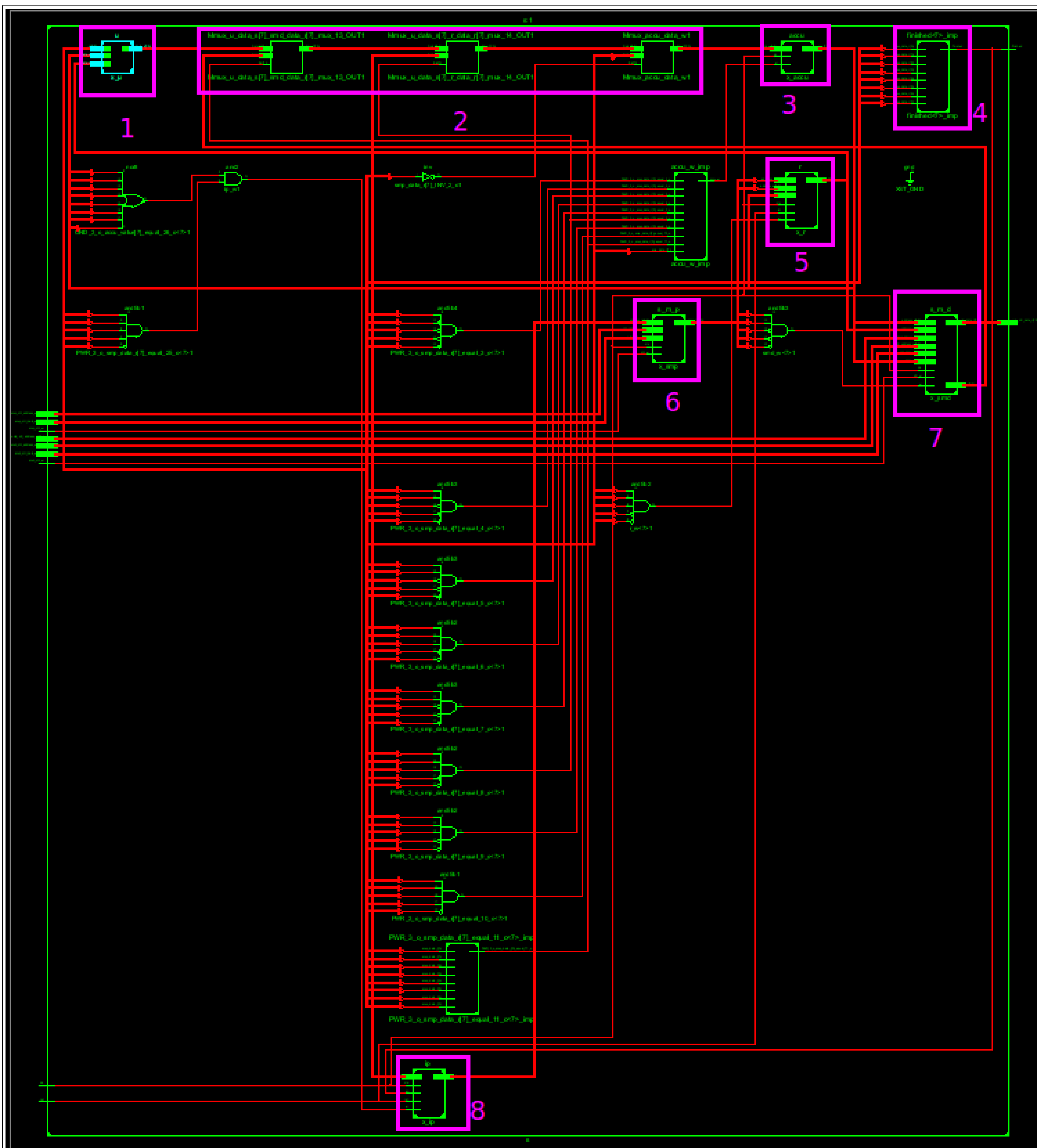
Le fichier *s.ngr* est apparemment un fichier texte dont voici les 2 premières lignes :

```
XILINX-XDB 0.1 STUB 0.1 ASCII
XILINX-XDM V1.6e
```

Une recherche rapide sur Internet nous apprend que ce fichier est un fichier de *netlist* généré par l'outil « Xilinx® Synthesis Technology (XST) » de la société Xilinx. Le fichier de *netlist* est une représentation graphique d'un système (ici un FPGA) pour afficher les différents composants logiques (portes AND, OR, composants ADD, MUX...).

On télécharge donc la toolchain complète de Xilinx pour pouvoir afficher notre *netlist* et on rage allègrement pendant les (nombreuses) heures requises pour télécharger le client de 8.5 Go à 100ko/s (merci le bridage côté Xilinx...).

On peut donc (enfin) ouvrir notre *netlist* et découvrir ceci :



Fichier de netlist s.ngr

On prend une grande inspiration, on « WTF is this », on *rage*, on *quit*, on re-*rage*, on re-*quit*, on re-re-*rage* et on s'y remet.

Après analyse, on découvre que le fichier *decrypt.py* va :

- lire une clé de 16 octets sur la ligne de commande (hex encodée)
- « initialiser » un FPGA avec l'architecture décrite dans *s.ngr*
- traiter les données du fichier *data* par bloc de 224 octets :
  - lire 224 octets du fichier *data* (ou moins pour le dernier bloc)
  - envoyer au FPGA via *send_smd* : *cle_16_octets|longueur_bloc|data_bloc*, 241 octets dans le cas classique (16 + 1 + 224)
  - envoyer au FPGA via *send_smp* le contenu de *smp.smp* (231 octets qui ne varient pas)
  - lire des données en réponse
- Une fois le fichier *data* complètement traité, on calcule le MD5 des données récupérées et on les compare à une valeur de référence
- Si le MD5 correspond, on *de-base64* les données récupérées et on inscrit le résultat dans le fichier *atad*

On en déduit donc que les données récupérées lors du traitement de chaque bloc de 224 octets sont encodées en base64 et donc dans une liste restreinte de caractères.

Intéressons-nous maintenant au FPGA lui-même. Il peut se décomposer en différentes unités :

1. **u** : ALU (*Arithmetic and Logical Unit*)
2. suite de MUX pour sélection de données à insérer dans ACCU
3. **ACCU** : registre spécial servant d'accumulateur (résultat de calculs...)
4. **finished** : passe un flag à 1 pour finir le programme si sa valeur entrée == 200
5. **r** : ensemble de 8 registres (appelés ici r0 à r7)
6. **s_m_p** : mémoire des instructions (chargé avec smp.smp dans *decrypt.py*)
7. **s_m_d** : mémoire des données (chargé avec 241 octets dans *decrypt.py*)
8. **ip** : fournit la valeur du pointeur d'instruction (EIP sur x86) à aller chercher dans **s_m_p**
9. différentes portes AND avec sortie à 1 si entrée == valeur particulière.

En analysant plus finement chaque composant, on se rend compte que la sortie de **s_m_p** (sur 1 octet) est réinjectée dans différents composants (**u, finished, un des MUX servant à initialiser ACCU, …**).

On comprend que le tableau *smp.smp*, initialisant l'état interne de **s_m_p**, représente les instructions qui vont être exécutées par ce FPGA et qu'il va falloir les décoder pour comprendre le code exécuté. Les différentes portes AND (non encadrées dans l'image du FPGA ou à l'intérieur de l'ALU **u**) vont permettre de connaître la valeur de sortie de **s_m_p** (l'instruction) et d'activer certaines unités du FPGA (selon l'instruction).

Les bus entre les différents composants permettent de traiter 1 octet à chaque fois (la sortie d'une unité est sur 1 octet maximum).

De manière générale, seuls les 5 bits de poids fort de la sortie de **s_m_p** sont utilisés pour déterminer l'instruction. Dans ces cas là, les 3 bits de poids faible sont utilisés comme adresse.

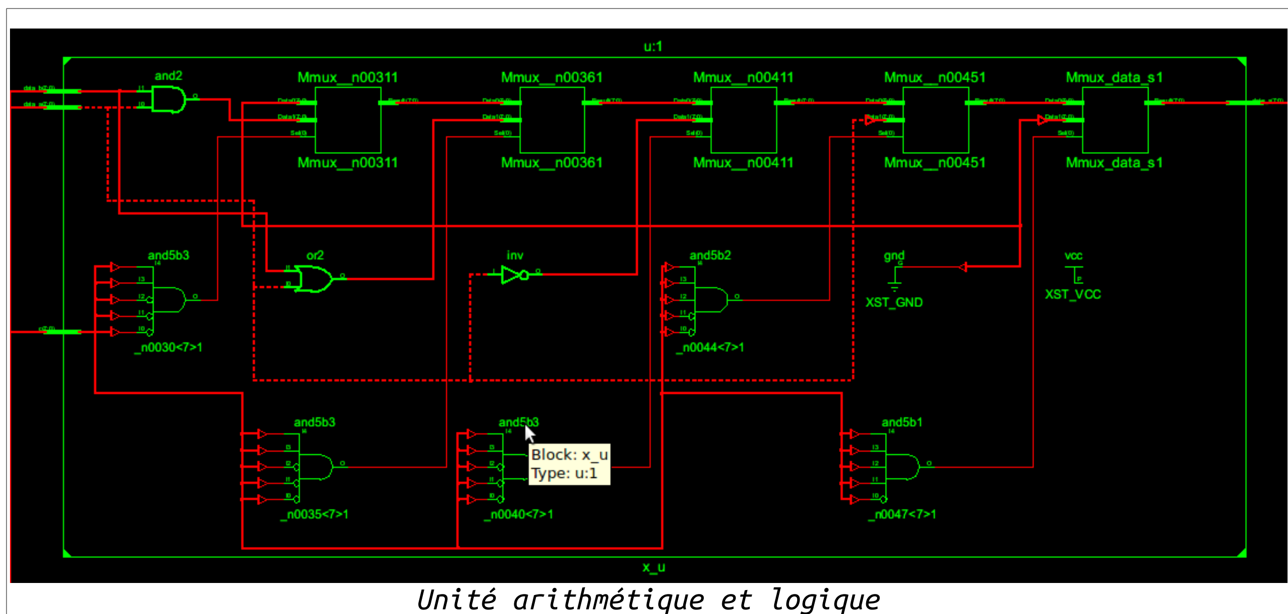
Par exemple, lors de l'accès à un registre dans l'unité **r**, les 3 bits de poids faible vont déterminer quel est le registre accédé en lecture ou écriture (3 bits = 8 valeurs = r0 à r7).

4.1 Analyse des unités

4.1.1 U : ALU

L'unité arithmétique et logique (ici **u**) va permettre d'effectuer différentes opérations sur des valeurs (présentées dans l'ordre des Mmux) :

- ET logique
- OU logique
- NOT
- OR 128 (mise à 1 du bit de poids fort)
- SHL 1 (shift left de 1 bit)



Unité arithmétique et logique

Les données en entrée sont soit l'accumulateur (**ACCU**), soit un registre de **r** (r0 à r7).

Les instructions fournies par l'ALU sont donc les suivantes :

- AND ACCU, registre
- AND ACCU, registre
- NOT ACCU
- SHL ACCU, 1
- OR ACCU, 128

L'ACCU ne pouvant être initialisé que avec une constante inférieure à 128 (voir partie sur **ACCU**), l'instruction **OR ACCU, 128** permet d'initialiser ACCU avec des valeurs entre 128 et 255, par exemple :

```
MOV ACCU, 40
OR ACCU, 128
```

==> ACCU == 168

4.1.2 ACCU : accumulateur

La valeur en entrée de l'accumulateur (**ACCU**) provient d'une suite de MUX.

Un MUX (multiplexer) est un composant à plusieurs entrées et avec un sélecteur pour choisir laquelle des entrée est répliquée sur la sortie. Par exemple, si un MUX possède un sélecteur sur 1 bit, il possédera 2 entrées : si le sélecteur est à 0, la première entrée sera recopiée sur la sortie, si le sélecteur est à 1, la deuxième entrée sera recopiée sur la sortie.

Les valeurs des MUX peuvent être :

1. résultat de l'ALU
2. données en sortie de **s_m_d** (données de la mémoire du programme)
3. données en sortie de **r** (donc valeur d'un des registres r0 à r7)
4. données en sortie de **s_m_p** (instruction) si instruction < 128

Seule la valeur d'un MUX sera sélectionnée selon la valeur de sortie de **s_m_p** (l'instruction courante).

On peut donc décoder les instructions suivantes :

1. instructions de l'ALU (voir paragraphe précédent)
2. MOV ACCU, SMD[ACCU]
3. MOV ACCU, registre
4. MOV ACCU, constante (avec constante < 128)

4.1.3 r : ensemble de registres

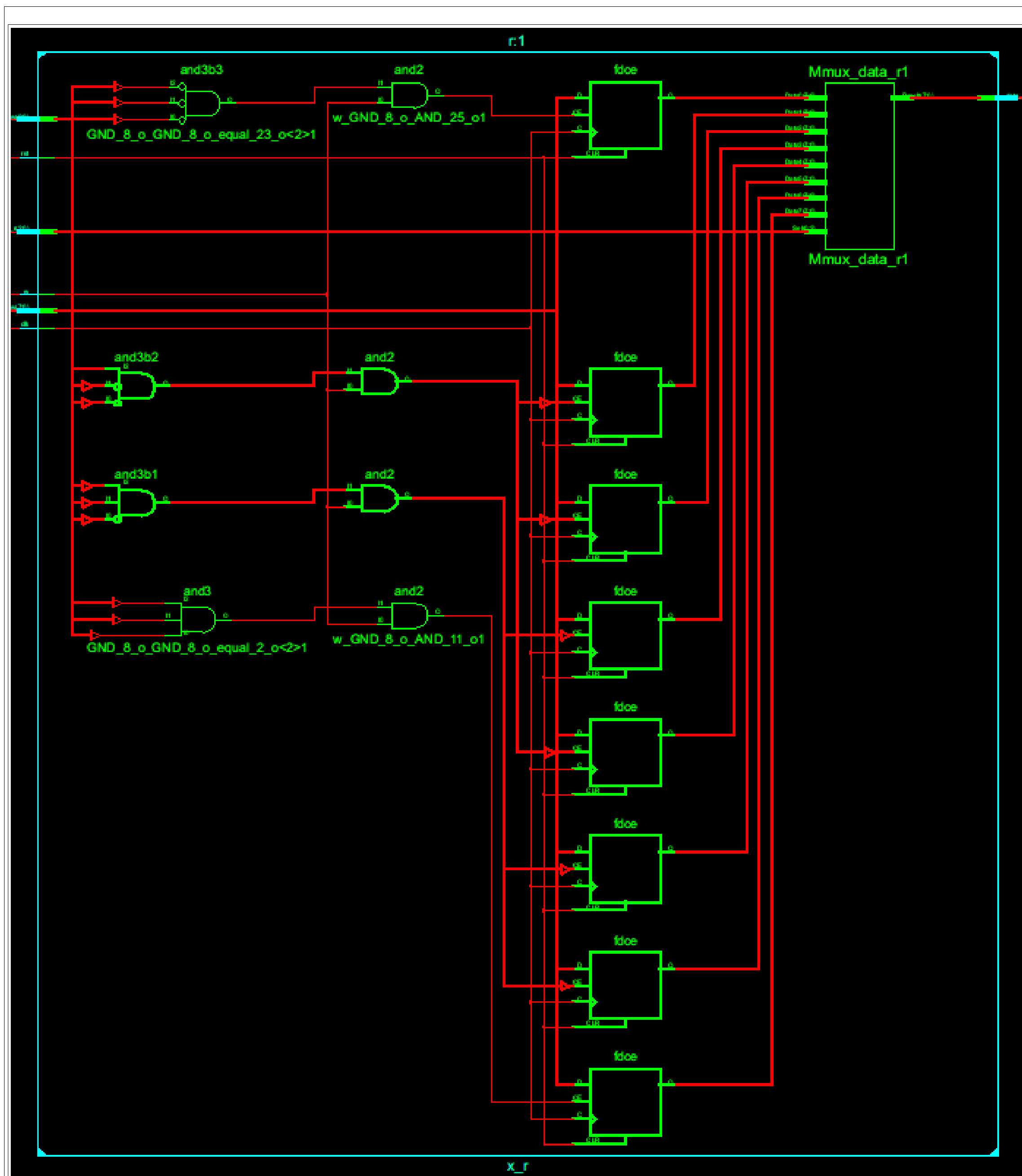
L'unité **r** représente un ensemble de 8 registres (ici appelés r0 à r7).

Lors de la lecture ou de l'écriture d'un registre, le registre sélectionné est défini par les 3 bits de poids faible de l'instruction (sortie de **s_m_p**).

En cas d'écriture, la donnée à écrire est contenue dans **ACCU**.

On peut donc décoder l'instruction suivante :

- MOV registre, ACCU



*Unité **r** : registres r0 à r7 sélectionnés via le MUX Mmux_data_r1*

4.1.4 s_m_p : mémoire des instructions

L'unité **s_m_p** contient les instructions que le programme va exécuter (un peu comme la section *.text* d'un binaire PE). Elle est initialisée avec les données de *smp.smp* (contenus dans *smp.py*).

L'instruction à exécuter est définie par l'adresse en entrée de l'unité (ici **address_r**) qui est la sortie de l'unité **ip**. L'instruction produite en sortie est **s_m_p[ip]**.

4.1.5 s_m_d : mémoire du programme

L'unité **s_m_d** contient la mémoire du programme en cours. Elle est initialisée, à chaque lancement/traitement du FPGA, avec les 241 octets définis dans *decrypt.py* :

- 16 octets de clé
- 1 octet de taille
- 224 octets du fichier *data* (ou moins pour le dernier bloc de *data*)

Lors de la lecture de données, l'adresse à laquelle lire est contenue dans ACCU.

Lors de l'écriture de données, les données à écrire sont contenues dans ACCU et l'adresse d'écriture est contenue dans un registre.

On peut donc décoder les instructions suivantes :

- MOV SMD[registre], ACCU
- MOV ACCU, SMD[ACCU]

4.1.6 ip : pointeur d'instruction

L'unité **ip** va fournir l'adresse de l'instruction à exécuter dans **s_m_p**. Sa valeur commence à 0 et s'incrémente de 1 à chaque tour d'horloge (clock tick).

La notion de « saut » est également gérée via l'instruction 184 : lorsque l'instruction courante (les 5 bits de poids fort pour être précis) est égale à 184 et que la valeur de ACCU vaut 0, IP est initialisé à la valeur contenue dans un registre (le choix du registre est fait par les 3 bits de poids faible de l'instruction).

On peut donc décoder l'instruction suivante :

- if ACCU == 0, set IP = registre

ip : pointeur d'instruction

4.2 Analyse du programme

A partir de l'analyse précédente, un désassembleur a été développé pour comprendre le code de *smp.smp*. Le code source du désassembleur est disponible en annexe 7.2. Il génère une trace pour une exécution « normale » du programme avec une fausse clé de 16 octets.

Le code désassemblé est également disponible en annexe 7.3 mais ne comporte que toutes les instructions désassemblées individuellement et non une trace des instructions exécutées lors d'une exécution complète.

Une analyse manuelle de la trace « complète » d'exécution (faisant plusieurs milliers de lignes) permet de comprendre le fonctionnement du programme, défini ci-après en pseudo-C :

```
for(i=0 ; i<len(data) ; i++) {
      t = data[i] ^ key[i%16]
      q = calc(t)
      data[i] = q
}
char calc(val)  {
      res = 0
      for(i=0 ; i<8 ; i++) {
            res <<= 1
            tt = (val>>i)%2
            if (tt == 1)
                  res |= 1
      }
      return (res & 0xFF)
}
```

Chaque octet de *data* (224 dans le cas général, moins pour le dernier bloc) va être *xoré* avec l'octet de clé en position i%16.

Il est à noter que le FPGA n'implémente pas de porte logique XOR, tout est réalisé avec des portes AND, OR et NOT :

data[i] = (data[i] | key[i%16]) & (~(data[i] & key[i%16])) = data[i] ^ key[i%16]

Puis le résultat va être passé dans une fonction (ici appelée *calc*) qui va inverser le sens de lecture de l'octet (le msb devient le lsb..). Par exemple :

bin(42) : 00101010

calc(bin(42)) : 01010100

Après traitement des 224 octets de *data*, les nouvelles valeurs (**calc(data[i]^key[i%16])**) vont être récupérées depuis le FPGA dans le script *decrypt.py*.

Chaque bloc de données à traiter fait 224 octets. Chaque caractère de la clé va servir à déchiffrer 14 caractères (224/16).

```
Data : a b c d e f g h i j k l m n o p q r s t u v w x y z...
key :  0 1 2 3 4 5 6 7 8 9 a b c d e f 0 1 2 3 4 5 6 7 8 9...
```

Comme l'on sait, via le script *decrypt.py*, que les caractères déchiffrés sont du *base64*, on va pouvoir bruteforcer chaque caractère de la clé.

On validera, pour les 255 octets possible à chaque position de la clé, que **calc(data[i] ^ octet_testé)** appartient au charset *base64*. Si il n'appartient pas au charset, on ne le sélectionne pas comme octet potentiel de clé. Il est à noter qu'il est nécessaire d'ajouter le caractère « **\n** » au *charset* pour pouvoir trouver la clé.

Bruteforce de la clé :

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

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

test_keys = {}
for i in range(16):
	test_keys[i] = [j for j in range(256)]

charset = [ord(i) for i in 
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=\n"]

def calc(val):
	res = 0
	for i in range(8):
		res <<= 1
		tt = (val>>i)%2
		if tt == 1:
			res |= 1
	return (res&0xff)

for i in range(0, len(d), 224):
	smd = d[i : (i + 224)]
	for j in range(16): #len key
		for k in range(j, len(smd), 16):
			t = ord(smd[k])
			test_key_tmp = test_keys[j]
			for l in test_key_tmp :
				if calc(l^t) not in charset: # not in charset, remove
					test_keys[j].remove(l)

for o in range(16):
	print "%d - %s"%(o, test_keys[o])
```

A la fin de l'exécution du script, il ne reste qu'un caractère possible pour chaque position de la clé :

```
$ ./bruteforce.py
0 - [230]
1 - [131]
2 - [220]
3 - [188]
4 - [22]
5 - [239]
6 - [88]
7 - [198]
8 - [101]
9 - [172]
10 - [35]
11 - [211]
12 - [30]
13 - [109]
14 - [161]
15 – [37]
```

On déchiffre les données de data et on vérifie le fichier obtenu :

```
#!/usr/bin/env python
key = [230,131,220,188,22,239,88,198,101,172,35,211,30,109,161,37]
d = open("data", "rb").read()
def calc(val):
      res = 0
      for i in range(8):
            res <<= 1
            tt = (val>>i)%2
            if tt == 1:
                  res |= 1
      return (res&0xff)

result = []
for i in range(0, len(d), 224):
      smd = d[i : (i + 224)]
      for j in range(len(smd)):
            char = ord(smd[j])
            result.append(chr(calc(char^key[j%16])))

open("atad","w").write("".join(result))

$ md5sum atad
c0708b3cf6e32cbae4236bdea062979  atad
```

On trouve bien le bon fichier avec le bon MD5.

On de-base64 le fichier et on trouve un fichier postscript.

5 Partie 3 : PostScript ou comment PS m'a tuer

5.1 *Script général*

Après déchiffrement et décodage, on obtient donc un script postscript (reconnaissable aux mots clé) **non indenté** (cf annexe 7.4, les variables **I1** à **I4** ont été abrégées pour ne pas prendre trop de place).

On essaie de le lancer en utilisant un interpréteur postscript : ghostscript :

```
$  gs  -q script.ps
missing '--' preceding script file
usage: gs -- script.ps key

$ gs  -q -- script.ps
no key provided
usage: gs -- script.ps key
```

On voit que l'on doit fournir une clé en paramètre au script. On commence alors l'analyse du script (indentation, cf annexe 7.5, apprentissage du postscript...).

On se rend vite compte que la clé fournie sur la ligne de commande doit faire 16 octets et doit être *hex-encoded* (donc 32 caractères sur la ligne de commande) :

```
put /main
{
	mark shellarguments % comme [
	{
		counttomark 1 eq % pour argument 1
		{
			dup
			length
			exch %%% stack : -mark- 16 (cle) <== top
			/ReusableStreamDecode filter
			exch %%%%  -mark- -file-  32
			2 idiv %%%  divise par 2 la longueur de la cle
			string readhexstring % clé dans string
			pop %%%%%%%% enleve true ou false
			dup %%%%%%% duplique cle pour length
			length 16 eq %%%%%%%%% longueur de cle == 16 ou pas
			{
				I1 32 exch
				mark
				…
```

On considère dans la suite de ce document que la clé saisie est « **00112233445566778899aabbccddeeff** ».

Si la clé fournie fait bien 16 octets, le script PS va ensuite lire les données de **I1** par blocs de 128 octets :

```
mark
      1 index
      resetfile
      1 index
      {
            counttomark 1 sub
            index %% index 0
            counttomark 2 add %%%% 4
            index
            4 mul %% 128
            string
            readstring %%% lit les 128 premiers octets de I1
                        pop %%% pop True
            dup %%% duplique la chaine de 128 pour test avec equal
            () eq {pop exit} if %%%% si chaine vide, exit loop
     } loop %%%% lit I1 par block de 128 octets (77 blocs de 128 bytes (154 en
hex encoded))
       counttomark -1 roll %%% met le dernier block en 1er (-file-)
       counttomark 1 add %%% ... -file- 79
       1 roll %%% -mark-  (cle) 32 -file- -file- -mark- data
 ]
```

5.2 Déchiffrement de I2

Puis le script va lire **I2** et le 3e et 4e caractère de la clé de chiffrement et les dupliquer pour en faire une clé de chiffrement de 4 octets. La clé utilisée ici est donc **"3"3** ("22332233".decode("hex") en python)

```
4 1 roll %%% -mark- (cle) [array avec 77 block de data I1] 32 -file- -file-
pop %%% enleve -file-
pop %%% enleve -file-
pop %%% enleve -32-
I2 %%%% -mark- (cle) [array avec 77 block de data I1] -file I2-
0 index %%% -copy file I2 en haut de stack-
resetfile %%% consome -file I2-
61440 string readstring %%%% lit 61440 octets de I2 dans string, seulement 2888 hex decoded disponibles
pop %%% pop False
dup %% duplique la data de I2
3 index %%% copy cle en top stack
2 2 getinterval %%% prend 2e et 3e caractere de la clé
dup %%% duplique ces 2 caracteres
exch %%% echange les memes données (WTF?)
dup %% copie une 3e fois les 2 caracteres
length %% 2
2 index %%% [array I1] - data-I2 - data-I2 - (sous-cle) -  (sous-cle) - 2- (sous-cle)
length add %%% (sous-cle) -  (sous-cle) - 4
string %%% (\000\000\000\000) ???
dup dup %%% (\000\000\000\000) * 3
4 2 roll %%%  - (\000\000\000\000) - (sous-cle) - (\000\000\000\000)
copy %%% ("3) -("3\000\000) - ("3\000\000) - ("3)
length %% 2
4 -1 roll %%%  ("3\000\000) - ("3\000\000) - 2 - ("3)
putinterval %%%% ("3"3)
0
0 1 1
{
     pop %%% pop un des 2 0 au dessus
     2 index %%% ("3"3) - 0 - data-I2
     length
} for %% exécuté 2 fois
%%%% data-I2 - data-I2 - ("3"3) - 0 - 2888 - 4
exch %%%% data-I2 - data-I2 - ("3"3) - 0 - 4 - 2888
1 sub %%%% data-I2 - data-I2 - ("3"3) - 0 - 4 - 2887
```

Le script va ensuite itérer sur les données de **I2** (2888 octets) et les déchiffrer.

Pour ce faire, il va utiliser la clé de 4 octets (**"3"3** dans notre cas) et appliquer un XOR caractère par caractère sur les données de **I2.** En C, cela donnerait :

```
     for(i=0 ; i<2887 ; i+=4) {
          I2[i]   = I2[i] ^ cle[i%4]
          I2[i+1] = I2[i+1] ^ cle[(i+1)%4]
          I2[i+2] = I2[i+2] ^ cle[(i+2)%4]
          I2[i+3] = I2[i+3] ^ cle[(i+3)%4]
     }
```

En postscript, c'est **_un peu_** plus verbeux et **moins** compréhensible :

```
%%%%%% for(i=0; i<2887; i+=4) %%%%%%
%%%% consomme 4 et 2887
%%% stack : (cle) - [array I1] - data-I2 - data-I2 - ("3"3) - 0
{
     3
     copy %%% duplique data-I2 - ("3"3) - 0
     exch %%%  data-I2 - 0 - ("3"3)
     length %%% data-I2 - 0 - 4
     getinterval %%% prend les 4 premiers caracteres de data-I2
     2 index %%% data-I2 - ("3"3) - 0 - 4premiers_char_I2 - ("3"3)
     mark
          3 1 roll %%% data-I2 - ("3"3) - 0 - mark - 4premiers_char_I2 - ("3"3)
          0 1
          3 -1 roll %%% data-I2 - ("3"3) - 0 - mark - 4premiers_char_I2 - 0 - 1 - ("3"3)
          dup
          length %%% data-I2 - ("3"3) - 0 - mark - 4premiers_char_I2 - 0 - 1 - ("3"3) - 4
          1 sub
          exch %%% data-I2 - ("3"3) - 0 - mark - 4premiers_char_I2 - 0 - 1 - 3 - ("3"3)
          4 1 roll %%% data-I2 - ("3"3) - 0 - mark - 4premiers_char_I2 - ("3"3) - 0 - 1 - 3
          %%% for (j=0; j<3; j++)
          {
               dup %%% 0 - mark - 4premiers_char_I2 - ("3"3) - 0 - 0
               3 2 roll %%%  0 - mark - 4premiers_char_I2  - 0 - 0 - ("3"3)
               dup %%% 0 - mark - 4premiers_char_I2  - 0 - 0 - ("3"3) - ("3"3)
               5 1 roll %%%  0 - mark - ("3"3)  - 4premiers_char_I2  - 0 - 0 - ("3"3)
               exch %%%  0 - mark - ("3"3)  - 4premiers_char_I2  - 0 - ("3"3) - 0
               get %%%%  0 - mark - ("3"3)  - 4premiers_char_I2  - 0 - " (char de la clé)
               3 1 roll %%%%  0 - mark - ("3"3)  - " - 4premiers_char_I2  - 0
               exch %%%% 0 - mark - ("3"3)  - " - 0 - 4premiers_char_I2
               dup %%%%  0 - mark - ("3"3)  - " - 0 - 4premiers_char_I2 - 4premiers_char_I2
```

```
                 5 1 roll %%%%0 - mark - 4premiers_char_I2 - ("3"3)  - " - 0 -
4premiers_char_I2
                 exch %%%% 0 - mark - 4premiers_char_I2 - ("3"3)  - " -
4premiers_char_I2 - 0
                 get %%%% data-I2 - ("3"3) - 0 - mark - 4premiers_char_I2 -
("3"3)  - " - 1e_char_I2
                 xor %%% real xor
                 3 1 roll %%% data-I2 - ("3"3) - 0 - mark - XORed_char-
4premiers_char_I2  - ("3"3)
           } for
           %%%% XORed_char_1 - XORed_char_2 - XORed_char_3 - XORed_char_4 -
4premiers_char_I2  - ("3"3)
           pop pop %%% enleve ("3"3) et 4premiers_chars_I2
      ]
      dup %%%% ("3"3) - 0 - [array 4 char XORed] - [array 4 char XORed]
      length %%%% ("3"3) - 0 - [array 4 char XORed] - 4
      string %%% (\000\000\000\000)
      0
      3 -1 roll %%%% ("3"3) - 0 -  (\000\000\000\000) - 0 - [array 4 char XORed]
      {
           3 -1 roll
           dup
           4 1 roll
           exch
           2 index
           exch
           put
           1 add
      } forall %%% met les 4 bytes dans une string
      pop %%%  ("3"3) - 0 - string_4_xored
      4 -1 roll %%%% ("3"3) - 0 - string_4_xored - data-I2
      dup %%%% ("3"3) - 0 - string_4_xored - data-I2 - data-I2
      5 1 roll %%%% data-I2 - data-I2 - ("3"3) - 0 - string_4_xored - data-I2
      3 1 roll %%%% data-I2 - data-I2 - ("3"3) - data-I2 - 0 - string_4_xored
      dup %%%% data-I2 - data-I2 - ("3"3) - data-I2 - 0 - string_4_xored -
string_4_xored
      4 1 roll %%%% data-I2 - data-I2 - ("3"3) - string_4_xored - data-I2 - 0 -
string_4_xored
      putinterval %%% reecrit data-I2 avec string_4_xored
      exch %%%% data-I2 - data-I2 - string_4_xored - ("3"3)
      pop
} for
```

Une fois **I2** déchiffré sur la pile d'appel de postscript, cette même pile est rendue exécutable et le contenue déchiffré de **I2** est exécuté :

```
%%%%% I2 déchiffré sur la stack à ce moment là
cvx  %%% rend la pile exécutable
exec %%% exécute I2
```

On va donc devoir bruteforcer les 2 octets de la clé permettant de déchiffrer **I2**. **I2** étant du postscript, on part du principe que le *charset* utilisé est : ***\n + tous les caractères entre 32 et 127***.

Tout comme dans la partie FPGA, on va devoir tester si un caractère de clé est conforme aux contraintes, *i.e* **I2[i] ^ cle[i%2]** est dans *charset*.

On bruteforce donc les 2 caractères de la clé :

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

I2 = 'c598d7cc5b...'.decode("hex")

charset = [ord("\n"), ord("\r")]
charset.extend([i for i in range(32,127)])

liste_1 = []
liste_2 = []
liste_3 = []
liste_4 = []
for i in range(0, len(I2), 4):
      liste_1.append(ord(I2[i]))
      liste_2.append(ord(I2[i+1]))
      liste_3.append(ord(I2[i+2]))
      liste_4.append(ord(I2[i+3]))

test_keys = [j for j in range(256)]

for curr_list in [liste_1, liste_3, liste_2, liste_4]:
      final_key = []
      final_key.extend(test_keys)
      for i in test_keys:
            current = i
            for j in curr_list:
                  t = current ^ j
                  if t not in charset:
                        final_key.remove(i)
                        break
                  current = t
      print final_key
```

```
$ python bruteforce_I2.py
[226, 227, 232, 242, 244, 246, 247, 249, 255]
[247]
[160, 166, 168, 169, 171, 173, 183, 188, 189]
[160, 166, 168, 169, 171, 173, 183, 188, 189]
```

Les 2 premières lignes sont les caractères possibles pour le 3e caractère de la clé, les 2 lignes suivantes sont les caractères possibles pour le 4e caractère de la clé.

Le 3e caractère est forcément **247**, on teste toutes les possibilité pour le 4e et on trouve que c'est **168.**

On peut donc déchiffrer **I2** et on connaît 2 caractères de la clé :

clé : xx xx **f7 a8** xx xx xx xx xx xx xx xx xx xx xx xx

Le contenu de **I2** déchiffré est disponible en annexe 7.6.

Le code de **I2** ne va faire que définir une fonction ***calc*** qui va effectuer des traitement (à base d'énormément de xor, bitshift, and...). Seule une analyse sommaire de son fonctionnement a été réalisée.

5.3 Déchiffrement et analyse de I4

On continu l'analyse du fichier principal et on se rend compte que, de manière identique à **I2**, **I4** est déchiffré puis exécuté depuis la stack.

Les octets de la clé utilisés pour déchiffrer sont les 2 premiers cette fois ci. On bruteforce de nouveau la clé de la même manière que pour **I2** et on obtient les 2 premiers octets de la clé :

clé : ba c9 f7 a8 xx xx xx xx xx xx xx xx xx xx xx xx

Le code de **I4** déchiffré est disponible en annexe 7.7.

L'analyse de **I4** n'a été réalisée que de façon sommaire. Il en ressort que le code va utiliser 6 fois différents morceaux de la clé successivement pour déchiffrer **I1**.

1e tour : ba c9 **f7 a8 xx xx** xx xx xx xx xx xx xx xx xx xx

2e tour : ba c9 f7 a8 **xx xx xx xx** xx xx xx xx xx xx xx xx

3e tour : ba c9 f7 a8 xx xx **xx xx xx xx** xx xx xx xx xx xx

…

Ces 6 itérations vont permettre d'utiliser toute la clé.

A chaque tour/itération, après avoir déchiffré **I1** pour ce tour, la fonction ***calc,*** définie dans **I2**, est appelée et le résultat (une chaine de 16 octets) est comparée à 16 octets de **I3 (**qui sert de référence).

Le résultat de ***calc*** au 1e tour doit être égal aux 16 premiers octets de **I3**, au 2e tour doit être égal aux octets 16-32 de **I3**...

Chaque itération utilise des « paramètres » qui seront modifiés entre chaque itération :

- 2 nombres (utilisés 2 fois)
- **I1** qui va être déchiffré au fur et à mesure

On peut plus ou moins exprimer **I4** avec l'algorithme suivant

```
nb1 = nb2 = nb3 = nb4 = 0
encrypted = I1_chiffré
for(i=2 ; i<12 ; i+=2) {
        cle_tmp = cle[i:i+4]
        _do_crypto_stuff_(nb1, nb2, nb3, nb4, encrypted, cle_tmp)
        output =  calc()
        if output != expected_I3 :
                exit
}
fd = open(« output.bin »)
fd.write(contenu_dechiffré)
```

Au 1e tour, la clé utilisée est **f7 a8 xx xx.** Comme on connaît 2 octets sur les 4, on va être en mesure de bruteforcer les 2 octets de clé manquants.

Pour ce faire, un script postscript a été développé pour lire la clé à tester sur la ligne de commande, réaliser les traitements (en reprenant le code de **I4** et **I2** déchiffrés) et valider que la clé entrée est la bonne par rapport au résultat de ***calc*** et à la valeur attendue dans **I3**.

Les clés à tester sont générées via un script python *gen_keys.py* :

```
#!/usr/bin/env python
key = "bac9f7a8445566778899aabbccddeeff"
for i in range(256):
        for j in range(256):
                s = "%s%02x%02x%s"%(key[:28], i, j, key[32:])
                print s
```

Puis elles sont testées sur le script de bruteforce via la commande « ***parallel*** » permettant de lancer plusieurs commandes en parallèle :

```
$ python gen_keys.py | parallel « gs -q --  bruteforce.ps » {}
```

Cette technique a l'avantage de ne pas avoir besoin de *reverser* la partie cryptographique du chiffrement en postscript, il suffit de reprendre le code *verbatim* !

Par contre, elle présente plusieurs inconvénients. Premièrement, c'est **lent** ! Initialement, le bruteforce était réalisé sur 1 seul CPU et le test des 65536 clés prenait environ 4 heures.

Avec l'utilisation de ***parallel*** et d'une machine quadri-coeur, on tombe à 1 heure maximum pour tester 1 itération. Etant donné qu'il y a 6 itérations, il faut donc 6 heures **maximum** pour tout tester.

De plus, il est nécessaire d'injecter les nouveaux paramètres trouvés à chaque itération :

- modifier *gen_keys.py* pour injecter les 2 nouveaux caractères trouvés
- modifier **I1** et les 2 * 2 nombres modifiés au début du tour suivant
- modifier la valeur de référence à la fin pour comparaison (16 caractères de **I3**)

Ces actions nécessitent de modifier manuellement 2 fichiers et n'est donc pas automatisable.

Voici par exemple la partie concernant la définition des 2 * 2 nombres successifs selon les tours et la valeur de référence de **I3** :

```
%% 0 0 0 0 2 incrementer de 2 a chaque fois
%% 75 10 75 10 4
%% 70 26 70 26 6
%% 13 8 13 8 8
%% 26 31 26 31 10
15 23 15 23 12
...
...
%% valeur de reference
%(3\217%f~\264\354Gv=\253Q\303\372A\313)
%%2e valeur de reference
%%(\243\51\341\205\66\270\61\131\263\246\220\240\46\136\305\31)
%% 3e valeur ref
%(\252\351\117\16\161\123\166\304\360\207\274\314\335\13\343\264)
%% 4e valeur ref
% (\241\24\370\276\164\141\102\304\111\170\372\247\155\256\142\317)
%% 5e valeur ref
%(\31\175\173\316\116\263\215\326\214\214\345\366\237\62\156\36)
% 6e valeur ref
(\377\316\256\77\162\370\352\243\216\1\232\131\261\334\11\227)
```

Une fois les 6 itérations effectuées, on possède la clé finale et on peut la fournir au script initial pour lui faire effecteur son traitement normal, *i.e* déchiffrer **I1** et écrire le contenu déchiffré dans le fichier *output.bin*.

```
$ gs -q - script.ps bac9f7a8721fad3c9fcf271eed9abbc8
$ file output.bin
output.bin: vCard visiting card
```

6 Partie 3.5 : vCard

Le fichier *output.bin* est un fichier de carte de visite contenant une entrée pour le challenge sstic :

```
BEGIN:VCARD
VERSION:2.1
FN:Challenge SSTIC
N:Challenge;SSTIC
ADR;WORK;PREF;QUOTED-PRINTABLE:;Campus Beaulieu;Rennes
TEL;CELL:
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
END:VCARD
```

L'adresse e-mail a été remplacée par des numéros de syscall. Une recherche online sur un certain nombre d'entre eux nous indique qu'ils sont définis dans le fichier ***unistd_64.h*** des sources du noyau Linux.

Chaque syscall représente ici le code ASCII d'un caractère de l'adresse e-mail à trouver. On les remplace donc et on obtient l'adresse suivante :

59575e0e71f1e3e9946bc307fc7a608d0b568458@challenge.sstic.org

En conclusion, un challenge assez varié qui m'aura permis de découvrir le monde palpitant (ou pas) des FPGA et ce magnifique langage qu'est le PostScript (les auteurs devraient créer un concours de code postscript obfusqué à la *ioccc*) , merci donc aux auteurs pour les nombreux WTF lancés au cours de celui-ci.

7 Annexes

7.1 Partie 1 : Déchiffrement fichier sstic.tar.gz-chiffre

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

from Crypto.Cipher import AES
import hashlib
import struct
import sys

iv = "76C128D46A6C4B15B43016904BE176AC".decode("hex")
digest = "61c9392f617290642f9a12499de6b688"
decalage = [2, 2, 2, 2, 2, 1, 2, 1, 1, 2, 1, 2, 2, 2, 2, 2, 1, 1, 1, 2, 1, 2,
2, 2, 1, 2, 1, 2, 1, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 1, 2, 1, 1, 2, 2, 2, 1, 2,
1, 2, 1, 2, 2, 2, 2, 2, 1, 2, 1, 1, 2, 2, 2, 1]
dscp = [ 2, 2, 4, 4, 2, 4, 2, 2, 4, 4, 4, 2, 4, 4, 2, 2, 2, 2, 4, 4, 2, 4, 4,
2, 4, 4, 4, 4, 4, 2, 4, 4, 2, 4, 2, 2, 2, 2, 4, 4, 4, 2, 4, 4, 4, 2, 4, 2, 2,
2, 4, 2, 4, 4, 2, 4, 4, 4, 4, 2, 4, 4, 2, 4]
ttl = [30, 30, 40, 30, 20, 10, 30, 30, 10, 20, 20, 40, 10, 10, 20, 10, 20, 10,
30, 20, 10, 10, 30, 30, 40, 20, 30, 10, 20, 40, 20, 40, 10, 20, 30, 40, 10, 10,
30, 30, 40, 20, 40, 20, 30, 30, 40, 30, 30, 10, 20, 20, 10, 10, 30, 40, 20, 10,
20, 10, 10, 20, 40, 30]



def get_decalage(index):
      return (decalage[index] - 1) # 2 = bits 1 , 1 = bits 0
def get_decalage_alter(index):
      return (decalage[index]%2) # 2 = bits 0 , 1 = bits 1
def get_dscp(index):
      return ((dscp[index]/2)-1) # 2 = 0 , 4 = 1
def get_dscp_alter(index):
      if dscp[index] == 2:
            return 1
      else:
            return 0 # 2 = 1, 4 = 0
def get_ttl(index): # 10 = 0, 20 = 1, 30 = 2, 40 = 3
      res = {10:0, 20:1, 30:2, 40:3}
      return res[ttl[index]]
def get_ttl_1(index):
      res = {10:0, 20:1, 30:3, 40:2}
      return res[ttl[index]]
def get_ttl_2(index):
      res = {10:0, 20:2, 30:1, 40:3}
      return res[ttl[index]]
def get_ttl_3(index):
      res = {10:0, 20:2, 30:3, 40:1}
      return res[ttl[index]]
def get_ttl_4(index):
      res = {10:0, 20:3, 30:1, 40:2}
      return res[ttl[index]]
def get_ttl_5(index):
      res = {10:0, 20:3, 30:2, 40:1}
      return res[ttl[index]]
def get_ttl_6(index):
      res = {10:1, 20:0, 30:2, 40:3}
```

```
      return res[ttl[index]]
def get_ttl_7(index):
      res = {10:1, 20:0, 30:3, 40:2}
      return res[ttl[index]]
def get_ttl_8(index):
      res = {10:1, 20:2, 30:0, 40:3}
      return res[ttl[index]]
def get_ttl_9(index):
      res = {10:1, 20:2, 30:3, 40:0}
      return res[ttl[index]]
def get_ttl_10(index):
      res = {10:1, 20:3, 30:0, 40:2}
      return res[ttl[index]]
def get_ttl_11(index):
      res = {10:1, 20:3, 30:2, 40:0}
      return res[ttl[index]]
def get_ttl_12(index):
      res = {10:2, 20:0, 30:1, 40:3}
      return res[ttl[index]]
def get_ttl_13(index):
      res = {10:2, 20:0, 30:3, 40:1}
      return res[ttl[index]]
def get_ttl_14(index):
      res = {10:2, 20:1, 30:0, 40:3}
      return res[ttl[index]]
def get_ttl_15(index):
      res = {10:2, 20:1, 30:3, 40:0}
      return res[ttl[index]]
def get_ttl_16(index):
      res = {10:2, 20:3, 30:0, 40:1}
      return res[ttl[index]]
def get_ttl_17(index):
      res = {10:2, 20:3, 30:1, 40:0}
      return res[ttl[index]]
def get_ttl_18(index):
      res = {10:3, 20:0, 30:1, 40:2}
      return res[ttl[index]]
def get_ttl_19(index):
      res = {10:3, 20:0, 30:2, 40:1}
      return res[ttl[index]]
def get_ttl_20(index):
      res = {10:3, 20:1, 30:0, 40:2}
      return res[ttl[index]]
def get_ttl_21(index):
      res = {10:3, 20:1, 30:2, 40:0}
      return res[ttl[index]]
def get_ttl_22(index):
      res = {10:3, 20:2, 30:0, 40:1}
      return res[ttl[index]]
def get_ttl_23(index):
      res = {10:3, 20:2, 30:1, 40:0}
      return res[ttl[index]]

# time, dscp, ttl, id
def assemble(a, b, c, d):

      result = []
      result.append( (a << 3) | (b << 2) | (c << 1) | d )
      result.append( (a << 3) | (b << 2) | (d << 1) | c )
      result.append( (a << 3) | (d << 2) | (b << 1) | c )
      result.append( (a << 3) | (d << 2) | (c << 1) | b )
      result.append( (a << 3) | (c << 2) | (b << 1) | d )
```

```
    result.append( (a << 3) | (c << 2) | (d << 1) | b )

    result.append( (b << 3) | (c << 2) | (d << 1) | a )
    result.append( (b << 3) | (c << 2) | (a << 1) | d )
    result.append( (b << 3) | (d << 2) | (c << 1) | a )
    result.append( (b << 3) | (d << 2) | (a << 1) | c )
    result.append( (b << 3) | (a << 2) | (d << 1) | c )
    result.append( (b << 3) | (a << 2) | (c << 1) | d )

    result.append( (d << 3) | (b << 2) | (c << 1) | a )
    result.append( (d << 3) | (b << 2) | (a << 1) | c )
    result.append( (d << 3) | (c << 2) | (b << 1) | a )
    result.append( (d << 3) | (c << 2) | (a << 1) | b )
    result.append( (d << 3) | (a << 2) | (c << 1) | b )
    result.append( (d << 3) | (a << 2) | (b << 1) | c )

    result.append( (c << 3) | (b << 2) | (d << 1) | a )
    result.append( (c << 3) | (b << 2) | (a << 1) | d )
    result.append( (c << 3) | (a << 2) | (b << 1) | d )
    result.append( (c << 3) | (a << 2) | (d << 1) | b )
    result.append( (c << 3) | (d << 2) | (a << 1) | b )
    result.append( (c << 3) | (d << 2) | (b << 1) | a )

    return result



tab_ttl = [get_ttl, get_ttl_1, get_ttl_2, get_ttl_3, get_ttl_4, get_ttl_5,
get_ttl_6, get_ttl_7, get_ttl_8, get_ttl_9, get_ttl_10, get_ttl_11, get_ttl_12,
get_ttl_13, get_ttl_14, get_ttl_15, get_ttl_16, get_ttl_17, get_ttl_18,
get_ttl_19, get_ttl_20, get_ttl_21, get_ttl_22, get_ttl_23]
tab_dscp = [get_dscp, get_dscp_alter]
tab_decalage = [get_decalage, get_decalage_alter]


data = open("payload.bin").read()

for t in tab_ttl:
    for d in tab_dscp:
        for g in tab_decalage:
            key_list = [""]*24
            for i in xrange(0, 64):
                curr_ttl = t(i)
                curr_dscp = d(i)
                curr_decal = g(i)
                nibbles_1 = assemble(curr_decal, curr_dscp,
curr_ttl%2, (curr_ttl>>1)%2)
                for mmm in xrange(24):
                    key_list[mmm] += "%x"%nibbles_1[mmm]

            keys = [i.decode("hex") for i in key_list]

            for key in keys:
                if len(key) != 32:
                    print "bad len : %d - %s"%(len(key),
key.encode("hex"))
                crypt = AES.new(key, AES.MODE_CBC, iv)
                crypted = crypt.decrypt(data)
                # check padding
                if ord(crypted[-1]) > 0 and ord(crypted[-1]) <
16 :
                    nb_pad_byte = ord(crypted[-1])
```

```
                                    print "found padding, removing %d
bytes"%nb_pad_byte
                                    crypted = crypted[:-nb_pad_byte]

                              hashed = hashlib.md5(crypted).hexdigest()
                              if hashed == digest:
                                    print "found the file, writing to
sstic.tar.gz"
                                    open("sstic.tar.gz","w").write(crypted)
                                    sys.exit(0)
```

7.2 Partie 2 : Désassembleur pour instructions du FPGA

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

import smp

data = open("data").read()

registers = [0]*8
accu = 0
ip = 0
finish = 0

smd = [i for i in range(40,56)] # fake key
smd.append(224) #len 1 block
smd.extend([ord(i) for i in data[0:224]])

def status():
	r = registers
	return "ACCU:%03d ,IP:%03d ,r0:%03d ,r1:%03d ,r2:%03d ,r3:%03d ,r4:%03d
,r5:%03d ,r6:%03d ,r7:%03d"%(accu, ip, r[0],r[1],r[2],r[3],r[4],r[5],r[6],r[7])


def decode_instr(i):

	global registers, accu, ip, smd

	addr  = i&0x7 # addr dans 3 bits de poids faible
	instr = i&0xF8 # instruction dans 5 bits de poids fort
	#print "---instruction : %d (addr : %d , real_instr : %d)"%( i,addr,instr)
	ip = ip + 1
	if instr < 128:
		accu = i
		return "MOV ACCU, %d"%i
	elif instr == 136:
		accu = accu & registers[addr]
		return "AND ACCU, r%d"%addr
	elif instr == 144:
		accu = accu | registers[addr]
		return "OR ACCU, r%d"%addr
	elif instr == 160:
		accu = (~accu)&0xFF
		return "NOT ACCU"
	elif instr == 168:
		accu = registers[addr]
		return "MOV ACCU, r%d"%addr
	elif instr == 176:
		registers[addr] = accu
		return "MOV r%d, ACCU"%addr
```

```
    elif instr == 184:
        if accu == 0:
            ip = registers[addr]
        return "if accu == 0, set IP = r%d\n"%addr
    elif instr == 192:
        smd[registers[addr]] = accu
        return "MOV SMD[r%d], ACCU"%addr
    elif instr == 200:
        finish = 1
        return "SET FINISH FLAG!!!"
    elif instr == 208:
        accu = smd[accu]
        if accu == 0:
            print "============ read 0 from smd ============="
        return "MOV ACCU, SMD[ACCU]"
    elif instr == 216:
        accu = (accu<<1)&0xFF
        return "SHL ACCU, 1"
    elif instr == 224:
        accu = accu | 128
        return "OR ACCU, 128"
    else :
        return "*"*50


while finish == 0:
    i = smp.smp[ip]
    string = "%d - %s"%(ip, decode_instr(i))
    if len(string) < 40:
        string += " "*(40-len(string))
    if "\n" in string:
        string = string.replace("\n"," ")
        print "%s - %s\n"%(string, status())
    else:
        print "%s - %s"%(string, status())
```

7.3 Partie 2 : Désassemblage de l'ensemble des instructions

```
0  -  MOV ACCU, 0
1  -  MOV r0, ACCU
2  -  MOV ACCU, 16
3  -  MOV ACCU, SMD[ACCU]
4  -  MOV r7, ACCU
5  -  MOV ACCU, r0
6  -  NOT ACCU
7  -  MOV r6, ACCU
8  -  MOV ACCU, 14
9  -  MOV r5, ACCU
10  -  MOV ACCU, 113
11  -  MOV r4, ACCU
12  -  MOV ACCU, 0
13  -  if accu == 0, set IP = r4
14  -  MOV ACCU, 1
15  -  MOV r6, ACCU
16  -  MOV ACCU, 22
17  -  MOV r5, ACCU
18  -  MOV ACCU, 113
19  -  MOV r4, ACCU
20  -  MOV ACCU, 0
21  -  if accu == 0, set IP = r4
22  -  MOV ACCU, 82
23  -  MOV r6, ACCU
24  -  MOV ACCU, r7
25  -  if accu == 0, set IP = r6
26  -  MOV ACCU, 17
27  -  MOV r7, ACCU
28  -  MOV ACCU, r0
29  -  MOV r6, ACCU
30  -  MOV ACCU, 36
31  -  MOV r5, ACCU
32  -  MOV ACCU, 113
33  -  MOV r4, ACCU
34  -  MOV ACCU, 0
35  -  if accu == 0, set IP = r4
36  -  MOV ACCU, r7
37  -  MOV r1, ACCU
38  -  MOV ACCU, r1
39  -  MOV ACCU, SMD[ACCU]
40  -  MOV r6, ACCU
41  -  MOV ACCU, 15
42  -  AND ACCU, r0
43  -  MOV ACCU, SMD[ACCU]
44  -  OR ACCU, r6
45  -  MOV r6, ACCU
46  -  MOV ACCU, r1
47  -  MOV ACCU, SMD[ACCU]
48  -  MOV r7, ACCU
49  -  MOV ACCU, 15
50  -  AND ACCU, r0
51  -  MOV ACCU, SMD[ACCU]
52  -  AND ACCU, r7
53  -  MOV r7, ACCU
54  -  MOV ACCU, r7
55  -  NOT ACCU
56  -  AND ACCU, r6
```

```
57  -  MOV r7, ACCU
58  -  MOV ACCU, 64
59  -  MOV r6, ACCU
60  -  MOV ACCU, 83
61  -  MOV r5, ACCU
62  -  MOV ACCU, 0
63  -  if accu == 0, set IP = r5
64  -  MOV ACCU, r7
65  -  MOV SMD[r1], ACCU
66  -  MOV ACCU, 1
67  -  MOV r6, ACCU
68  -  MOV ACCU, r0
69  -  MOV r7, ACCU
70  -  MOV ACCU, 76
71  -  MOV r5, ACCU
72  -  MOV ACCU, 113
73  -  MOV r4, ACCU
74  -  MOV ACCU, 0
75  -  if accu == 0, set IP = r4
76  -  MOV ACCU, r7
77  -  MOV r0, ACCU
78  -  MOV ACCU, 2
79  -  MOV r6, ACCU
80  -  MOV ACCU, 0
81  -  if accu == 0, set IP = r6
82  -  SET FINISH FLAG!!!
83  -  MOV ACCU, 1
84  -  MOV r5, ACCU
85  -  MOV ACCU, 0
86  -  MOV r4, ACCU
87  -  MOV ACCU, 109
88  -  MOV r3, ACCU
89  -  MOV ACCU, r5
90  -  if accu == 0, set IP = r3
91  -  MOV ACCU, 102
92  -  MOV r3, ACCU
93  -  MOV ACCU, r4
94  -  SHL ACCU, 1
95  -  MOV r4, ACCU
96  -  MOV ACCU, r5
97  -  AND ACCU, r7
98  -  if accu == 0, set IP = r3
99  -  MOV ACCU, 1
100  -  OR ACCU, r4
101  -  MOV r4, ACCU
102  -  MOV ACCU, r5
103  -  SHL ACCU, 1
104  -  MOV r5, ACCU
105  -  MOV ACCU, 87
106  -  MOV r3, ACCU
107  -  MOV ACCU, 0
108  -  if accu == 0, set IP = r3
109  -  MOV ACCU, r4
110  -  MOV r7, ACCU
111  -  MOV ACCU, 0
112  -  if accu == 0, set IP = r6
113  -  MOV ACCU, 0
114  -  MOV r1, ACCU
115  -  MOV r3, ACCU
```

```
116  -  MOV ACCU, 1
117  -  MOV r2, ACCU
118  -  MOV ACCU, 99
119  -  OR ACCU, 128
120  -  MOV r4, ACCU
121  -  MOV ACCU, r2
122  -  if accu == 0, set IP = r4
123  -  MOV ACCU, 44
124  -  OR ACCU, 128
125  -  MOV r4, ACCU
126  -  MOV ACCU, r7
127  -  AND ACCU, r2
128  -  if accu == 0, set IP = r4
129  -  MOV ACCU, 22
130  -  OR ACCU, 128
131  -  MOV r4, ACCU
132  -  MOV ACCU, r6
133  -  AND ACCU, r2
134  -  if accu == 0, set IP = r4
135  -  MOV ACCU, 15
136  -  OR ACCU, 128
137  -  MOV r4, ACCU
138  -  MOV ACCU, r1
139  -  if accu == 0, set IP = r4
140  -  MOV ACCU, r3
141  -  OR ACCU, r2
142  -  MOV r3, ACCU
143  -  MOV ACCU, r2
144  -  MOV r1, ACCU
145  -  MOV ACCU, 89
146  -  OR ACCU, 128
147  -  MOV r4, ACCU
148  -  MOV ACCU, 0
149  -  if accu == 0, set IP = r4
150  -  MOV ACCU, 34
151  -  OR ACCU, 128
152  -  MOV r4, ACCU
153  -  MOV ACCU, r1
154  -  if accu == 0, set IP = r4
155  -  MOV ACCU, r2
156  -  MOV r1, ACCU
157  -  MOV ACCU, 89
158  -  OR ACCU, 128
159  -  MOV r4, ACCU
160  -  MOV ACCU, 0
161  -  if accu == 0, set IP = r4
162  -  MOV ACCU, r3
163  -  OR ACCU, r2
164  -  MOV r3, ACCU
165  -  MOV ACCU, 0
166  -  MOV r1, ACCU
167  -  MOV ACCU, 89
168  -  OR ACCU, 128
169  -  MOV r4, ACCU
170  -  MOV ACCU, 0
171  -  if accu == 0, set IP = r4
172  -  MOV ACCU, 72
173  -  OR ACCU, 128
174  -  MOV r4, ACCU
```

```
175  -  MOV ACCU, r6
176  -  AND ACCU, r2
177  -  if accu == 0, set IP = r4
178  -  MOV ACCU, 62
179  -  OR ACCU, 128
180  -  MOV r4, ACCU
181  -  MOV ACCU, r1
182  -  if accu == 0, set IP = r4
183  -  MOV ACCU, r2
184  -  MOV r1, ACCU
185  -  MOV ACCU, 89
186  -  OR ACCU, 128
187  -  MOV r4, ACCU
188  -  MOV ACCU, 0
189  -  if accu == 0, set IP = r4
190  -  MOV ACCU, r2
191  -  OR ACCU, r3
192  -  MOV r3, ACCU
193  -  MOV ACCU, 0
194  -  MOV r1, ACCU
195  -  MOV ACCU, 89
196  -  OR ACCU, 128
197  -  MOV r4, ACCU
198  -  MOV ACCU, 0
199  -  if accu == 0, set IP = r4
200  -  MOV ACCU, 87
201  -  OR ACCU, 128
202  -  MOV r4, ACCU
203  -  MOV ACCU, r1
204  -  if accu == 0, set IP = r4
205  -  MOV ACCU, r2
206  -  OR ACCU, r3
207  -  MOV r3, ACCU
208  -  MOV ACCU, 0
209  -  MOV r1, ACCU
210  -  MOV ACCU, 89
211  -  OR ACCU, 128
212  -  MOV r4, ACCU
213  -  MOV ACCU, 0
214  -  if accu == 0, set IP = r4
215  -  MOV ACCU, 0
216  -  MOV r1, ACCU
217  -  MOV ACCU, r2
218  -  SHL ACCU, 1
219  -  MOV r2, ACCU
220  -  MOV ACCU, r1
221  -  SHL ACCU, 1
222  -  MOV r1, ACCU
223  -  MOV ACCU, 118
224  -  MOV r4, ACCU
225  -  MOV ACCU, 0
226  -  if accu == 0, set IP = r4
227  -  MOV ACCU, r3
228  -  MOV r7, ACCU
229  -  MOV ACCU, 0
230  -  if accu == 0, set IP = r5
```

7.4 Partie 3 : fichier postscript obtenu

```
/I1 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter
/ReusableStreamDecode filter cf760bc77db1f282e881ede9a10122b2... cafebabe def

/I2 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter
/ReusableStreamDecode filter
c598d7cc5b5354440b0613490c45483d4e7b0f6c0368120f1001070501030202120914031508150
2020305040... cafebabe def

/I3 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter
/ReusableStreamDecode filter 338f25667eb4ec47763dab51c3fa41cba329e1853...
cafebabe def

/I4 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter
/ReusableStreamDecode filter 8ae98ae900000000020002000316121411... cafebabe def

/error { (%stderr)(w) file exch writestring } bind def errordict /handleerror {
quit } put /main { mark shellarguments { counttomark 1 eq { dup length exch
/ReusableStreamDecode filter exch 2 idiv string readhexstring pop dup length 16
eq { I1 32 exch mark 1 index resetfile 1 index { counttomark 1 sub index
counttomark 2 add index 4 mul string readstring pop dup () eq {pop exit} if }
loop counttomark -1 roll counttomark 1 add 1 roll ] 4 1 roll pop pop pop 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 {pop 2 index length} for exch 1 sub { 3 copy exch
length getinterval 2 index mark 3 1 roll 0 1 3 -1 roll dup length 1 sub exch 4
1 roll { dup 3 2 roll dup 5 1 roll exch get 3 1 roll exch dup 5 1 roll exch get
xor 3 1 roll } for pop pop ] 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 I3
resetfile 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 {pop 2 index length} for exch 1 sub { 3
copy exch length getinterval 2 index mark 3 1 roll 0 1 3 -1 roll dup length 1
sub exch 4 1 roll { dup 3 2 roll dup 5 1 roll exch get 3 1 roll exch dup 5 1
roll exch get xor 3 1 roll } for pop pop ] 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
```

7.5 Partie 3 : fichier postscript indenté

```
/I1 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter
/ReusableStreamDecode filter cf760bc77db1f282e881ede9a10122b2208... cafebabe
def

/I2 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter
/ReusableStreamDecode filter c598d7cc5b5354440b0613490c45483d4e7b0... cafebabe
def

/I3 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter
/ReusableStreamDecode filter 338f25667eb4ec47763dab51c3fa41cba329e... cafebabe
def

/I4 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter
/ReusableStreamDecode filter 8ae98ae900000000020002000316121... cafebabe def
```

```
/error { (%stderr)(w) file exch writestring } bind def

errordict /handleerror { quit }

put /main
{
      mark shellarguments % comme [
      {
            counttomark 1 eq % pour argument 1
            {
                  dup
                  length
                  %pstack
                  exch % stack : -mark- 16 (cle) <== top
                  %%%%%%% pstack
                  /ReusableStreamDecode filter
                  exch %  -mark- -file-  16
                  %%%%%% pstack
                  2 idiv % divise par 2 la longueur de la cle

                  string readhexstring % clé dans string
                  %%%%% pstack   -mark- (\252\273\314\335\356\377) true
                  pop %%%%%%%% enleve true ou false
                  dup %%%%%%% duplique cle pour length
                  length 16 eq %%%%%%%%% longueur de cle == 16 ou pas
                  {
                        %%%% pstack
                        I1 32 exch
                        %%%% pstack -mark-
(\000\021"3DUfw\210\231\252\273\314\335\356\377) 32  -file-
                        mark
                              1 index %%% -mark-
(\000\021"3DUfw\210\231\252\273\314\335\356\377) 32  -file-  32
                              resetfile
                              1 index
                              {
                                    counttomark 1 sub
                                    %%%% pstack -mark-
(\000\021"3DUfw\210\231\252\273\314\335\356\377) 32 -file- -mark- -file-  0
                                    index %% index 0
                                    %%% pstack  au 1e tour pstack -mark-
(\000\021"3DUfw\210\231\252\273\314\335\356\377) 32 -file- -mark- -file-
-file-
                                    counttomark 2 add %%%% 4
                                    index %%% -mark-
(\000\021"3DUfw\210\231\252\273\314\335\356\377) 32 -file- -mark- -file-
-file- 32
                                    4 mul %% 128
                                    string
                                    readstring %%% lit les 128 premiers octets
```

```
de I1
                                    pop %%% pop True
                                    dup %%% duplique la chaine de 128 pour test
avec equal
                                    () eq {pop exit} if %%%% si chaine vide,
exit loop
                              } loop %%%% lit I1 par block de 128 octets (77
block de 128 bytes (154 en hex encoded))
                              counttomark -1 roll %%% met le dernier block en
1er (-file-)
                              counttomark 1 add %%% ... -file- 79
                              1 roll %%% -mark-  (cle) 32 -file- -file- -mark-
data
                        ]
                        %%%pstack -mark-  (cle) 32 -file- -file- [array avec 77
block de data I1]
                        4 1 roll %%% -mark- (cle) [array avec 77 block de data
I1] 32 -file- -file-
                        pop %%% enleve -file-
                        pop %%% enleve -file-
                        pop %%% enleve -32-
                        I2 %%%% -mark- (cle) [array avec 77 block de data I1]
-file I2-
                        0 index %%% -copy file I2 en haut de stack-
                        resetfile %%% consome -file I2-
                        61440 string readstring %%%% lit 61440 octets de I2 dans
string, seulement 2888 hex decoded disponibles
                        pop %%% pop False
                        dup %% duplique la data de I2
                        3 index %%% copy cle en top stack
                        2 2 getinterval %%% prend 3e et 4e caracteres de la clé
                        dup %%% duplique ces 2 caracteres
                        exch %%% echange les memes données (WTF?)
                        dup %% copie une 3e fois les 2 caracteres
                        length %% 2
                        2 index %%% [array I1] - data-I2 - data-I2 - (sous-cle)
-  (sous-cle) - 2-  (sous-cle)
                        length add %%% (sous-cle) -  (sous-cle) - 4
                        string %%% (\000\000\000\000) ???
                        dup dup %%% (\000\000\000\000) * 3
                        4 2 roll %%% (sous-cle) - (\000\000\000\000) -
(\000\000\000\000) - (sous-cle) - (\000\000\000\000)
                        copy %%% ("3) -("3\000\000) - ("3\000\000) - ("3)
                        length %% 2
                        4 -1 roll %%%  ("3\000\000) - ("3\000\000) - 2 - ("3)
                        putinterval %%%% ("3"3)
                        0
                        0 1 1
                        {
```

```
                        pop %%% pop un des 2 0 au dessus
                        2 index %%% ("3"3) - 0 - data-I2
                        length
                  } for %% exécuté 2 fois
                  %%%% data-I2 - data-I2 - ("3"3) - 0 - 2888 - 4
                  exch %%%% data-I2 - data-I2 - ("3"3) - 0 - 4 - 2888
                  1 sub %%%% data-I2 - data-I2 - ("3"3) - 0 - 4 - 2887
                  %%%%%% for(i=0; i<2887; i+=4) %%%%%%
                  %%%% consomme 4 et 2887
                  %%% stack : (cle) - [array I1] - data-I2 - data-I2 -
("3"3) - 0
                  {
                        3
                        copy %%% duplique data-I2 - ("3"3) - 0
                        exch %%%  data-I2 - 0 - ("3"3)
                        length %%% data-I2 - 0 - 4
                        getinterval %%% prend les 4 premiers caracteres de
data-I2
                        2 index %%% data-I2 - ("3"3) - 0 -
4premiers_char_I2 - ("3"3)
                        mark
                              3 1 roll %%% data-I2 - ("3"3) - 0 - mark -
4premiers_char_I2 - ("3"3)
                              0 1
                              3 -1 roll %%% data-I2 - ("3"3) - 0 - mark -
4premiers_char_I2 - 0 - 1 - ("3"3)
                              dup
                              length %%% data-I2 - ("3"3) - 0 - mark -
4premiers_char_I2 - 0 - 1 - ("3"3) - 4
                              1 sub
                              exch %%% data-I2 - ("3"3) - 0 - mark -
4premiers_char_I2 - 0 - 1 - 3 - ("3"3)
                              4 1 roll %%% data-I2 - ("3"3) - 0 - mark -
4premiers_char_I2 - ("3"3) - 0 - 1 - 3
                              %%% for (j=0; j<3; j++)
                              {
                                    dup %%% data-I2 - ("3"3) - 0 - mark -
4premiers_char_I2 - ("3"3) - 0 - 0
                                    3 2 roll %%% data-I2 - ("3"3) - 0 -
mark - 4premiers_char_I2  - 0 - 0 - ("3"3)
                                    dup %%% data-I2 - ("3"3) - 0 - mark -
4premiers_char_I2  - 0 - 0 - ("3"3) - ("3"3)
                                    5 1 roll %%% data-I2 - ("3"3) - 0 -
mark - ("3"3)  - 4premiers_char_I2  - 0 - 0 - ("3"3)
                                    exch %%% data-I2 - ("3"3) - 0 - mark -
("3"3)  - 4premiers_char_I2  - 0 - ("3"3) - 0
                                    get %%%% data-I2 - ("3"3) - 0 - mark -
("3"3)  - 4premiers_char_I2  - 0 - "  (caractere de la clé)
                                    3 1 roll %%%% data-I2 - ("3"3) - 0 -
mark - ("3"3)  - " - 4premiers_char_I2  - 0
                                    exch %%%% data-I2 - ("3"3) - 0 - mark
```

```
- ("3"3)  - " - 0 - 4premiers_char_I2
                                        dup %%%% data-I2 - ("3"3) - 0 - mark -
("3"3)  - " - 0 - 4premiers_char_I2 - 4premiers_char_I2
                                        5 1 roll %%%% data-I2 - ("3"3) - 0 -
mark - 4premiers_char_I2 - ("3"3)  - " - 0 - 4premiers_char_I2
                                        exch %%%% data-I2 - ("3"3) - 0 - mark
- 4premiers_char_I2 - ("3"3)  - " - 4premiers_char_I2 - 0
                                        get %%%% data-I2 - ("3"3) - 0 - mark -
4premiers_char_I2 - ("3"3)  - " - 1e_char_I2
                                        xor %%% real xor
                                        3 1 roll %%% data-I2 - ("3"3) - 0 -
mark - XORed_char- 4premiers_char_I2  - ("3"3)
                                  } for
                                  %%%% XORed_char_1 - XORed_char_2 -
XORed_char_3 - XORed_char_4 - 4premiers_char_I2  - ("3"3)
                                  pop pop %%% enleve ("3"3) et
4premiers_chars_I2
                            ]
                            dup %%%% ("3"3) - 0 - [array 4 char XORed] -
[array 4 char XORed]
                            length %%%% ("3"3) - 0 - [array 4 char XORed] - 4
                            string %%% (\000\000\000\000)
                            0
                            3 -1 roll %%%% ("3"3) - 0 -  (\000\000\000\000) -
0 - [array 4 char XORed] ([231 171 245 255])
                            {

                                  3 -1 roll
                                  dup
                                  4 1 roll
                                  exch
                                  2 index
                                  exch
                                  put
                                  1 add
                            } forall %%% met les 4 bytes dans une string
                            pop %%%  ("3"3) - 0 - string_4_xored
                            4 -1 roll %%%% ("3"3) - 0 - string_4_xored - data-
I2
                            dup %%%% ("3"3) - 0 - string_4_xored - data-I2 -
data-I2


                            5 1 roll %%%% data-I2 - data-I2 - ("3"3) - 0 -
string_4_xored - data-I2
                            3 1 roll %%%% data-I2 - data-I2 - ("3"3) - data-I2
- 0 - string_4_xored
                            dup %%%% data-I2 - data-I2 - ("3"3) - data-I2 - 0
- string_4_xored - string_4_xored
                            4 1 roll %%%% data-I2 - data-I2 - ("3"3) -
string_4_xored - data-I2 - 0 - string_4_xored
```

```
                              putinterval %%% reecrit data-I2 avec
string_4_xored
                              exch %%%% data-I2 - data-I2 - string_4_xored -
("3"3)
                              pop
                        } for
                        0 1 1
                        {
                              pop
                              pop
                        } for

                        %%%%% I2 déchiffré sur la stack à ce moment là

                        cvx
                        exec
                        I3 resetfile 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 {pop 2 index length}

                        for exch 1 sub { 3 copy exch length getinterval 2 index
mark 3 1 roll 0 1 3 -1 roll dup length 1 sub exch 4 1 roll
                        { dup 3 2 roll dup 5 1 roll exch get 3 1 roll exch dup 5
1 roll exch get xor 3 1 roll }

                        for pop pop ] 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 %%%%% counttomark 1 eq
```

```
} bind def



main
clear
quit
```

7.6 Partie 3 : I2 déchiffré

```
20 dict begin /T [ 8#32732522170 8#35061733526 8#4410070333 8#30157347356
8#36537007657 8#10741743052 8#25014043023 8#37521512401 8#15140114330
8#21321173657 8#37777655661 8#21127153676 8#15344010442 8#37546070623
8#24636241616 8#11155004041 8#36607422542 8#30020131500 8#4627455121
8#35155543652 8#32613610135 8#221012123 8#33050363201 8#34764775710
8#4170346746 8#30315603726 8#36465206607 8#10526412355 8#25170764405
8#37473721770 8#14733601331 8#21512446212 8#37776434502 8#20734373201
8#15547260442 8#37571234014 8#24457565104 8#11367547651 8#36656645540
8#27657736160 8#5046677306 8#35250223772 8#32473630205 8#442016405
8#33165150071 8#34666714745 8#3750476370 8#30453053145 8#36412221104
8#10312577627 8#25345021647 8#37444720071 8#14526654703 8#21703146222
8#37773772175 8#20541056721 8#15752077117 8#37613163340 8#24300241424
8#11602010641 8#36724677202 8#27516571065 8#5265751273 8#35341551621 ] def /F [
{ c d /xor b /and d /xor } { b c /xor d /and c /xor } { b c /xor d /xor }
{ d /not b /or c /xor } ] def /R [ 8#7 8#414 8#1021 8#1426 8#2007 8#2414 8#3021
8#3426 8#4007 8#4414 8#5021 8#5426 8#6007 8#6414 8#7021 8#7426 8#405 8#3011
8#5416 8#24 8#2405 8#5011 8#7416 8#2024 8#4405 8#7011 8#1416 8#4024 8#6405
8#1011 8#3416 8#6024 8#2404 8#4013 8#5420 8#7027 8#404 8#2013 8#3420 8#5027
8#6404 8#13 8#1420 8#3027 8#4404 8#6013 8#7420 8#1027 8#6 8#3412 8#7017 8#2425
8#6006 8#1412 8#5017 8#425 8#4006 8#7412 8#3017 8#6425 8#2006 8#5412 8#1017
8#4425 ] def /W 1 31 bitshift 0 gt def /A W { /add } { /ma } ifelse def /t W
{ 1744 } { 1616 } ifelse array def /C 0 def 0 1 63 { /i exch def /r R i get def
/a/b/c/d 4 i 3 and roll [ /d/c/b/a ] { exch def } forall t C [ a F i -4
bitshift get exec a A /x r -8 bitshift /get A T i get A W { 1 32 bitshift 1 sub
/and } if /dup r 31 and /bitshift /exch r 31 and 32 sub /bitshift /or b A
/def ] dup length C add /C exch def putinterval } for 1 1 C 1 sub { dup 1 sub t
exch get /def cvx eq {pop} {t exch 2 copy get cvx put} ifelse } for W /mt t end
cvx bind def not { /ma { 2 copy xor 0 lt { add } { 16#80000000 xor add
16#80000000 xor } ifelse } bind def } { /ma { add 16#0FFFFFFF and } bind def }
ifelse /calc { 20 dict begin /a 8#14721221401 def /b 8#35763325611 def /c
8#23056556376 def /d 8#2014452166 def /x 16 array def /origs exch def /oslen
origs length def /s oslen 72 add 64 idiv 64 mul dup /slen exch def string def s
0 origs putinterval s oslen 16#80 put s slen 8 sub oslen 31 and 3 bitshift put
s slen 7 sub oslen -5 bitshift 255 and put s slen 6 sub oslen -13 bitshift 255
and put 0 64 slen 64 sub { dup 1 exch 63 add { s exch get } for 15 -1 0 { x
exch 6 2 roll 3 { 8 bitshift or } repeat put } for a b c d mt d ma /d exch def
c ma /c exch def b ma /b exch def a ma /a exch def } for 16 string [ [ a b c
d ] { 3 { dup -8 bitshift } repeat } forall ] 0 1 15 { 3 copy dup 3 1 roll get
255 and put pop } for pop end } bind def

%
```

7.7 Partie 3 : I4 déchiffré

```
0 0 0 0 2 2 16 4 sub { 6 index exch 4 getinterval 10240 { 0 0 1 3 { 3 -1 roll
dup 4 1 roll exch get exch 8 bitshift add } for exch pop dup -2 bitshift exch
dup -3 bitshift 1 index -7 bitshift xor exch dup 4 1 roll xor xor 1 and 31
bitshift exch -1 bitshift or 4 string exch 3 -1 0 {3 copy exch 255 and put pop
-8 bitshift} for pop dup <55555555> le {1} { dup <aaaaaaaa> le {-1} {0}
ifelse } ifelse 4 -1 roll add 5 index length add 5 index length mod 3 1 roll 0
0 1 3 { 3 -1 roll dup 4 1 roll exch get exch 8 bitshift add } for exch pop dup
-2 bitshift exch dup -3 bitshift 1 index -7 bitshift xor exch dup 4 1 roll xor
xor 1 and 31 bitshift exch -1 bitshift or 4 string exch 3 -1 0 {3 copy exch 255
and put pop -8 bitshift} for pop dup <55555555> le {1} { dup <aaaaaaaa> le {-1}
{0} ifelse } ifelse 3 -1 roll add 5 index 0 get length 4 idiv add 5 index 0 get
length 4 idiv mod exch 0 0 1 3 { 3 -1 roll dup 4 1 roll exch get exch 8
bitshift add } for exch pop dup -2 bitshift exch dup -3 bitshift 1 index -7
bitshift xor exch dup 4 1 roll xor xor 1 and 31 bitshift exch -1 bitshift or 4
string exch 3 -1 0 {3 copy exch 255 and put pop -8 bitshift} for pop 6 -2 roll
2 copy 8 2 roll get 4 index 4 mul 7 index 5 index get 4 index 4 mul 4 5 copy
dup 4 1 roll getinterval 4 1 roll getinterval exch dup length string 0 3 -1
roll { 3 copy put pop 1 add } forall pop exch 3 -1 roll pop 4 -2 roll 3 -1 roll
putinterval putinterval 5 index 5 index get 4 index 4 mul 4 getinterval 1 index
0 0 1 1 {pop 2 index length} for exch 1 sub { 3 copy exch length getinterval 2
index mark 3 1 roll 0 1 3 -1 roll dup length 1 sub exch 4 1 roll { dup 3 2 roll
dup 5 1 roll exch get 3 1 roll exch dup 5 1 roll exch get xor 3 1 roll } for
pop pop ] 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 pop pop 5 1 roll 2 copy 7 -3 roll pop pop } repeat
pop 4 index 0 1 index { length add } forall string 0 3 2 roll { 3 copy
putinterval length add } forall pop calc I3 16 string readstring pop ne {0 1
1073741823 {pop} for (Key is invalid. Exiting ...\n) error flush quit } if }
for pop pop pop pop (output.bin) (w) file exch 1 index resetfile {1 index exch
writestring} forall closefile

%%%
```